Overcooked: Food-programmable Gate Array

Digital Systems Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Julia Arnold
jul@mit.edu

Kye Burchard
kyeb@mit.edu

Lacthu Vu
lacthu@mit.edu

***Abstract*—Overcooked is a traditionally multiplayer cooking game where players must prepare and serve dishes under time pressure. During a round, the players are presented with orders which must be completed within a short time window, otherwise they lose points. Food that is cooked too long catches on fire. Our project implements a streamlined version of the original video game. The greatest challenges we faced were. Future work includes adding more levels and foods, a leaderboard saved in the server and more dynamic fire.**

I. Introduction

In order to create Overcooked as an FPGA game, the project was split into three major groups which have been tasked to the three team members respectively: game state & logic, graphics and multiplayer communications. Each were written and tested fairly independently, then integrated fully together in the final weeks.

II. Structural Description of the Top Level Module (All)

The top level module brings everything together to determine player actions, sync between players and then make that appear on each screen. First we define the 25.175 MHz clock needed for the graphics as well as keeping the 100 MHz clock for serial communications. We define all the player inputs available. This includes:

- Player ID: sw[1:0]
- Number of players: sw [3:2]
- Reset: sw[4]
- Pause: sw[14]
- Carry: sw[15]
- Player movement/direction: Up, down, left, right buttons
- Chop: Center button

In order for the game to work in multiplayer, each player has to input the correct switches for player ID and number of players, and make sure player ID is unique among participants. Player 0 requires a one line variation in the ESP32 code.

Next, all of the button inputs are debounced. We define variables for comms, local, and truth that can be switched through depending on number of players and player ID. Comms takes in local player variables and outputs the truth for all four players.

Also included in the top level are the variables tied to making the level timer start and stop, dependent on the game state determined by the main FPGA. Finally the modules for game logic and graphics are called, as well as handling the signals needed for the VGA display.

III. Game Logic (Julia)

The overall game logic module is responsible for controlling all the inner workings of the game, and its outputs are fed to the graphics modules in coordination with the communications module based on how many players are involved. The game is primarily controlled by three modules: individual player control, main FPGA control and time remaining.

A. Individual Player Movement

Modules for player movement and player state were combined in this module to run locally on each FPGA. We designed this so that a local player would not experience any lag versus a situation where their button presses were sent to the server and processed there.

The player_move module was responsible for individual player movement on the board. I also added collisions here by considering the other players' locations received from the server. Player collisions took much longer than expected to implement to get the logic just right. I used a series of inequalities to define the moving player in relation to the others. Then I weighted the equations by a few pixels to make them easier to pass except for the direction towards the collision object. This allows the player to unstick once the two have collided and want to move away in another direction.

Outputs:

- Player locations in pixels, 9 bits each
  - x-position
  - y-position
- Player facing direction, 2 bits
  0) Left

1) Right
2) Up
3) Down

The other part of the overall module was to update the player state based on their chop or carry action and location. This module defined all of the actions a player could do, like how to grab soup from a pot or turn on the fire extinguisher. This would move the player state between the following 11 options:

0) Do nothing
1) Chopping
2) Holding whole onion
3) Holding chopped onion
4) Holding empty pot
5) Holding raw pot
6) Holding cooked pot
7) Holding empty bowl
8) Holding bowl with soup
9) Holding fire extinguisher, off
10) Holding fire extinguisher, on

These three variables (direction, location, state) would then be sent to the main FPGA in order to update the global object grid.

B. Main FPGA Player Movement

The main FPGA was in charge of outputting many variables for all of the players. The same code was used for every FPGA, so the main FPGA control module was on every FPGA. However, a mux at the top level determined whether the FPGA should listen to the local or network version of the variables based on player ID, so each FPGA would always only listen to player 0.

The first thing main control does is determine the overall game state. The game consists of 5 states as follows:

0) Welcome Menu
    - All players set their individual player number and number of players total
    - Hit btnc to start game
1) Game Introduction - Wait 3 seconds so players can view map, players can't move
2) Start Game - Timer starts, players can move
3) Pause Game - Timer pauses, all objects freeze
4) Finish Game - Once timer runs out, press any button to return to State 0

In the welcome menu, I also implemented a state machine where player 0 can create a team name that consists of 3 characters, in ASCII that could be saved with the player score to the server.

The main FPGA control module also contains an orders and points module and a module that considers players actions in order to update the object grid.

The orders and points module out puts a 4 bit array that has a 1 for every order that is active, and a corresponding array of time remaining to fill the order. The module checks to see if there are no active orders, and adds a new one if so. Otherwise it adds an order every 20 seconds unless the array is full.

The module also receives the state of the two turn in spots, and if there is soup, adds points. Players receive a baseline 20 points for each order filled and an additional 2 to 6 points as a time-scaled bonus for completing an order early. Once the points are added, the module sends a clear space signal to the action module so that the soup disappears. This module also takes away 10 points if the order timer expires and removes the order.

The other module under main control is ”action” which updates the global object grid. The object grid is a 8 by 13 by 4 bit array that divides the player space by those dimensions. Each space has a 4 bit state, which corresponds to those listed here:

0) Empty
1) Onion, whole
2) Onion, chopped
3) Bowl, empty
4) Bowl, full of cooked soup
5) Pot, empty
6) Raw pot, full
7) Cooked pot, full
8) Pot on fire
9) Empty counter space on fire
10) Fire extinguisher

The module takes all the player actions, including the local player, and updates the grid of objects based on changes in player state. For example, if a player state was “holding bowl” and now it is nothing, we know they put the bowl down and now the object grid should have a bowl one space in front. The supporting modules convert player pixel location to grid and check what object is in front of a player.

Since it is updating the object grid, this module is also responsible for tracking chopping, cooking and combustion time. When a player puts an onion down and starts chopping, a timer counts down to when the onion converts into a chopped onion. Once an uncooked pot is placed on the stove, another timer starts to count down to when it is cooked. These timers were made to address the grid coordinates for each location that needed it, so two cutting boards and two pots meant four timers total. In addition, there were two timers for combustion length. Once a pot became cooked, this started a timer for when the pot would catch on fire. Each stove top required a state machine that switched from nothing to cooking to on fire and back once a player put the fire out with the extinguisher.

Pixel to grid is a quick conversion for players' locations in order to interact correctly with objects. This is implemented as a quick subtraction and shift. Originally I wrote every line out as an inequality before Lacthu showed it could be done by this simple method.

The grid in front module returns the coordinates of the grid in front of a player by considering their direction and location. Again, it was more streamlined to write a quick reference function than to write out the specifics every time.

Similarly, the check in front module returns the object in front of the player based on the grid and the player's direction and location. This helps the main module know what object is there and therefore what state the player should switch into when an action is taken.

C. Time Remaining

This module starts once the state becomes "Start Game". It counts down once every second unless the game is paused. Once time left is zero, the game moves to the next state, "Finish Game". The amount of time remaining appears on the hex display in decimal after being converted by Lacthu's hex-to-decimal module.

We were able to make this module act entirely local so that communications didn't have to send the time over the network, since minimal bits is preferred. It is solely determined off the game state, which takes just 3 bits to send.

