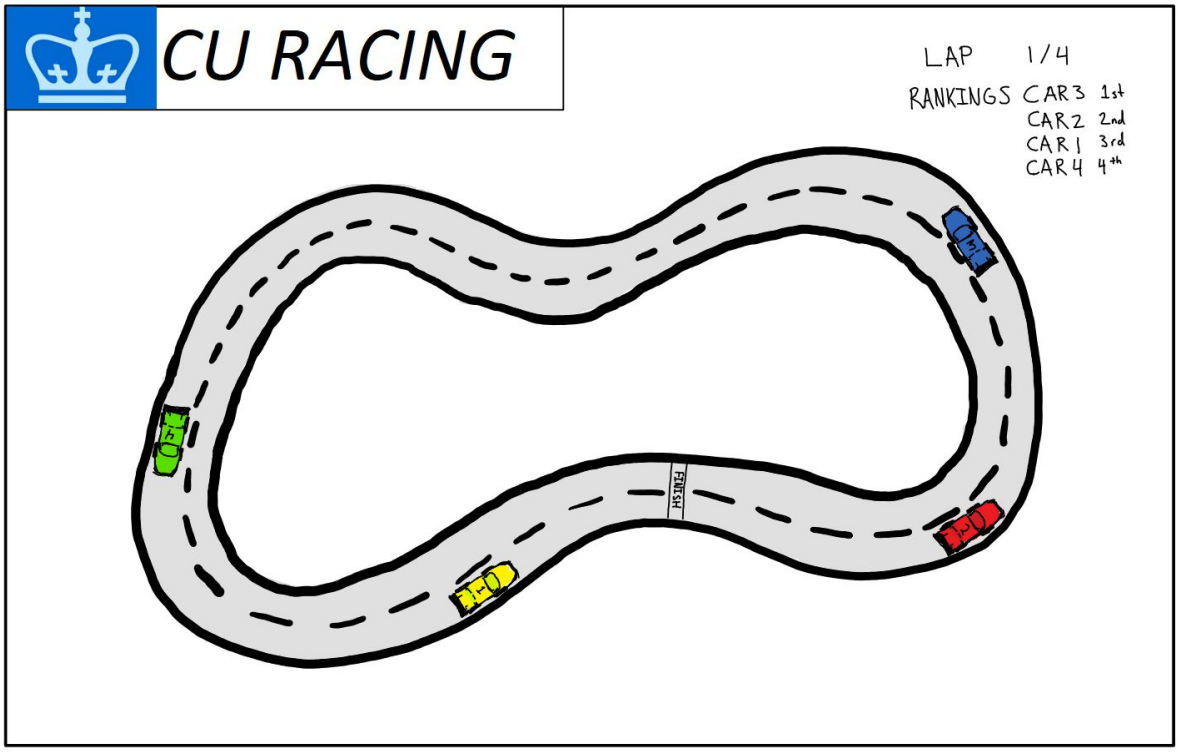
CSEE 4840 Embedded System Design

DESIGN DOCUMENT



Chandan Kanungo (ck2749)

Raghavendra Sirigeri (rs3603)

Robert Kettlewell (rbk2135)

Shikhar Kwatra (sk4094)

Introduction

CU Racing is a retro 2D car racing game. Both single and multiplayer modes will be supported. Single player mode will allow players to refine their skills and compete in time trials. Alternatively, the two player option will enable each gamer to compete with each other using their respective USB controllers.

The game involves three different tracks to choose from. Each track has different friction coefficients and bend radii. Players have to trade off speed and control or risk falling off the track resulting in time penalties. Furthermore, opponent vehicles will be able to bump one another off the track, which introduces some strategy to the play beyond maximizing driving speed. Once all the cars have completed the total number of laps, the race ends and a new track is selected. Once all three tracks are complete, the game will end and return to the main menu.

UX Flow/Game Experience

1) Start Page includes the following options: SINGLE-PLAYER, MULTI-PLAYER, RACE STATISTICS and QUIT
2) Once the multiplayer settings are entered, the game navigates to a new page which displays various TRACKS and CARS to select from.
    - Each TRACK has a different friction coefficients.
    - Each CAR has different STEERING CAPACITY and MAX SPEED.
3) The game then navigates to the actual race which includes the selected TRACK and CAR with game stats PLAYER POSITION, TOTAL TIME ELAPSED, LAP displayed on the top right corner.
    - Skids and rotations based on the track selected based on the game engine (*cuphysics*)
    - Dynamic sound generation based on position on the track, accelerating speed and collision events with a continuous background tone always playing.

