FPGArio Kart Preliminary Report

Kiersten Mitzel
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
kmitzel@mit.edu

Ragulan Sivakumar
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
rskumar@mit.edu

***Abstract*—We present a design for Ethernet-enabled Mario Kart on FPGA systems. In this design, two FPGA's are connected over an Ethernet cable to simultaneously play a game of Nintendo's Mario Kart. The system primarily functions through an Ethernet Module, a Graphics Module, and a Game Module to bring together technicality with functionality.**

***Index Terms*—Digital systems, Field programmable gate array, Ethernet, Framerate, Sprite, BRAM**

I. System Design

As seen in Figure 7, the system is composed of three primary modules: Ethernet, Graphics, and Game. The game module keeps track and calculates the player and opponent's position, direction, and game status. These values are then sent both to the Ethernet and Graphics module so that they can both be transmitted and updated in the graphics. These modules run simultaneously with one another to determine the state of the game overall.

Key design choices not noted in the diagram were that the entire system runs on both a 50MHZ clock for the ethernet and a 65MHZ clock for the graphics. This meant we tackled clock crossing by implementing BRAMs to synchronize the values going in and out of the ethernet transmission.

The same code is run on both FPGA's, but the starting positions and cart identifiers were hard coded for each racer to prevent any initial overlap or glitches. In addition, the collision effects were both differed with the carts to prevent them from bouncing off of each other in the same direction.

II. Ethernet (Kiersten)

The Ethernet consists of 2 primary modules: receive and transmit. As seen in Figure 1, when the transmit module is enabled by hcount and vcount, a message is sent over by dibits to the connected FPGA. Once as the transmit message is seen by the other FPGA, the receive module in the connected FPGA is enabled, and it validates and parses the message to get only the bits we use.

A message is formatted through a 512 bit (64 bytes) buffer that is constantly fed values from the game module. The buffer message consists of the structure as seen in 6.205's Ethernet Lab [1], and when hcount and vcount are equal to 1024 and 768, we send what's in buffer by dibits. The specific values for hcount and vcount were chosen because that's the first pixel that's inside the hsync and v sync interval, as seen in 6.205's Ethernet Lab [1], meaning we would have sufficient

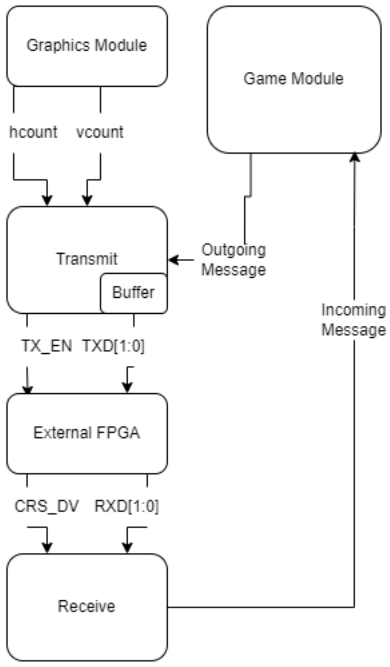

Fig. 1. Ethernet System Diagram

clock cycles after to send the message, process the message, and update values before the frame had to be updated.

Since we chose to only send the given FPGA player's x position, y position, direction, game status, and reset input, we opted to have a hard set length and definition for the message to be 44 bits. However, an Ethernet message has a minimum size requirement of 512 bits (64 bytes), so we ended up padding a lot of 0's within the data we would be sending. As a simplification, since we know that there will always only be two FPGA's sending back and forth to one another by a defined two way connection, messages are sent to the broadcast destination address, which is 48 bits of 1's, so that the two FPGA's will never miss an incoming message.

Since the two FPGA's are sending the exact same size message and contain the same Ethernet modules, this causes the two FPGA's to complete transmitting and receiving messages at the same time, meaning they will be in sync with one another. This was proven so during test benching, as well as proven on hardware as the game play successfully transmitted locations back and forth.

A. *Transmit*

The Transmit module consists of a state machine, that when enabled starts the transmit process. It's signaled to go high at the end of every frame that's been loaded. The transmit module then grabs the loaded message from the message buffer, which was formatted in the previous frame. The module then sends along the message by setting output TX_EN to high and loads the message into TXD[1:0] until the full message has been sent, where TX_EN is set low again.