D. Lessons Learned

Overall, I learned a lot over time about how to use methods that work smarter not harder. With Verilog there is a balance between hardwiring things and finding more algorithmic ways to solve the problem, like the pixel-to-grid module. I wasn't able to make the fire spread when food was on the stove for too long because I couldn't find a way to do it feasibly.

It was also a challenge to keep everything organized over time, and we were occasionally tripped up by slightly different variable names that demanded hours of debugging time. Given more time, I would want to clean up extraneous items like unneeded variables and poor naming, as well as adding extra features to the game. Overall, our team effort was well integrated via Github, frequent meetings and overall responsiveness so this made it easier to solve other challenges.

Most of the time went to designing state machines and debugging them once they were written. There are a lot of combinations of possibilities that the various state machines had to address, and a lot of loopholes that I had to go back and address. I tested my code via testbenches and then moved to tracking down bugs we found as we played the game. For example, it took me a while to find the loophole that caused the pots to catch on fire immediately when an onion was added to the stove because the timer wasn't resetting on exit the first time through. Things mostly worked the way I expected the first time through, which means the time invested in thinking things out first and communicating with my team about how we would connect things was worth it.

IV. Graphics (Lacthu)

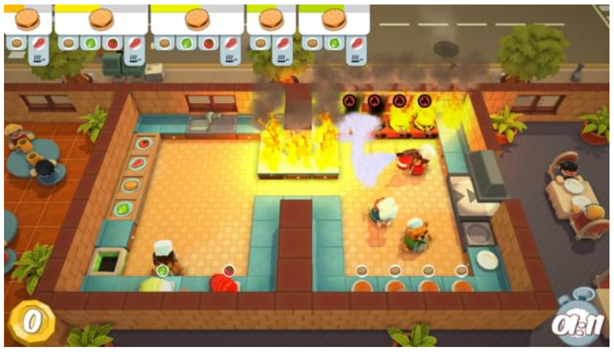


Fig. 1. Screenshot of the real Overcooked! game

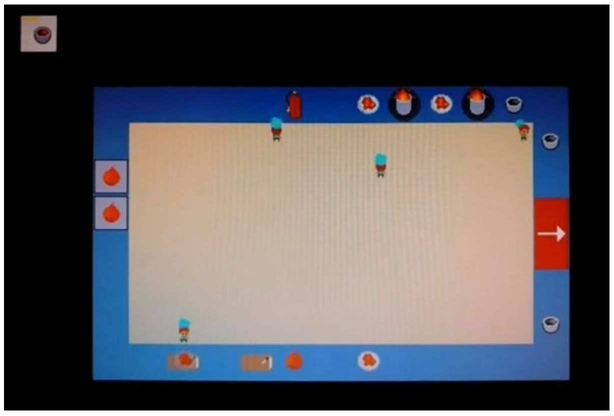


Fig. 2. Screenshot of our Overcooked! FPGA game, in 4 player multiplayer

The graphics module was responsible for reading state from the game logic module and displaying the game map, order displays, and the player and food sprites on the screen. This overall module takes as input the clock, reset, and VGA signals (hcount, vcount, hsync, vsync, and blank), as well as the game state of the main FPGA, the current grid state, the orders and time remaining on the orders, and the direction, (x, y) position, and state of all four players and outputs the RGB value for a pixel at the current hcount and vcount values. This game used a 640x480 pixel display, which we converted into a 20x15 top-down grid, with each square of the grid being 32x32 pixels. The game board itself only uses up a 13x8 grid space on a rectangular map, which is the same size as the one in the screenshot in Figure 1, without the bottleneck in the center, to avoid having as many collisions in game.

The graphics modules were divided into a few modules for each category of item that needed to be displayed. `pixel_out` values from each of these modules were combined in the top level graphics module based on a z-index, determining which pixel should be output: (items on counters had a higher index than counters, the player had the highest index to always be displayed).

States:

- Welcome Menu
- Game

Inputs:

- `clock, reset`
- From VGA Module
  - `hcount, vcount`
  - `hsync, vsync`
  - `blank`
- From Game Logic Module
  - local `player_state`
  - local `player_direction`
  - local `player_x, player_y`
- From Networking Module
  - online `game_state`
  - online `player_state`
  - online `player_direction`
  - online `player_x, player_y`
  - main `object_grid`
  - main `orders, order_times`
  - main `object_grid`

Outputs:

- `pixel_out` - 12 bit RGB values to display
- `hsync_out, vsync_out, blank_out`

A. Game Map

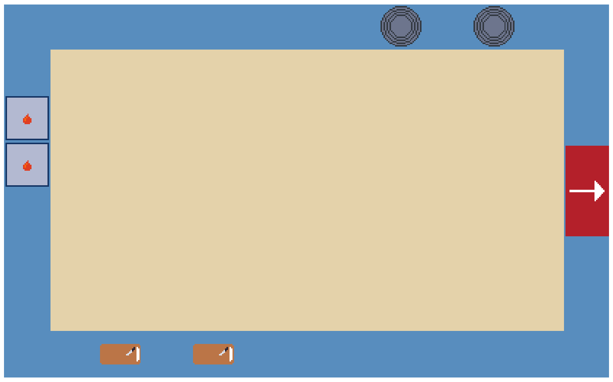


Fig. 3. Sprite used to display the static objects in the background of the map.

The game map takes in an hcount and vcount value and outputs a pixel at the current (hcount, vcount) for any constant tiles independent of the game state, such as the background, tabletop counters, walls, and flooring. Initially, this was meant to be done using a blue rectangle for a counter and a light brown rectangle for the floor, while tiles for stovetops, cutting boards, and the output area were meant to be small 32x32 COE files that would be placed as picture blobs at a certain tile.

However, this implementation gave us undesired color outputs while trying to tweak RGB values, so we ended up drawing the entire background as a 416x256 pixel sprite and uploading this as a COE. This method was pretty inefficient, and probably could have been done in the method we outlined above to save up on BRAM, but did end up saving us time generating bitstream from centering and placement issues, which we did end up having while implementing grid objects.