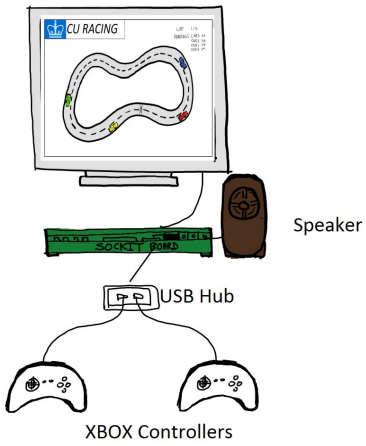


Figure 1 -Standalone SoCKit board for local play.

System Level Architecture

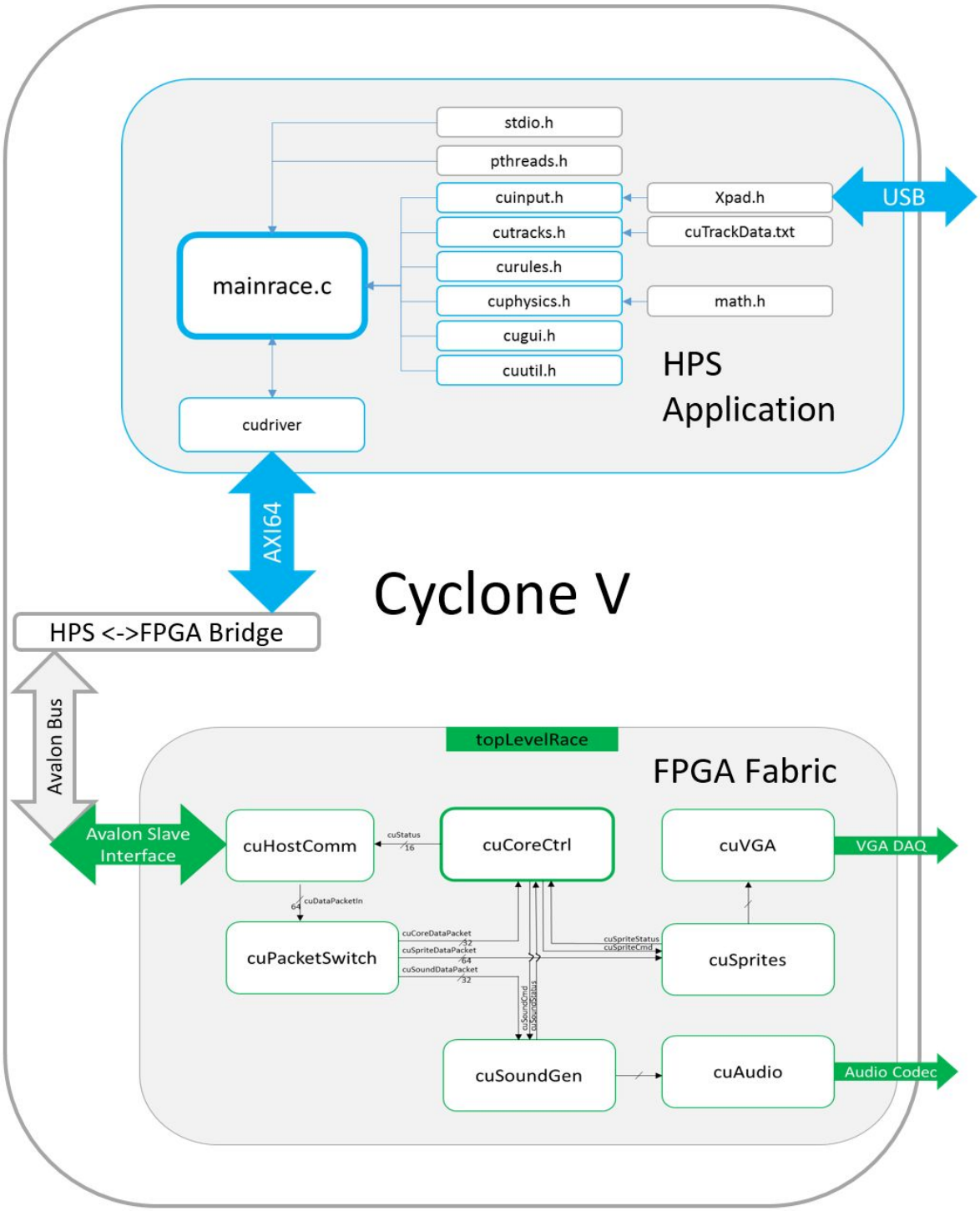


Figure 2 - High-level architectural organization