B. *Receive*

The Receive module works alongside the Transmit module. Once TX_EN goes high, the message is sent over the wire and receive is enabled when CRS_DV is high. Once this happens, the message is read through RXD[1:0] and parsed until we only get the 44 bits of information needed. This module is similar to the work from 6.205 Ethernet Lab [1].

III. Graphics (Ragulan)

A. *Sprite and Track Storage*

Image Sprite Sizes:

- 32 $\times$ 32 images (III.B)

Track Storage:

- 16 $\times$ 16 memory file with each entry corresponding to an road, sand, or grass image sprites. These image sprites together form the base of the track.
- 16 $\times$ 16 memory file with each entry corresponding to obstacle image sprites. These image sprites are placed on top of the track base. The example in III.F shows an oil spill overlaid on a road.

Sprite BROMs:

- Image-to-Palette BROM
- Palette-to-Pixel BROM

Image sprites for things, such as roads, sand, and grass, will be saved in two sizes above via the two BROM files noted. The Image-to-Pixel BROM has depth equal to the number of pixels in the sprite. Each entry corresponds to a palette location in the Palette-to-Pixel BROM where the corresponding pixel data is stored. Thus, after indexing into that location in the Palette-to-Pixel, we can retrieve the pixel.

The track base and track obstacles are stored as two distinct 16 $\times$ 16 memory files, in which each entry corresponds to a certain image sprite already defined. Altogether, our memory consumption is vastly reduced by abstracting the component sprites of a track into discrete modules, for if we chose to store tracks explicitly and include all sprite data in the track, identical sprites would be saved many times over and waste memory.

Note that the track is designated as 2048 $\times$ 2048 to give granularity in our position updates.

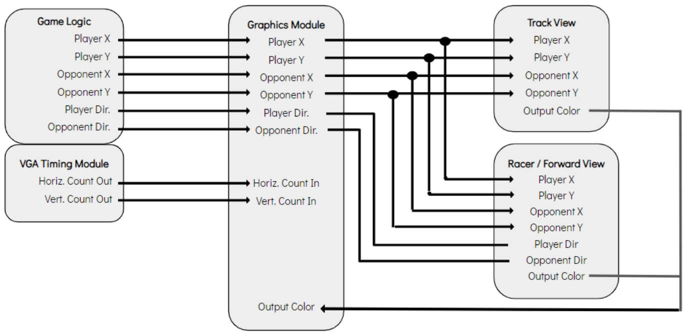

Fig. 2. Graphics System Diagram

B. *Graphics Module*

Inputs

• player-x,	player-y
• opponent-x,	opponent-y
• hcount-in,	vcount-in
• player-dir	opponent-dir

Outputs

- pixel

Graphics as a whole are controlled within this module. The following submodules determine the pixel output for different regions of the screen, and those results are all filtered back to this module where appropriate logic and pipelining is done to output the correct pixel to the VGA monitor.

The overall throughout of the graphics module is 1, and the latency is 8 clock cycles.

C. *Top-Down Track View*

Of the same inputs as the graphics module minus $direction$, this module controls the the upper left 512 $\times$ 512 pixels. It renders an image of the entire course and where the players are located within it, thereby giving players an absolute frame-of-reference as to where there are in the course.

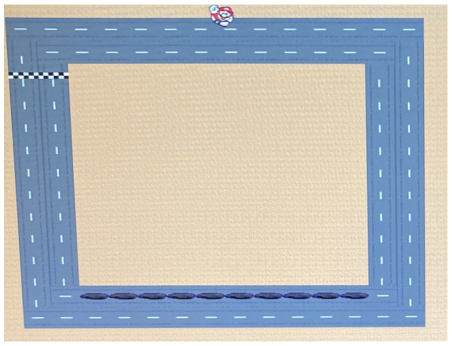

Fig. 3. An example of the Top-Down Track View

To simplify the following logic, any discussion of obstacles will be omitted until section III.F. Also, any discussion of the track memory file refers to the track base memory file.