B. Static Grid Sprites

Inputs

- `clock`
- `object_grid`
- `hcount, vcount`
- `x, y` – of the top left pixel of a grid square

Outputs

- `pixel_out`

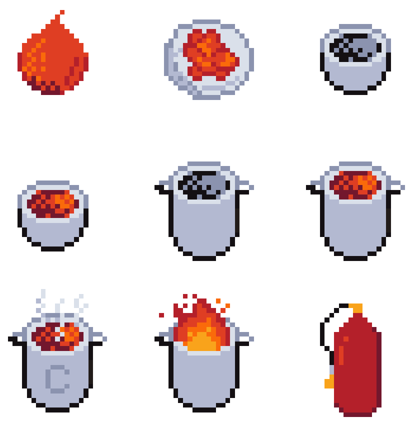


Fig. 4. Food and other grid items spritesheet

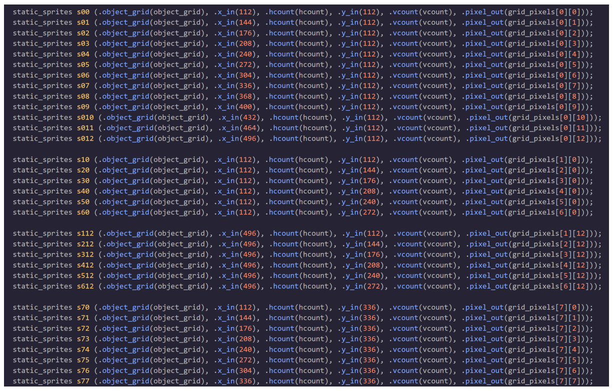
This module includes components that are placed on counters that can be interacted with (picked up or put down) by the player, such as pots, bowls, and the fire extinguisher. We had nine different 32x32 pixel sprite COEs that we used for this, representing each possible state of the grid:

- onion, whole
- onion, chopped
- bowl, empty
- bowl, full
- pot, empty
- pot, full and raw
- pot, full and cooked

- pot, on fire
- fire extinguisher

This module converts the input `(x, y)` location to a `(grid_x, grid_y)` location, which was then used to index into `object_grid` to determine the state of the object at a certain tile. It then reads the corresponding image address of the COE in order to output the correct pixel.

This module takes in an x and y value equivalent to the top left corner of a square of the grid, which we found by converting `(hcount, vcount)` to `(grid_x, grid_y)` using `pixel_to_grid`, and then converted the grid coordinates back to `(x, y)` using `grid_to_pixel`: this is necessary because there is a 112 pixel offset from the top right of the screen – $(0, 0)$ is actually $(112, 112)$! – and we want to drop the 5 least significant bits of our `(x, y)` after accounting for this offset. By using this method to control the x and y locations, we only had to call this module once, instead of multiple times for each location of the grid and using a mux to determine which of the pixels would be outputted for a certain `(hcount, vcount)`, which was more efficient and saved us from a bunch of copy-pasting (see Figure 5).



```
static_sprites s00 (.object_grid(object_grid), .x_in(112), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][0]));
static_sprites s01 (.object_grid(object_grid), .x_in(144), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][1]));
static_sprites s02 (.object_grid(object_grid), .x_in(176), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][2]));
static_sprites s03 (.object_grid(object_grid), .x_in(208), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][3]));
static_sprites s04 (.object_grid(object_grid), .x_in(240), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][4]));
static_sprites s05 (.object_grid(object_grid), .x_in(272), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][5]));
static_sprites s06 (.object_grid(object_grid), .x_in(304), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][6]));
static_sprites s07 (.object_grid(object_grid), .x_in(336), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][7]));
static_sprites s08 (.object_grid(object_grid), .x_in(368), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][8]));
static_sprites s09 (.object_grid(object_grid), .x_in(400), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][9]));
static_sprites s010 (.object_grid(object_grid), .x_in(432), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][10]));
static_sprites s011 (.object_grid(object_grid), .x_in(464), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][11]));
static_sprites s012 (.object_grid(object_grid), .x_in(496), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[0][12]));

static_sprites s10 (.object_grid(object_grid), .x_in(112), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[1][0]));
static_sprites s20 (.object_grid(object_grid), .x_in(112), .hcount(hcount), .y_in(144), .vcount(vcount), .pixel_out(grid_pixels[2][0]));
static_sprites s30 (.object_grid(object_grid), .x_in(112), .hcount(hcount), .y_in(176), .vcount(vcount), .pixel_out(grid_pixels[3][0]));
static_sprites s40 (.object_grid(object_grid), .x_in(112), .hcount(hcount), .y_in(208), .vcount(vcount), .pixel_out(grid_pixels[4][0]));
static_sprites s50 (.object_grid(object_grid), .x_in(112), .hcount(hcount), .y_in(240), .vcount(vcount), .pixel_out(grid_pixels[5][0]));
static_sprites s60 (.object_grid(object_grid), .x_in(112), .hcount(hcount), .y_in(272), .vcount(vcount), .pixel_out(grid_pixels[6][0]));

static_sprites s112 (.object_grid(object_grid), .x_in(496), .hcount(hcount), .y_in(112), .vcount(vcount), .pixel_out(grid_pixels[1][12]));
static_sprites s212 (.object_grid(object_grid), .x_in(496), .hcount(hcount), .y_in(144), .vcount(vcount), .pixel_out(grid_pixels[2][12]));
static_sprites s312 (.object_grid(object_grid), .x_in(496), .hcount(hcount), .y_in(176), .vcount(vcount), .pixel_out(grid_pixels[3][12]));
static_sprites s412 (.object_grid(object_grid), .x_in(496), .hcount(hcount), .y_in(208), .vcount(vcount), .pixel_out(grid_pixels[4][12]));
static_sprites s512 (.object_grid(object_grid), .x_in(496), .hcount(hcount), .y_in(240), .vcount(vcount), .pixel_out(grid_pixels[5][12]));
static_sprites s612 (.object_grid(object_grid), .x_in(496), .hcount(hcount), .y_in(272), .vcount(vcount), .pixel_out(grid_pixels[6][12]));

static_sprites s70 (.object_grid(object_grid), .x_in(112), .hcount(hcount), .y_in(336), .vcount(vcount), .pixel_out(grid_pixels[7][0]));
static_sprites s71 (.object_grid(object_grid), .x_in(144), .hcount(hcount), .y_in(336), .vcount(vcount), .pixel_out(grid_pixels[7][1]));
static_sprites s72 (.object_grid(object_grid), .x_in(176), .hcount(hcount), .y_in(336), .vcount(vcount), .pixel_out(grid_pixels[7][2]));
static_sprites s73 (.object_grid(object_grid), .x_in(208), .hcount(hcount), .y_in(336), .vcount(vcount), .pixel_out(grid_pixels[7][3]));
static_sprites s74 (.object_grid(object_grid), .x_in(240), .hcount(hcount), .y_in(336), .vcount(vcount), .pixel_out(grid_pixels[7][4]));
static_sprites s75 (.object_grid(object_grid), .x_in(272), .hcount(hcount), .y_in(336), .vcount(vcount), .pixel_out(grid_pixels[7][5]));
static_sprites s76 (.object_grid(object_grid), .x_in(304), .hcount(hcount), .y_in(336), .vcount(vcount), .pixel_out(grid_pixels[7][6]));
static_sprites s77 (.object_grid(object_grid), .x_in(336), .hcount(hcount), .y_in(336), .vcount(vcount), .pixel_out(grid_pixels[7][7]));
```

Fig. 5. Initially, we tried calling this module for every square of the grid. It was not a great idea.