and bus data flow for CU Racing on the Cyclone V SoC.

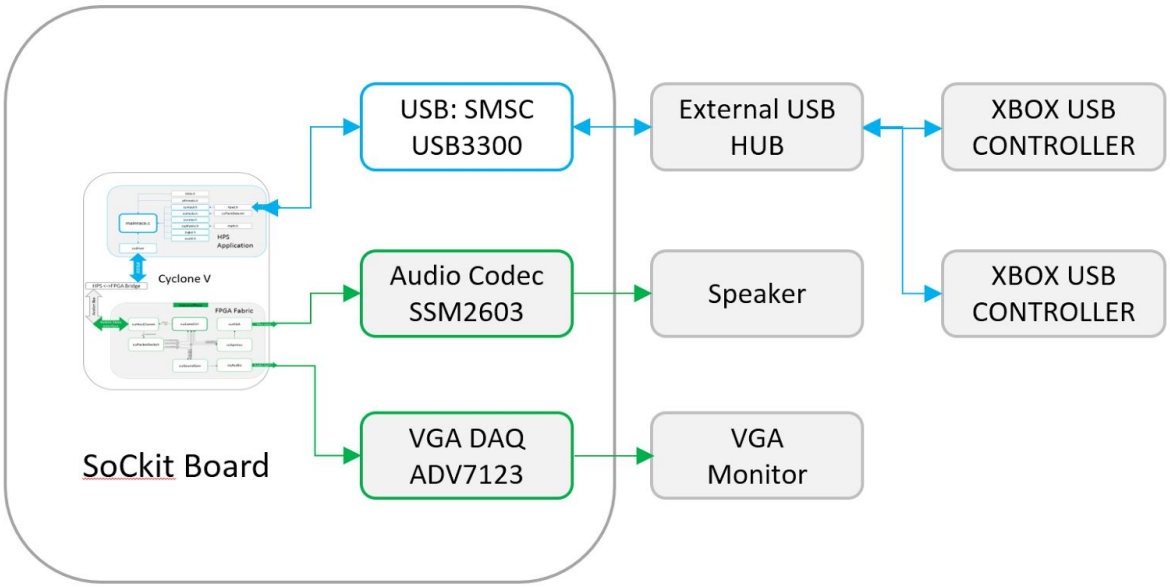


Figure 3 - Block level diagram of CU Racing game system.

Software Architecture

HPS Application Level Hierarchy:

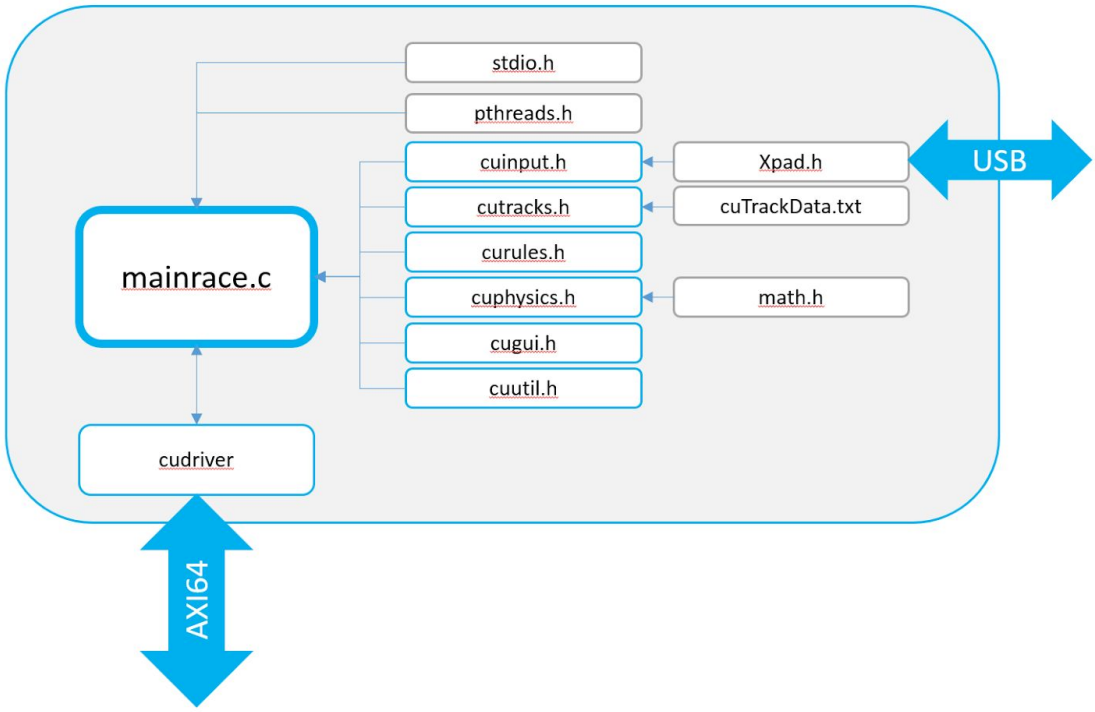


Figure 4 - CU racing library organization and external dataflow

Filename	Description
mainrace.c	Main game engine which imports all CU Racing library dependencies, handles user inputs, controls and maintains game state, and communicates to the FPGA the sounds and sprite location updates.
cuinput.h	Thin interface to the xpad driver for the Xbox 360 controllers with the restricted set of buttons and joystick features enabled.
cutracks.h	Collection of helper functions that define the unique features of each track: unique name identifier, sounds, sprites, physics constants (coefficient of friction: on/off pavement) for each track. Additionally, a geometrical representation of the track will be accessible.
curules.h	Contains information regarding the race car attributes, time penalties for crashes, number of laps per track, time limits, and the game
cuphysics.h	Exposes a function which, given past and current state, returns new car position with a velocity and angle of rotation, unique to each race car's attributes and track constants.
cusounds.h	Contains helper functions that determines race noise events given the car locations, speeds and track type.
cugui.h	Contains the state machine for the acceptable GUI transitions
cuutil.h	Timers, race stat utilities, and miscellaneous helper functions will be contained in this file
cudriver.h	Interface that wraps the character device driver for the Host-to-FPGA communication. This file contains additional functions to packetize information into U64 words to be sent to the FPGA.

Table 1: CU Racing library file descriptions

FPGA Toplevel Hierarchy:

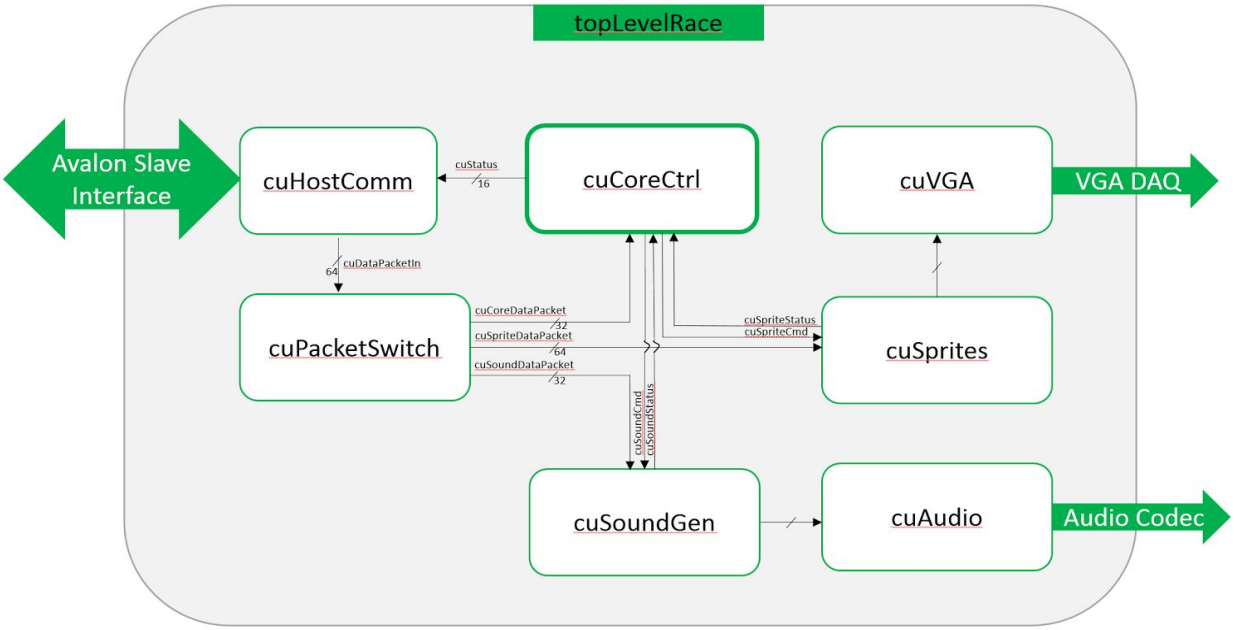


Figure 5 - Top-level fpga module organization and external port dataflow

Filename	Description
topLevelRace.sv	Top level file which instantiates and connects up the modules described below.
cuHostComm.sv	Main communication data interface to the HPS and the FPGA. Supports U64 data writes and U64 status register reads.
cuPacketSwitch.sv	Redirects incoming data to the respective control modules: cuCoreCtrl, cuSprites, cuAudio.
cuSprites.sv	Sprite controller architecture based on the TMS9918 with extensions for flipping images.
cuSoundGen.sv	Sound generation capability with multiple pure tone, random noise, and distortion wave generators with enveloping and mixing features.
cuAudio.sv	Controls the audio outputs to the onboard audio codec.
cuVGA.sv	VGA controller which receives its display data from cuSprites

Table 2: CU Racing - System Verilog Hierarchy Descriptions

SPRITE REGISTERS (0x37-0x20)	CU_SPRITE_VDP	0x34
	CU_SPRITE_REG_7_4	0x30
	CU_SPRITE_REG_3_0	0x2C
	CU_SPRITE_CONFIG	0x28
	CU_SPRITE_CMD	0x24
	CU_SPRITE_STATUS	0x20
SOUND REGISTERS (0x1F-0x14)	CU_SOUND_CONFIG	0x1C
	CU_SOUND_CMD	0x18
	CU_SOUND_STATUS	0x14
CORE REGISTERS (0x13-0x00)	CU_CORE_RESERVED	0x10
	CU_CORE_RESERVED	0x0C
	CU_CORE_SCRATCH	0x08
	CU_CORE_CMD	0x04
	CU_CORE_STATUS	0x00

Figure 6 - Proposed Register Map for CU Racing FPGA hardware architecture.

Packet Transfer Discussion

Our proposed approach is to have the *cuPacketSwitch* receive U64 packets from the *cuHostComm* module which contains all the necessary signaling to act as an Avalon Slave interface. These U64 packets will have an 8 bit address located at the MSB which will contain the destination register address. This destination register address range will allow the packet switch to redirect the packet to its appropriate endpoint. The remainder of the packet will consist of a data payload which will contain the 32 bit register contents.

Race Modeling

The CU physics engine will calculate the position, velocity, acceleration, angular rotation of each race car as it traverses the virtual tracks. Car bumps and crashes will also be modeled inelastically and allow for kinetic energy transfer between interacting cars. The model will assume all cars are front wheel drive and that the center of mass of the car is fixed at the origin shown below in Figure 7.

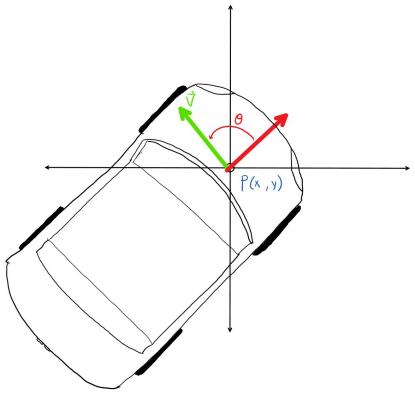


Figure 7 - Dynamic model constants (posX, posY), v , θ which the physics engine calculates for all cars in a given race.