The track memory file is first indexed appropriately by our $(hcount_{in}, vcount_{in})$ to find which 32 $\times$ 32 sprite we should look in to find the correct pixel to output. We then index said sprite in the manner detailed above to find the specific pixel p that we should display . However, if our $(hcount_{in}, vcount_{in})$ corresponds to a location in the track that is in the player or the opponent, which we determine by seeing if it's distance from $(person_x, person_y)$, the person's center is less than $|(64, 64)|$, then we display the appropriate pixel p for them instead. Note that collision logic prevents players and opponents from occupying the same location in the track, so having players and opponents at the same spot doesn't need to be analyzed.

D. Top-Down Racer View

With inputs that are the same as for the graphics module, this module controls the upper right 512×384 pixels. Players will have an aerial view of their nearby vicinity scaled up 4-fold in both x and y direction from the track view. Moreover, this view is also in the direction that the player is driving, so if a tree is to the right of them in the overall track but they're about to drive into it, it will appear directly above them in the racer view.

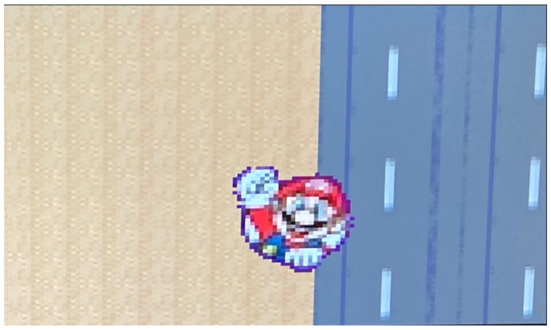
Fig. 4. An example of the Top-Down Racer View

A player's cart is centered about $hcount_{in}$ of 767 and $vcount_{in}$ of 255. This dictates the 128×128 bubble surrounding that point. Everything else should be in the perspective of whichever direction the player is driving, so a rotation matrix involving sines and cosines is needed to compute original track location of the pixel that should be displayed at a certain point in the racer's perspective. The general outline of the algorithm is as follows:

- First, we want to calculate our relative distance vector from the player in the racer view ($racerRDV$). This is
$$\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} 767 - hcount \\ 255 - vcount \end{pmatrix}$$
.
- We want to get the $trackRDV$ in the overall 2048×2048 game grid. Thus, we want to rotate our $racerRDV$ appropriately. Given our angle θ that the player is facing, we multiply our $racerRDV$ by our rotation matrix
$$\begin{pmatrix} \sin\theta & -\cos\theta \\ \cos\theta & \sin\theta \end{pmatrix}$$
Note that this is not the standard rotation matrix: $vcount$ decreases as you go up the screen, whereas in a normal plane, y would increase as you go up, so a different matrix is used.
- To make our $trackRDV$ absolute, we add our player position to it. This gives our track lookup location, which we use in the same manner as in the track view.

As for how trigonometric values are calculated, the cosines for degree values 0 to 359 are stored in a BROM, and sines are extracted from the same BROM since sine and cosine are offset by 90 degrees. They are stored with precision to the $\frac{1}{512}^{th}$ so as to enable proper granularity in the track lookup location.

To determine if we should display a pixel from our opponent, we check if the track lookup location is within the 128×128 bubble surrounding our opponent's center. If so, we grab the appropriate pixel based on how far they are from the opponent's center.

E. Forward View

Unlike the Track and Racer Views, which give aerial views from high above the track, the forward view give players a view from the height at which they are driving of their nearby vicinity. This module controls the bottom right 512×256 pixels.