C. *Player Sprites*

Inputs

- `clock`
- `hcount, vcount`
- `player_x, player_y`
- `player_direction`
- `player_state`

Outputs

- `pixel_out`

Sprites:

- Do nothing
- Chopping

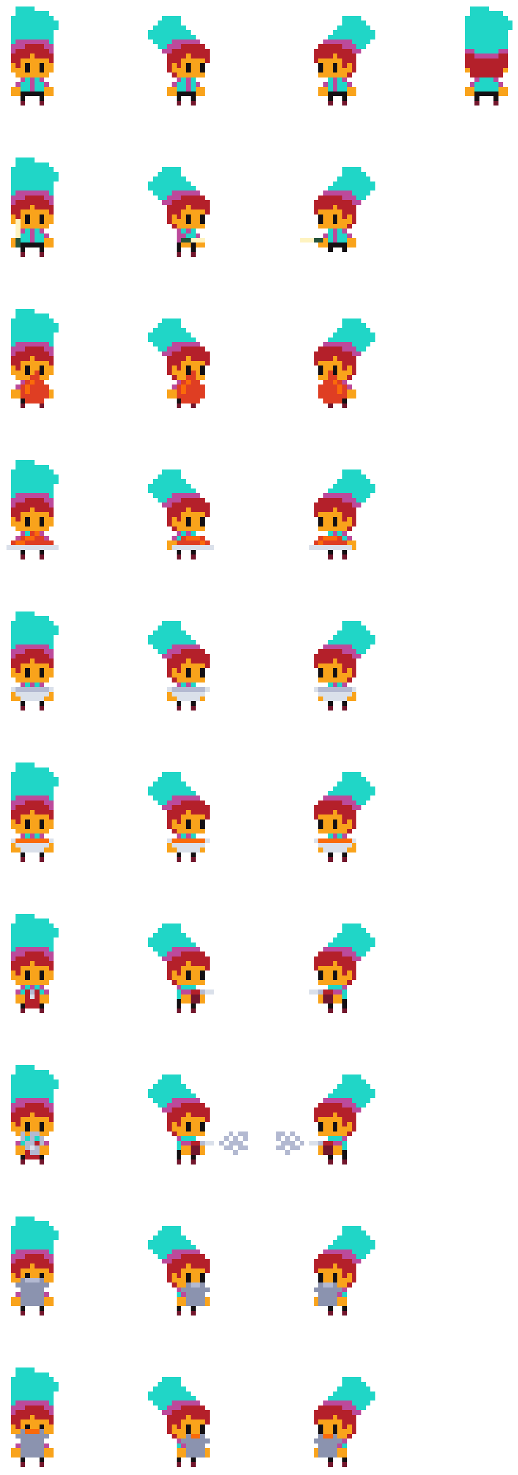


Fig. 6. Player spritesheet

- Holding whole onion
- Holding chopped onion
- Holding empty bowl
- Holding bowl with soup
- Holding fire extinguisher, off
- Holding fire extinguisher, on
- Holding empty pot
- Holding pot with soup

This module displays a different sprite for each player depending on the current state of the player and the direction that the player is facing, resulting in a total of 31 COEs (see Figure 6). We used the same sprite for up (since carrying an item while moving up was barely visible, and would have cost us about 9.2kB of memory),

The player sprite module finds the color address corresponding the same image address of each COE. We then have a mux determining which address is valid, depending on the current player state and the current player direction, and then finds the appropriate color on the color map.

An instance of this module was called for each player: depending on the number of players in the game, we also had to determine which player pixels `pixel_out` would return. Since all of the sprites returned a white background, this was done using case statements: if there was only one player, we would output player 1's `pixel_out`. If we were in multiplayer mode, we would output the first player whose `pixel_out` was not white (12'hFFF, the background color of the sprite), starting from player 1 taking precedence.

We additionally implemented one sprite animation, when the player is currently chopping: in the chop state, the player alternates between the chopping sprite and the move down sprite, to make it seem like the knife is moving every few frames. Additional sprite animations would have doubled the number of COEs we had, as we would have to have different sprites showing the legs of the player moving, which would have exceeded our BRAM limit.

With the exception of the chop state, all other player sprites had mirrored left/right sprites. From this, one improvement that we could make on this game could be to try using the same COE for both directions, but try calculating the image address differently for a mirrored sprite. This would save us on 10 COEs, which would save us about 10kB of memory, allowing us to implement other sprites.

D. Loading COEs

With so many sprites, it quickly became tedious for all 3 of us to manually add all of them as IPs each time we wanted to regenerate the Vivado project. Instead, we wrote a small Python script[1] to generate a TCL script, which does the IP generation for us. There's probably a better approach, but this worked pretty well for us[2]!

E. Order Displays

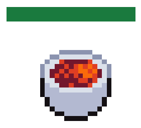


Fig. 7. Order display shown in the top left of the screen.

Inputs

- `clock`
- `hcount, vcount`
- `x, y` - position of the order timer on the screen
- `order`
- `order_time`

Outputs

- `pixel_out`

This component of the display the number of orders out, and time remaining on each order to a player. Since there could be a maximum of four orders at a time, we called this module four times with different x locations (so that they would be aligned side by side on the top left of the screen), indexing into the `order` and `order_time` arrays to determine whether or not there was an order – if there wasn't an order, output black pixels (12'h000); if there was, display the order and time remaining.

The order display shows the current order and how much time is remaining for that order, and are displayed in the top left corner of the screen. There is a timer bar showing how much time is remaining on the current order, which starts as a full (30 pixel wide) green bar, representing 30s remaining. This bar gets shorter by one pixel every second, turning yellow at 20s remaining, and red at 10s remaining. Conveniently, since an order time was set to 30s max, and each order display had a timer displayed on top of a 32x32 pixel sprite, we didn't have to scale the order timer before displaying it.

F. Combining Graphics

Graphics were combined at the top-level graphics module, which calls the previous four modules, combines the graphics of each of the display components,

[1]https://github.com/kyeb/overcooked-fpga/blob/main/fpga/generate_bram_generator.py

[2]Until the script ran all of our computers out of RAM, that is. We're pretty sure the script is duplicating work that happens during the Generate Bitstream flow, but we're not familiar enough with Vivado commands/terminology to fix that.

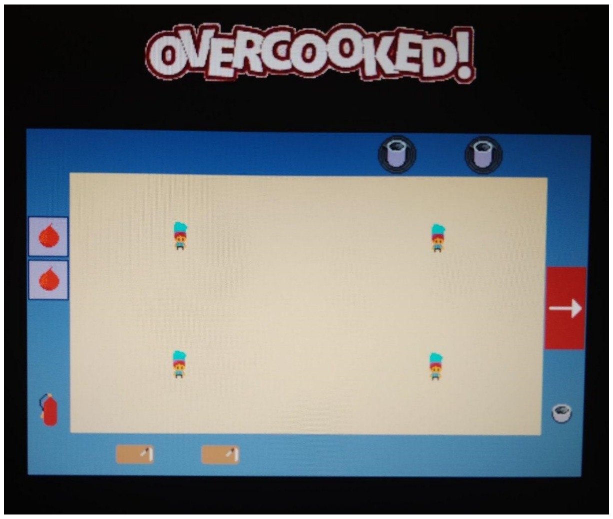


Fig. 8. Screenshot of the game start screen

and determines which pixel to output based on a z-value from each of the modules if there were multiple components displayed on a single tile. The precedence of z-values is as follows:

1) start_screen – only if state is `WELCOME`
2) player_pixel – which player is already determined within the *Player Sprites* module
3) object_pixel
4) floor_pixel + info_out0 + info_out1 + info_out2 + info_out3

If the current game state is `WELCOME`, we would display the start screen as long as the pixel from the start screen COE was not black. Otherwise, we would display the player pixel (already put through a mux to determine which player is getting displayed) if the player existed at that pixel if it was not white; otherwise the grid object pixel if it was not white; or lastly, the background and any order displays.