If a player attempts to steer through a bend of the track at a velocity which is sufficiently high, the normal vector of the front bumper and the velocity vector of the car will diverge (Figure 7: red θ). If this angular rotation reaches a critical level, the race car will lose control and slow down - incurring a time penalty for the player.

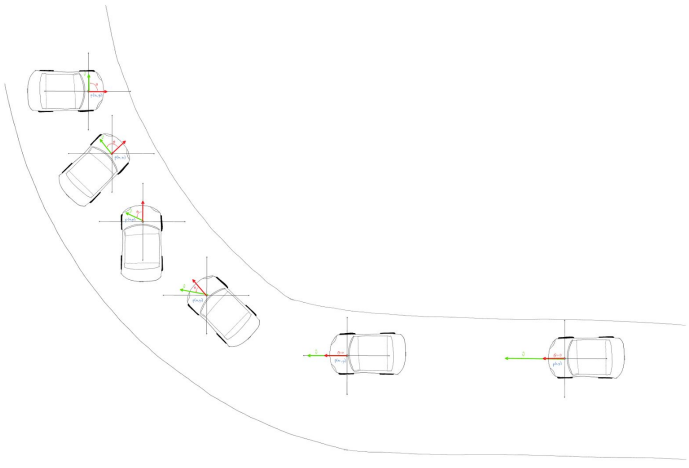


Figure 7 - Race car turning through a track bend at a velocity that is too great, resulting in drift and loss of control.

Race Car Sprites

To support smooth rotation of the race cars as they navigate through the track, there will be 16 sprites with an angle increment of approximately 11.5 degrees (blue box : Figure 8). These sprites will be able to be mirrored to complete 360 degrees of rotation. Our plan is to start with this approach and then if we need more fine grain rotation we can cut the angle increment value in half and store the rotations through 90 degrees.

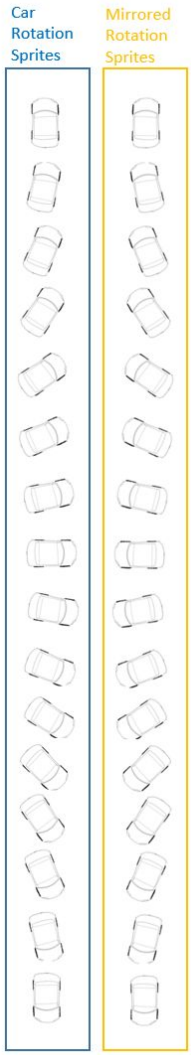
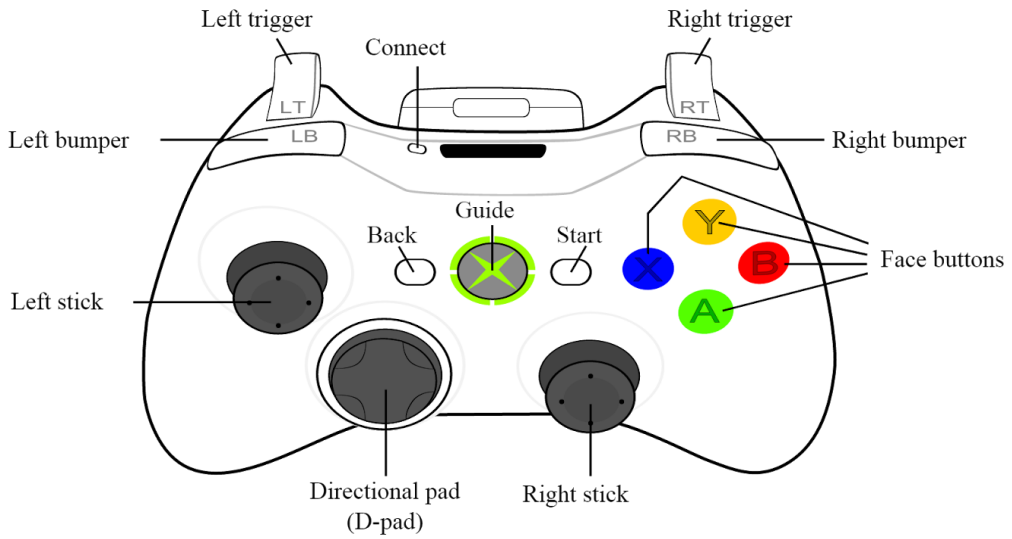


Figure 8 - Race car sprite rotations and mirroring.