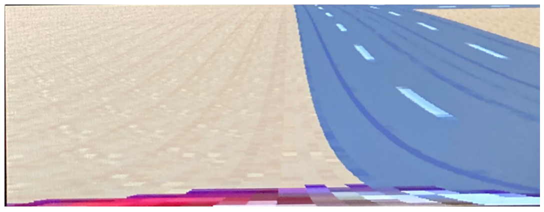
Fig. 5. An example of the Forward View

This module follows similarly to the racer view, but we must determine how to scale distance appropriately before doing the rotation and adding in player position to get the track lookup location. A novel approximation based on the fact that objects twice as far away from a player appear twice as small was used to calculate the track lookup location. The exact formula used for $forwardRDV$ was

$$\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} \frac{hcount-767}{256} \times (1056 - 128 \times \log vcount) \\ 1056 - 128 \times \log vcount \end{pmatrix}$$

Note that objects are closer to the player lower in the forward view, so a larger vcount has our Δy smaller as desired. Moreover, the logarithm in base 2 ensures the exponential scaling up in distance desired. The numbers 1056 and 128 were chosen arbitrarily to make the earliest pixel seen be 32 in front of the player and the farthest about half a track side away.

Note that in the formula for Δx, all points a given Δy away were considered the same distance from a player. This is negligible for large Δy. However, for small Δy, this gives undesired curves to straight lines. A more rigorous treatment of Δx in a similar manner to Δy could have eliminated this.

A BROM of the logarithms base 2 of the numbers 1 to 256 is stored so as to find the scaling factor of how far a point is away from the player. The precision is once again to the $\frac{1}{512}^{th}$.

Everything else follows exactly the same as the racer view from the second bullet point on.

F. Displaying Obstacles

While indexing into the track base memory file and then the corresponding sprite, we also index into the track obstacle memory file and then its corresponding sprite. If the entry in the track obstacle memory file is 0, then there is no obstacle. Otherwise, there is an obstacle, and we'll want to display it. However, obstacles generally do not take up the full 32 by 32 pixels. Let the 32 by 32 obstacle sprite without the obstacle be denoted as the $canvas$ region. The underlying track base should be displayed in the $canvas$ region. To do this, both track files and sprites are indexed in parallel. The $canvas$ area of an obstacle sprite was filled with the color $12'h406$, so if the obstacle pixel was $12'h406$, we used the underlying track base pixel, whereas otherwise we used the obstacle pixel. Thus, obstacles were incorporated.

Note that obstacles is a loose term, for the finish line is also incorporated in this manner.

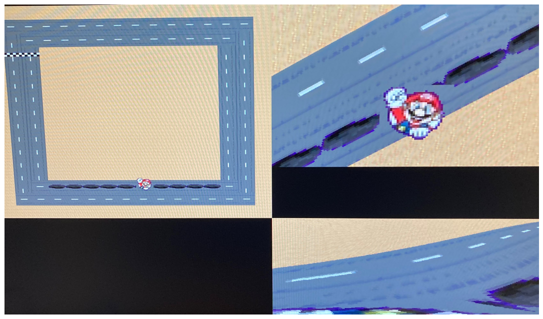

Fig. 6. An example all three views with an oil spill overlayed on the track

IV. Game Logic (Kiersten)

The Game Logic is driven by a state machine that checks the status of both FPGA's once every frame. This includes checking where both player's are, whether or not someone has won, and whether or not someone has reset the game. The state machine also calculates the players next position based off of their directions, and sends this to the ethernet module once a frame.

The game starts when the FPGA's first load up, or when one of the players hits btnc to reset the game. Key controls of the system include:

- BTNC: Used to reset the game logic and start new rounds.
- BTNU: Used to reset the ethernet tranmission.
- SW[15]: Used to turn player's cart left.
- SW[0]: Used to turn player's cart right.
- LED[15:0] Used to display opponent's position, primarily used to make sure ethernet is transmitting.

A. Win Conditions

Each FPGA internally keeps track of how many laps their respective kart has gone around the track. To win a game, a player must complete 3 laps before the opponent does. Laps are only considered complete and incremented if the cart has previously hit all 3 corners before hitting the corner that contains the finish line, as tracked by variables corner1, corner2, corner3, and corner4. This lap count is held in a variable called lap_count and whether or not lap_count is equal to 3 determines game_status, which defines whether or not the game should be running. At the end of each frame, this game_status is sent over the Ethernet to the other FPGA, to signify to the opponent whether or not they have lost the game.

When one FPGA completes three laps, their screen freezes, but the opponent's does not. In the classic Mario Kart, the game still continues to run despite one player crossing the finish line first, so as a choice we continued to let the opponent race. When the opponent finishes racing, they are sent to the sideline indicating they have lost but still completed all laps.