Using white for the background of sprites made it a bit easier to compare z-values: for all COEs except the welcome screen, getting a `pixel_out` value of 12'hFFF meant that we did not have an object located at that pixel, and could display something else with a lower precedence instead.

G. Lessons Learned

Remember to initialize the sizes of values! This ended up messing up the color map readings and giving us messed up looking sprites, and took an embarrassingly long time to fix.

A lot of the issues we found were faced during debugging, when sprites were off-center or had the wrong color. Doing the calculations for the locations of each sprite to have them all align on paper beforehand is important to avoiding spending a bunch of time waiting to generate bitstream after changing a single value to get your background to move left three pixels.

To save on BRAM, we could attempt reusing sprites (for instance, the head of the character is the same across all 31 player COEs, and could probably be reused. We could try having only four COEs for the player facing in four different directions, and then different held object COEs (which would only be 5x5) COEs, saving memory that can be used for other sprites + more sprite animations. Additionally, we could try using the same COE for the left and right facing sprites, but change the address for the mirrored image, to save up on additional COEs. Using these optimizations, we would only end up using 3kB for all of the player sprites, instead of 31kB.

V. Multiplayer Communications (Kye)

In the era of COVID-19, it's more difficult to simply enjoy a game of Overcooked with your friends. We propose an online multiplayer version of the game that can be played with anyone, anywhere (if they have a Nexys 4 DDR FPGA lying around, that is).

In multiplayer mode, our game will allow players to work together to collaboratively cook their soup. At a high level, this will work by designating a single FPGA as the "main" and letting the others act as "secondary." This is important to ensure there is a single source of truth for the game state.

A. Serial Interface[3]

The FPGA needs an interface to the ESP32 in order to send commands and receive network data. This ended up taking the form of a UART interface, since we need data to flow in both directions and want a simple-as-possible protocol. Using UART also means the C++ side is as simple as Serial.write() and Serial.read(), instead of having to deal with an SPI or I2C library.

The module to transmit (TX) was simple, since we use build a slow version in lab 2. The receive module (RX) was much more difficult to get working properly.

Serial TX Module

Inputs

- `trigger` — set to 1 when `data_in` is set to trigger a send to the ESP32
- `data_in` — data to actually send

Outputs

- `line_out` — connected to `ja[0]` at the top level and wired to the ESP32 by hand
- `ready` — set to 1 when the TX module is in the idle state, used by the networking module to determine when to trigger the next send

[3]https://github.com/kyeb/overcooked-fpga/blob/main/fpga/serial.sv

Serial RX Module

Inputs

- `line_in` — connected to `ja[1]` at the top level and wired to the ESP32 by hand

Outputs

- `valid` — set to 1 for 1 cycle when `data_out` is valid
- `data_out` — decoded data from the UART

On top of the two basic TX/RX modules, there are also modules to transmit and receive 32 bits (4 bytes) at a time, in quick succession. This is primarily for convenience when writing the higher level networking logic (Section V-B), since our data types to be synchronized primarily fit neatly into single 32-bit chunks or divides evenly by 32. The word "packet" is used in the following sections to refer to 32 bits, sent as 4 UART transmissions.

B. Networking Logic[4]

Depending on various game state and inputs, some data needs to be sent to the server and some needs to be recalled from the server. Since network requests take a *very* long time compared to the timescales of the FPGA clock, we use a state machine keeping track of the status requests.

We began with implementing networked player movement. This just required an X-Y coordinate pair (18 bits together), plus 4 bits of "player state" to determine which player sprite to be used, plus 2 bits of player direction for which direction they're facing. This all fits neatly into 32 bits, which is easily handled by our high-level serial TX/RX modules as described in Section V-A. The format of those 32 bits are as follows:

Packet structure

- `data[31:30]` (2 bits) — player number
- `data[29:28]` (2 bits) — player direction
- `data[27:19]` (9 bits) — x position
- `data[18:10]` (9 bits) — y position
- `data[9:6]` (4 bits) — player state
- `data[5:3]` (3 bits) — game state if `player_ID == 0`, else `0`
- `data[2:0]` (3 bits) — packet type

Packet types

- `000` — player state
- `111` — HTTP completion ACK
- `001` — board state start packet (indicates the next 15 packets will be the entire board state)

[4]https://github.com/kyeb/overcooked-fpga/blob/main/fpga/network.sv

This structure is encoded on one side, transmitted through the ESP32 and server, then decoded in the same way on the other FPGAs in order to synchronize the player positions and actions.

The `comms` module contains 4 state machines. The first is trivial — it simply sets the local positions for the other players when it receives a packet containing a player position, as determined by the last 3 bits, which are different for different packet types. This runs at all times, on all FPGAs.

The second handles transmitting the player state. This also needs to run on all 4 FPGAs, since every player needs to be able to see all the other players. The state machine sits in idle unless the local player's state changes, which is almost always true. If the player state is different from the last time it was when in idle[5], the serial TX module is triggered with the player state data. It then sits in the "wait" state until it receives an ACK packet back, as designated by the least-significant 3 bits.

The third and fourth state machines handle transmitting and receiving the board state, respectively. That means the third only runs on the main FPGA, and the fourth only on secondaries. They work similarly: when the transmit/receive modules are not busy handling player states, they send/read the total board state as a series of 15 packets, totalling 480 bits. The transmit module leads with a board state start packet (defined in the Packet Types section), and the receive module is triggered to start reading board state upon receiving a board state start packet. They each then handle reading/writing the correct registers in the `object_grid`, `time_grid`, and `points_total` to maintain synchronization.

Comms module

Inputs

- game state (primary only)
- `local_x`, `local_y`, `local_direction`, `local_state` — state for this player, to be sent to the server
- `player_ID` — to make sure the server knows which player this state is for, and to determine if it should act as the main or as a secondary
- `local_object_grid`, `local_time_grid`, `local_point_total` — state stored on the main to be synchronized out to the other FPGAs

Outputs

- Full player state (x, y, direction, state) from all 4 players
- Full game state (board, times, points) from

[5]This handles the case where the local state changes while the ESP32 is still sending the POST request and then stops changing — those will still be caught and transmitted on the next idle

This module serves to abstract the complexity of both HTTP requests and syncing data across a network so the display/game logic don't need to worry about it. There is significant complexity in data synchronization, especially from the low-level perspective of FPGA hardware, so this module ended up being somewhat complex to design.

Testing this module was done by syncing simple data between two FPGAs at first (just the first 8 switch bits). Gradually increasing the complexity of the synchronized data allowed us to build functionality modularly.

C. ESP32[6]

The code running on the ESP32 purely forwards the bits that the FPGA sends it on to the server. Depending on the state, as specified in section V-B, the FPGA will send various data to the ESP32. The ESP32 will then translate those into HTTP requests, send them to the server (section V-D), and transmit the data it receives back to the FPGA.