USB Controller

The XBOX Controller is used for playing the racing game. The Left Analog Stick is used for steering, the Right trigger for accelerating and Left Trigger for braking. Buttons for PAUSE and RESUME are configured as well.



The Xbox Controller has two main components, USB gamepad and a two port USB hub.
The hub has 3 ports[1].

Port1 -------> Gamepad
Port2 -------> Memory Unit Slot
Port3 -------> Memory Unit Slot

The USB cable and the memory unit connector carries five signals, four of which are standard USB signals and a clock signal[1].
The Linux kernel includes xpad.c library which supports various XBOX Controllers. The HID Descriptor Table for the controller is given below[2]

offset	data
+0	0x00
+1	0x14 (size of the whole report)
+2	digital buttons
	bit0 D-pad up
	bit1 D-pad down
	bit2 D-pad left
	bit3 D-pad right
	bit4 start button
	bit5 back button
	bit6 left stick press
	bit7 right stick press

+3	0x00 (reserved for more digital buttons?)
+4	A button (*)
+5	B button (*)
+6	X button (*)
+7	Y button (*)
+8	black button (*)
+9	white button (*)
+10	L trigger (*)
+11	R trigger (*)
+12	left stick x (**)
+14	left stick y (**)
+16	right stick x (**)
+18	right stick y (**)

- (*) unsigned 8-bit
- (**) signed 16-bit, little-endian, north/east positive

Audio Features

The type of generated sounds will be as follows:

1. Count down chimes for the start of the race
2. Engine acceleration/de-acceleration noises
3. Braking noise
4. Skidding noise
5. Tires off track noise
6. Crash tone when the car crashes
7. Simple arcade style piano music for end of race

The tones are generated in the *cuSoundGen* module on the fpga using a mixture of waveform generating functions. These functions will be optimized to limit their footprint on the FPGA fabric as much as possible and alleviate the need to constantly stream audio data from the HPS.

As previously described, sounds will vary based on the track and car selections. *Mainrace.c* will obtain constants which will be sent to the FPGA and directed to the *cuSoundGen* module for amplitude, frequency and waveform mixing adjustments. These tone functions are invoked at specific situations during the race.

The audio output will be performed via a high-quality 24-bit audio via the Analog Devices SSM2603 audio CODEC (Encoder/Decoder). This chip supports microphone-in, line-in, and line-out ports, with a sample rate adjustable from 8kHz to 96kHz. The I2C bus interface is used to control the SSM2603. The pin diagram is as shown below[4]:

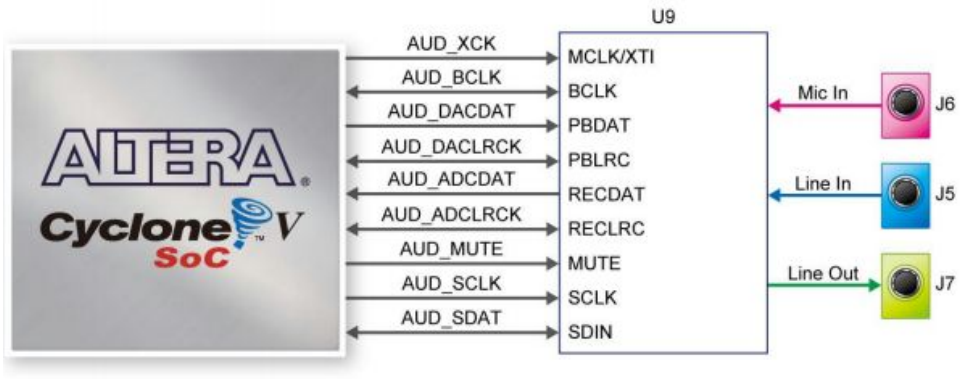


Table 3-14 Pin Assignments for Audio CODEC

Signal Name	FPGA Pin No.	Description	I/O Standard
AUD_ADCLRCK	PIN_AG30	Audio CODEC ADC LR Clock	3.3V
AUD_ADCDAT	PIN_AC27	Audio CODEC ADC Data	3.3V
AUD_DACLRCK	PIN_AH4	Audio CODEC DAC LR Clock	3.3V
AUD_DACDAT	PIN_AG3	Audio CODEC DAC Data	3.3V
AUD_XCK	PIN_AC9	Audio CODEC Chip Clock	3.3V
AUD_BCLK	PIN_AE7	Audio CODEC Bit-Stream Clock	3.3V
AUD_I2C_SCLK	PIN_AH30	I2C Clock	3.3V
AUD_I2C_SDAT	PIN_AF30	I2C Data	3.3V
AUD_MUTE	PIN_AD26	DAC Output Mute, Active Low	3.3V