If neither player completes three laps, the game will continue to run until one has done so.

B. Collisions

At the end of each frame, a message is sent and received by each FPGA. Immediately after, there is a check in the both game modules to see if the two carts positions will overlap. If this is the case, the FPGA's forcibly offsets both carts x and y positions by 64 pixels to simulate a bounce between the two. In classic Mario Kart, the physics sends carts flying across the screen, and this is simulated in our bounce mechanics.

C. Reset

Resets happen when one or both players hit BTNC, causing both FPGA's to be reset for the next game at the same time. The reset sets all variables back to their initial conditions, including the players starting positions. After doing so, the system then starts a new game immediately.

D. Design Evaluation

Throughout the project we were sure to check our individual modules to ensure there was no slack or multidriven errors, for this could cause problems when sending and receiving data. We were successful at doing this individually. However when combining modules together, we ended up with a final WNS of -2.636 and TNS of -2284.853. While this would generally be considered a bad statistic, we found it didn't impact the functionality of our system. We attempted to bridge the clocks via registers and BRAMs and still had a negative WNS, so we suspect the negative WNS is coming about because of when we're transferring data from the 65 MHz clock to the 50 MHz clock, for more would be going in than could be processed theoretically if we were always transferring data. However, we only transfer minute amounts at select times in each frame. Moreover, given how we found that if we removed the clock crossing and used set values, the game logic and graphics had a slack of 1.8, we suspect that the majority of the our WNS slack came from clock crossing. Thus, we deem it negligible for our project.

As for our BRAM and LUT usage, it's 8.15% and 3.82%, respectively. We were very conservative in our memory usage, so I don't think it could be optimized further, for the vast majority of that is memory file storage for the different sprites. However, we could have pushed our resource usage a lot further with more granular graphics for the racer and forward view alongside multitudes of tracks and still been well within our limits.

As for latency and throughout, it's 1 and 8 clock cycles, respectively, for the graphics module, and it's 261 and 517 clock cycles for the Ethernet module. With graphics, there is the significant overhead of the blanking periods where nothing is done, so we took advantage of those by employing all of Ethernet, transmitting and receiving, within that duration.

V. Retrospective

A. *Project Commitments and Goals*

At the start we made the bare commitment to deliver basic Mario Kart functions, such as traditional gameplay and collisions, ethernet transmission and receiving, having a top-down track view, and having the game run on two FPGA's at once. We did accomplish all of these as shown throughout the rest of the report.

The goals we outlined were to create a top-down track view and top-down racer view, which were likewise accomplished alongside the forward view, which was our stretch.

B. *Graphics Reflection*

If we were to redo graphics, it would be useful to create the three views in parallel. There was so much overlap in the logic between the modules, and every time whenever a new optimization technique was discovered for a new view, the old ones would need to be updated to have the same optimization. This made for a lot of time wasted redoing old logic.

One thing that proved key was sketching out internal diagrams of variable flow throughout time within a module. Without those diagrams, pipelining would have been a nightmare.

C. *Game Logic and Ethernet Reflection*

If we were to redo Ethernet, it would be helpful to properly set and define the MAC addresses for both FPGA's. Although using broadcast addresses for the project works, there is a chance that a message can be missed due to incorrect addresses.

If we were to redo Game Logic, it would be useful to spend a bit more time thinking on collision logic. Collisions were one of the last features we implemented, and the logic we were initially going to use proved to not be functional in reality. It would've been helpful to have backup logic in mind to implement a proper bounce, rather than the solution of doing a forced offset.

In addition, with more time I would've focused on implementing items into the game. With Ethernet working, it would've been possible to send effects back and forth to the other player, such as slowing down their speed or making them teleport across the map as item effects. It also may have been possible to coordinate graphics effects along to.

VI. Code Base

The code base can be found here.

VII. Credits

Credits for Ethernet and the game logic go to Kiersten, whereas credits for graphics goes to Ragulan. Special thanks to Fischer, Jay, and Joe for being great course staff and always helping us debug, to Pleng for helping to debug and having a VGA to HMDI cable that enabled graphics to be tested outside of lab, and to all of our classmates for keeping us sane during labs.