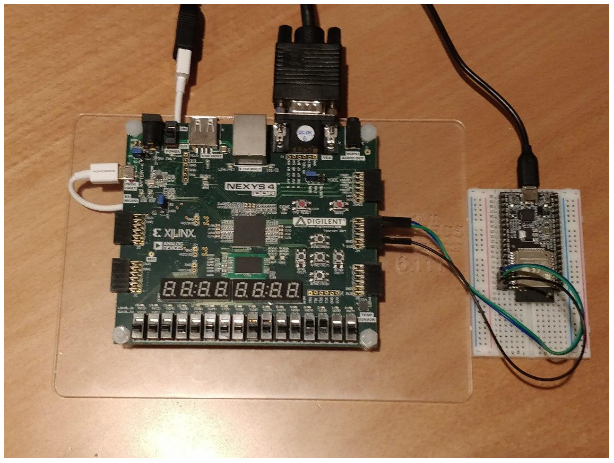


Fig. 9. The full setup

ESP32

Inputs

- RX — pin 22 on the ESP32, connected to `ja[0]` on the FPGA
- HTTP POST responses over WiFi

Outputs

- TX — pin 23 on the ESP32, connected to `ja[1]` on the FPGA
- HTTP POST requests over WiFi

Most of the logic in C++ for the ESP32 is to translate between usable data formats for the FPGA and for the Python server. The FPGA works in raw bits, which are sent over the wire to the ESP32 and read in using the simple `Serial.read()`. Each `Serial.read()` call returns 8 bits (1 byte). With each 4 bytes received, the ESP32 uses some bit tricks to combine them into a single, easily-readable 32-bit integer (`uint32_t`). This gives an easy data type to work with, since it can always be represented by a 10 digit integer.

On the secondary ESP32s, the received data is just a single 32-bit integer representing the complete player state. That is sent via POST request to the server, running on AWS. The server then returns the latest board state it has stored in the database, as a the body of the HTTP response. Upon receiving a response string from the server, the ESP32 forwards an "ACK" packet to the FPGA to reset the FPGA's TX state machine and indicate it's ready to receive another player position. The response string is parsed into an array of 15 32-bit integers. The ESP32 sends a "start packet" marked with a unique bit set to indicate it is about to send a board state. It then forwards the full board state it received on to the FPGA, one 32-bit packet at a time.

On the main ESP32, the received data is the full board state plus one player's state, which are handled independently but sent to the server in a single POST request. The response contains the latest 4 player's positions, which are relayed back to the main FPGA.

D. Server[7]

A tiny server is running on an AWS VM, somewhere in Ohio. Its only purpose is to take in the state from the ESP32s, write it to a database, and return it to the others on demand. Using Python made it particularly simple. The entire server is under 100 lines of meaningful code. The complexity of this kind of simple CRUD server is very low.

Since player states are represented as 32-bit integers, it is simple to do the same in Python. `update_player_state()` computes which player the player state came from (from the top 2 bits of the number), then updates the relevant entry in the database. `get_player_states()` returns a string concatenating the up-to-4 player states in a way that is easily parsed by the ESP32 back into integers.

Since the board state is just a single concatenation of 15 32-bit integers into a string to send over HTTP and HTTP responses must also be strings, it's pointless to parse the integers in Python at all. So, we just save the string in a singleton database table for the board state, returning the exact same string when needed.

[6]https://github.com/kyeb/overcooked-fpga/blob/main/esp32/esp32.ino

[7]https://github.com/kyeb/overcooked-fpga/blob/main/server/fast_app.py

E. Addressing Network Latency

In any multiplayer game, latency is the enemy. We made several key optimizations to make the latency reasonably playable.

First, network requests were reduced to an absolute minimum — all information that each ESP32 needs to send is concatenated into a single request, and all the information it needs to receive is sent back in the response. Barring asynchronous requests, this is optimal.

Second, I noticed that there was an increasing delay as the game went on. Eventually, with some research into the internal hardware serial implementation in the ESP32, I had a hunch that UART bytes were building up in the internal buffer and only reaching the ESP32 with a significant delay. This also makes sense as a bottleneck, since the ESP32 runs code significantly slower than the FPGA's pure hardware implementation. Amazingly, inserting a call to `FPGASerial.flush()` to clear the internal buffer was able to massively reduce this delay and keep most communications much more up-to-date, confirming my hunch.

Using a non-HTTP protocol such as directly sending UDP packets would likely reduce lag substantially. Unfortunately, we did not have time to explore that in this project.

F. Lessons Learned

It is crucial to actually look at a pinout of the device you are using. Somehow, I managed to waste almost 4 hours debugging just getting the basic UART TX/RX working. The problem? The FPGA numbers its header pins horizontally, not vertically. Pin 1 is to the left, not below pin 0.

If you're designing a protocol for two devices to communicate, treat it as such from the beginning. I hacked it together, only getting the basic functionality working at each level before moving on to the next task. This meant I often had to go back and revise previous code to fit new requirements. If I had just designed a protocol from the beginning to cover all the requirements, there would have been substantially less rewriting.

Finally, debugging many-step communication systems is hard. It's very important, at least while building, to have some form of tracing to make sure you know exactly where it is breaking down. There were many times when the communications wouldn't be working, and I would spend an hour or two rereading or tweaking some piece of code only to realize it was a problem with an entirely different device. Design an easy way to find where things are breaking in the pipeline of secondary FPGA $\rightarrow$ secondary ESP32 $\rightarrow$ server $\rightarrow$ main ESP32 $\rightarrow$ main FPGA.

VI. Block Diagram

Block diagram on next page.

VII. Verilog Code

https://github.com/kyeb/overcooked-fpga.