The I2C interface can be established using selectable control registers. The write and read sequences are as given below[3]

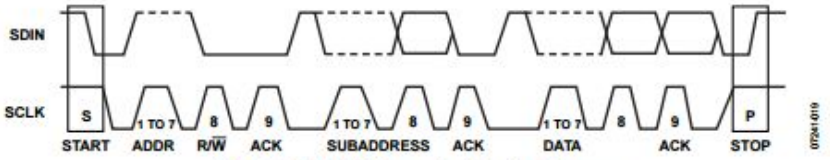


Figure 28. 2-Wire I²C Generalized Clocking Diagram

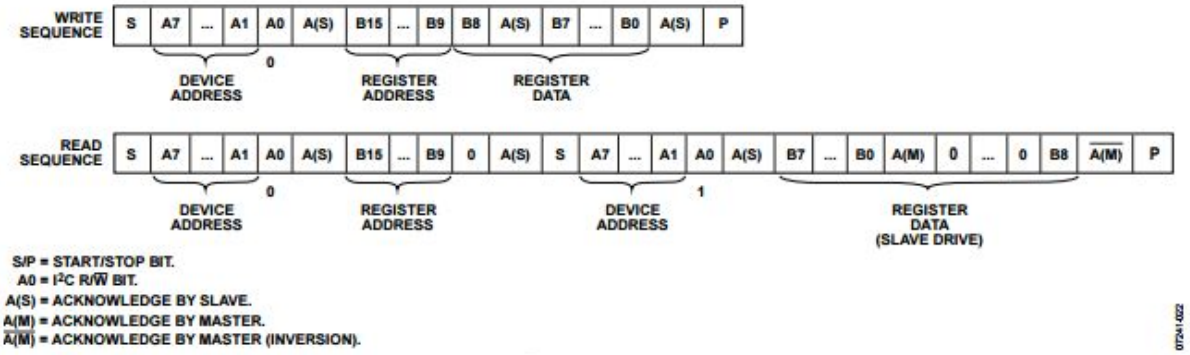


Figure 29. I²C Write and Read Sequences

Each data word is 16bits long, MSB first.

BIT15-BIT9 ---------> Register Map Address
BIT8-BIT0 ---------> Register Data
SDIN ---------> Serial Control Data Word
SCLK ---------> Clock for Serial Transmission
CSB ---------> Chip Select Pin to select the slave address

If the CSB pin is set to 0, the address selected is 0011010; if 1, the address is 0011011[3].

Development Milestones

Milestone 1 (Thursday March 31)

- Interface XBOX controller via Xpad driver successfully on SoCKit board
  - Wrap interface in *cuinput.c*
- Implement alpha *cuSoundGen* and *cuAudio*
- Implement alpha *cuHostComm* and *cuDriver* for U64 packet transfers

Milestone 2 (Tuesday April 12)

- Integrate alpha version of *topLevelRace*
  - Complete alpha *cuPacketSwitch*, *cuCoreCtrl*, *cuSprites*, *cuVGA*
- Complete alpha *mainrace.c* with its dependent library files
- Preliminary artwork
- Preliminary GUI complete
- Begin alpha testing
  - Single player mode should be running fairly smoothly,

Milestone 3 (Tuesday April 26)

- Complete beta version of CU Racing C library
- Complete beta version of *topLevelRace*
- Refine artwork
- Refine UX
- Begin beta testing

References

1. http://euc.jp/periphs/xbox-controller.ja.html
2. http://eleccelerator.com/tutorial-about-usb-hid-report-descriptors/
3. http://www.analog.com/media/en/technical-documentation/data-sheets/SSM2603.pdf
4. http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=816&FID=a9e8cb474881606fa975d2420a309fb6