References

[1] As previously built and tested during Fall 2022's 6.205 Introductory Digitial Deisgn System's Lab 5, which can be found at fpga.mit.edu

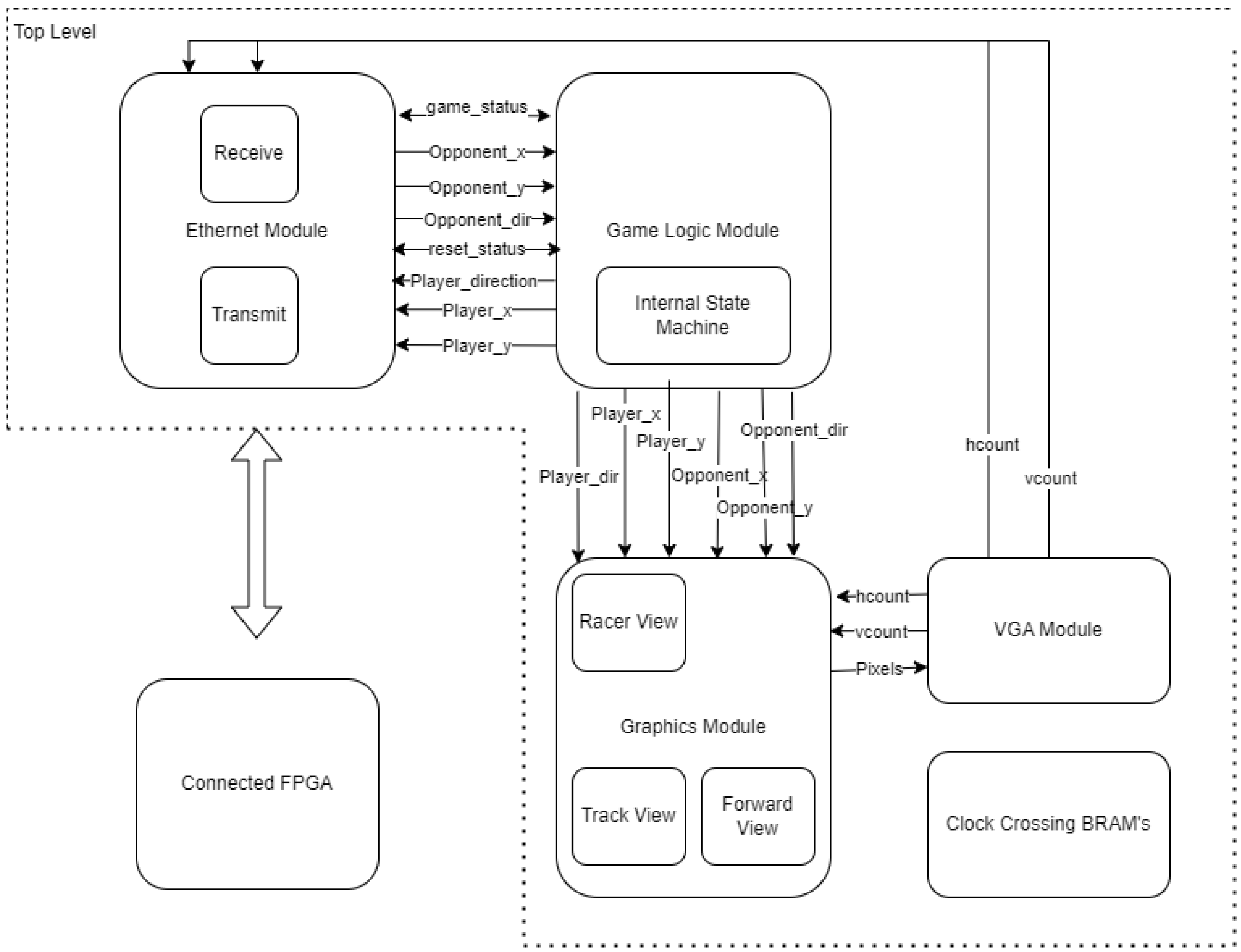

Fig. 7. Overall System Diagram