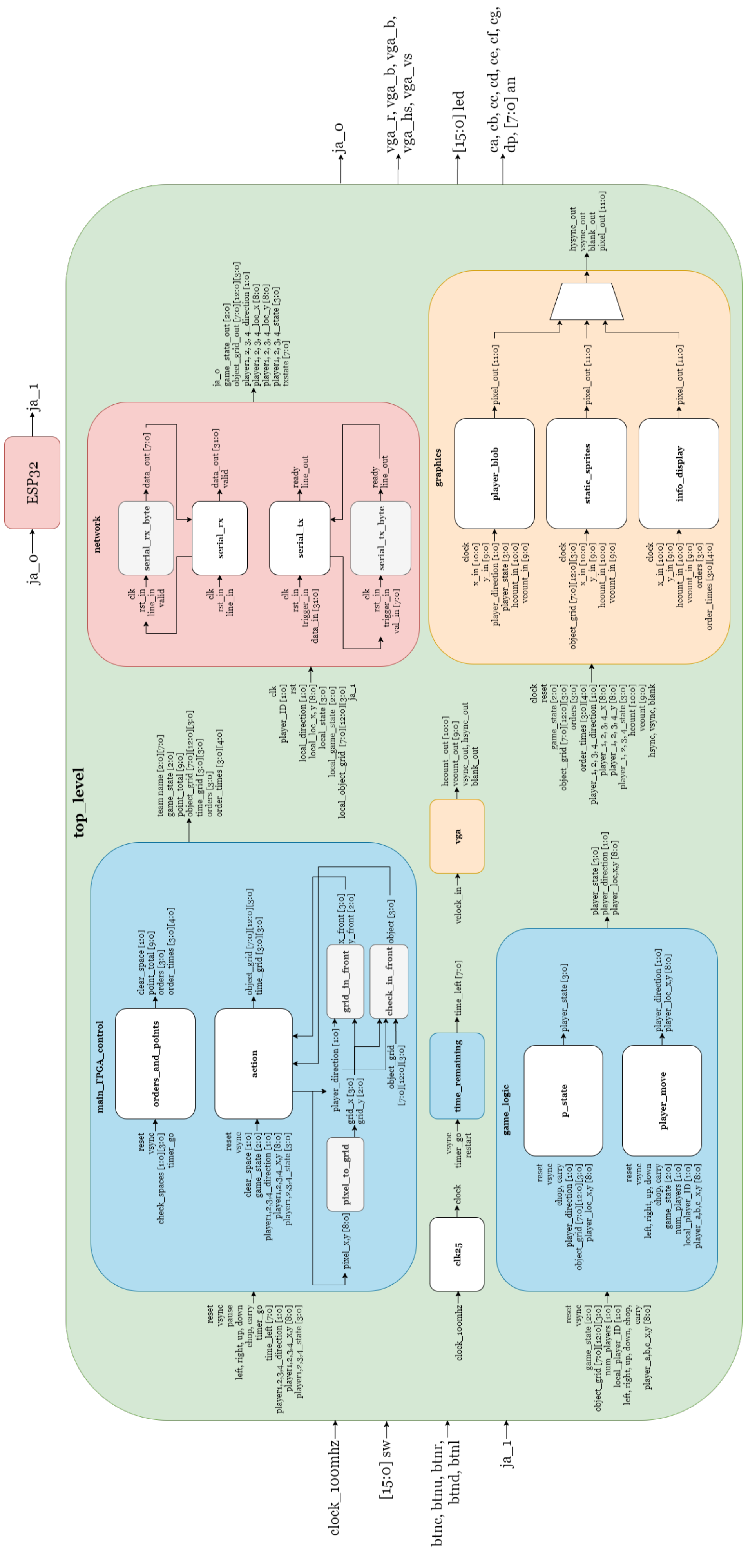


Fig. 10. Block Diagram

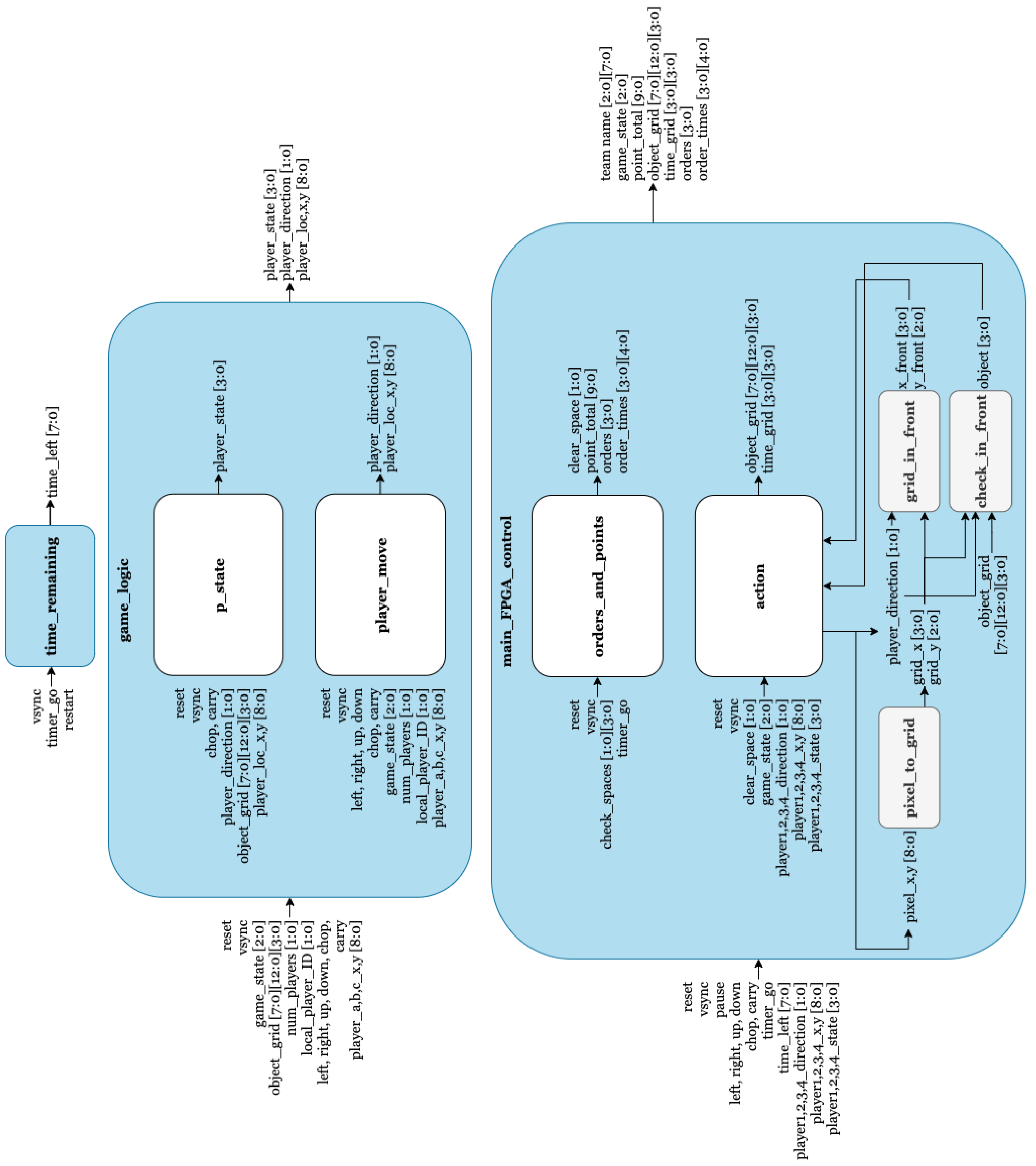


Fig. 11. Game Logic Block Diagram

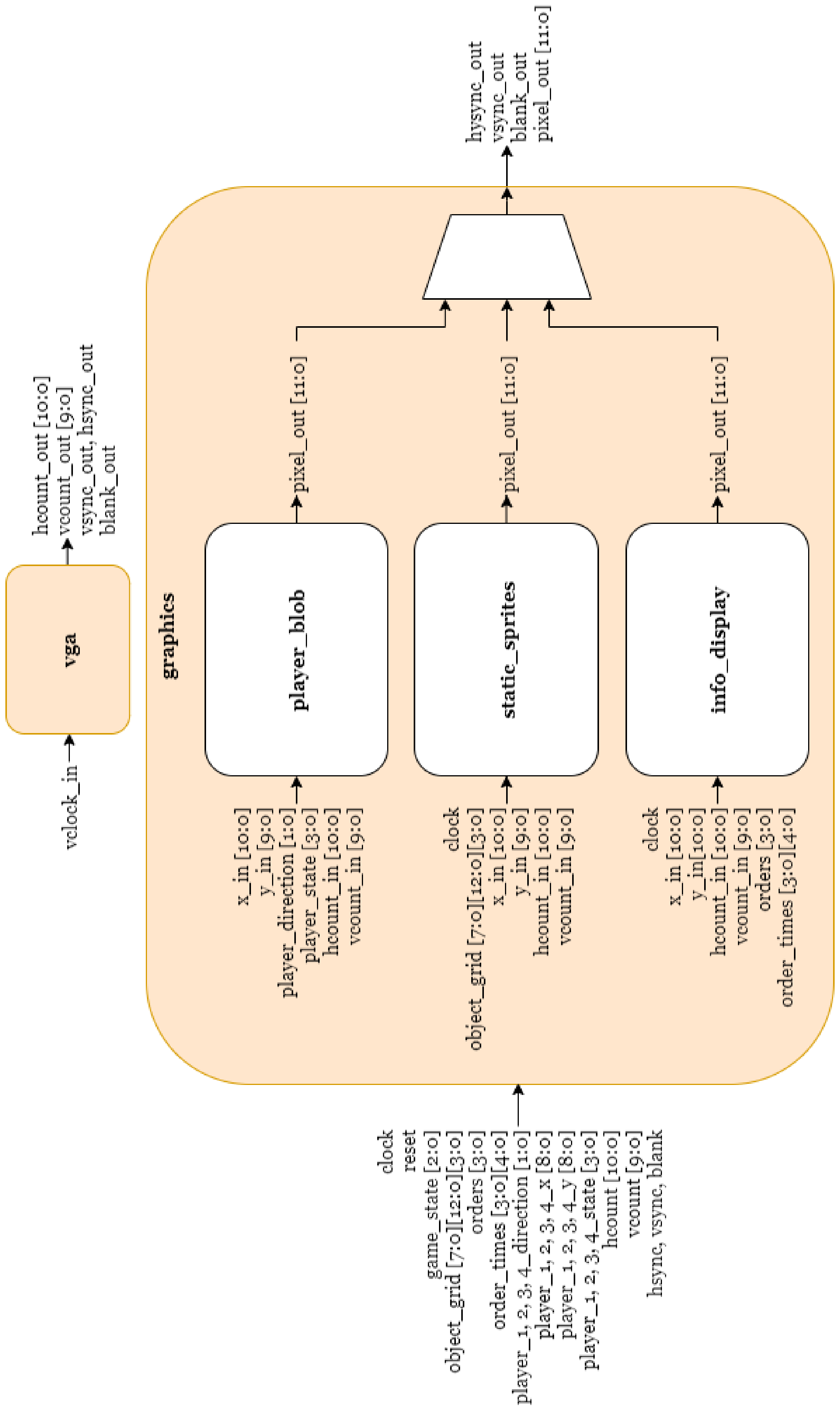


Fig. 12. Graphics Block Diagram

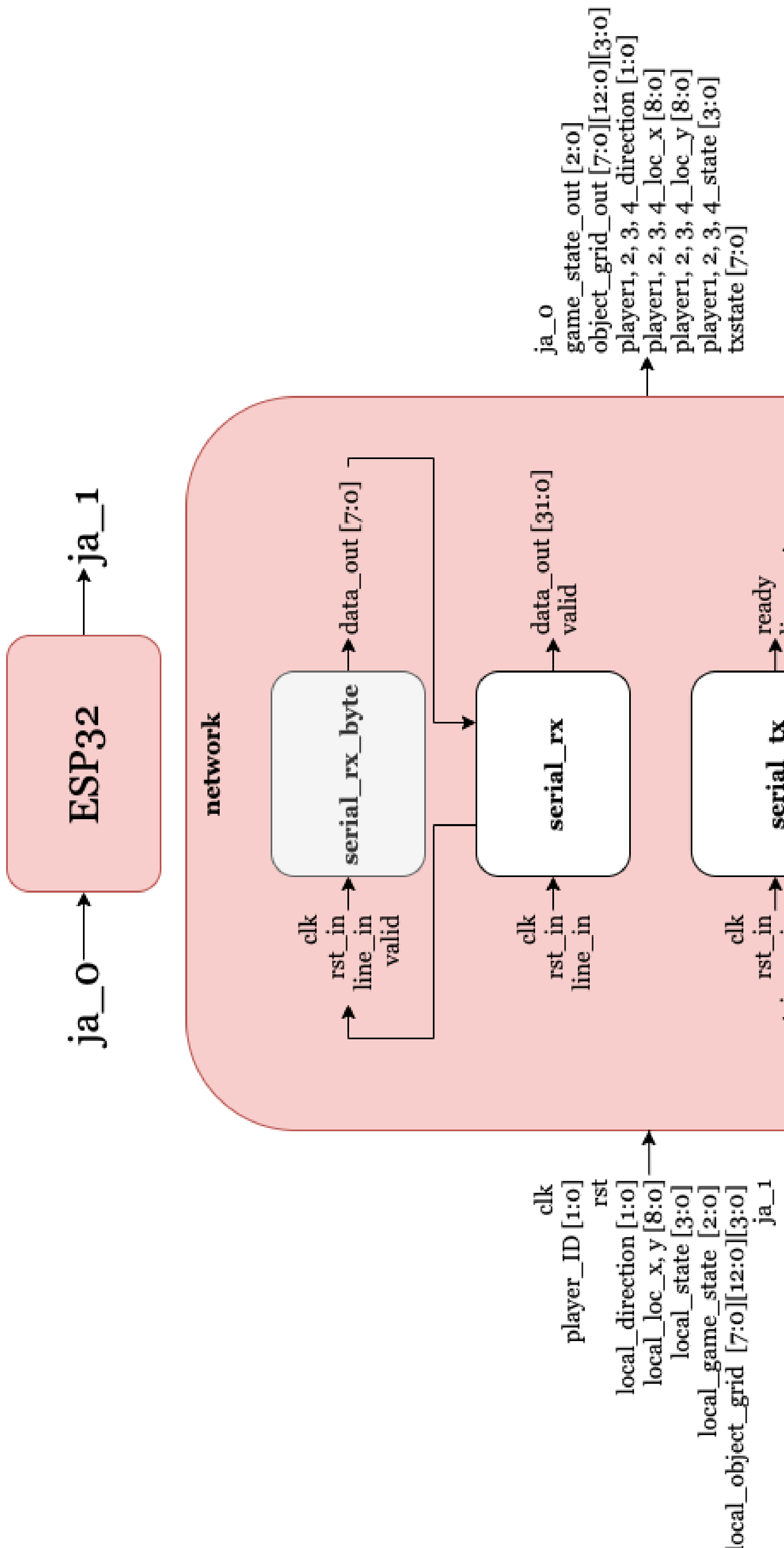


Fig. 13. Comms Block Diagram