# The Game : Pacman

# Graphical Interface

So the very first part of creating a game has to be controlling video output. There could have been a couple of options to select from like HDMI(High-Definition Multimedia Interface), DP(Display Port), DVI(Digital Visual Interface) etc but on NEXYS A7 Dev. board the only video interface connector available is VGA.

## VGA (Video Graphics Array) (module: vga_out ; file: vga_out.v)

**Theory**

A VGA connector carries R, G, B, Vertical sync, Horizaontal sync analog signals. Depending on how fast or slow one triggers hsync and vsync signals, video resolution and refresh rate will be determined by the display and be scaled automatically.

Here were fixed our resolution to 1280x800(addressable) 60Hz. The whole frame(including sync time, front porch, back porch) is going to be 1679*827 i.e 1388533. At 60Hz(count 1388533 60 time a second), it's going to be 83311980 which tell's that we need to achieve 83.3 MHz in order to output 60Hz refresh rate for a 1280x800(addressable) resolution screen.

**Implementation**

So the VGA Control circuit has to be synchronous to a 83.3 MHz clock. At every clock edge counter's needs to incremented or reset after the max values.

We will create two additional counters(curr_x, curr_y) which will be incremented only in the visible region(Hortizontal : 336 - 1615 ; Vertical : 27 - 826) of the screen i.e a counter for 0 - 1280 and 0 - 800. This would make the game logic and other top module easy to manage rather than keeping the track of visible or non-visible region.

A simple synchronous always block would suffice the control the counters

```
always@(posedge clk) begin

    hcount <= hcount + 1;
    if(hcount == H_COUNT_MAX) begin
        hcount <= 0;
        vcount <= vcount + 1;
    end
    if(vcount == V_COUNT_MAX) begin
        vcount <= 0;
    end

end
```

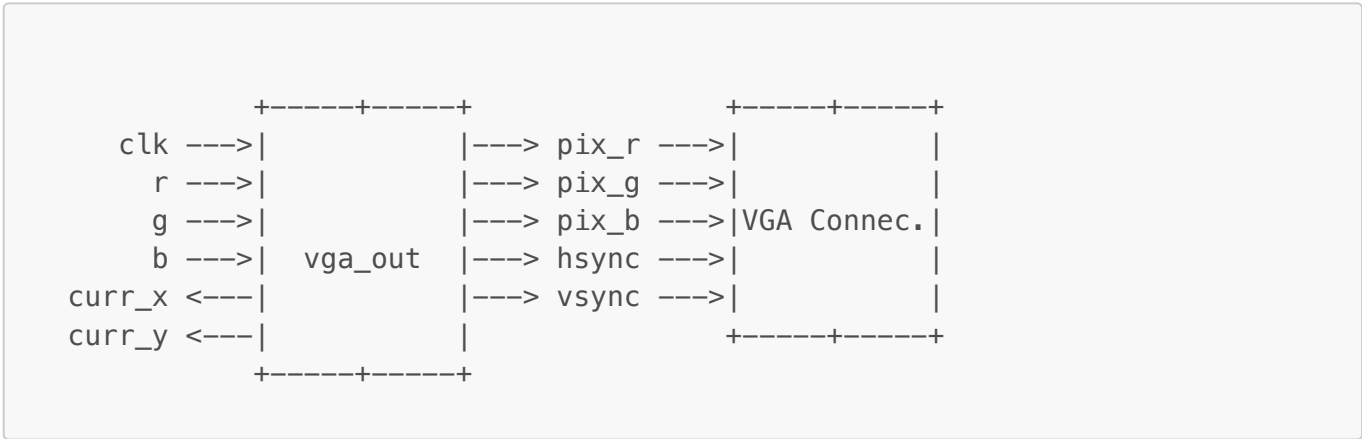
hsync, vsync, r, g, b signals could be totally combinational and can be controlled with the conditional operators depending on the sequential counters.

```
assign hsync = hcount <= H_SYNC_TOGGLE ? 0 : 1;
```

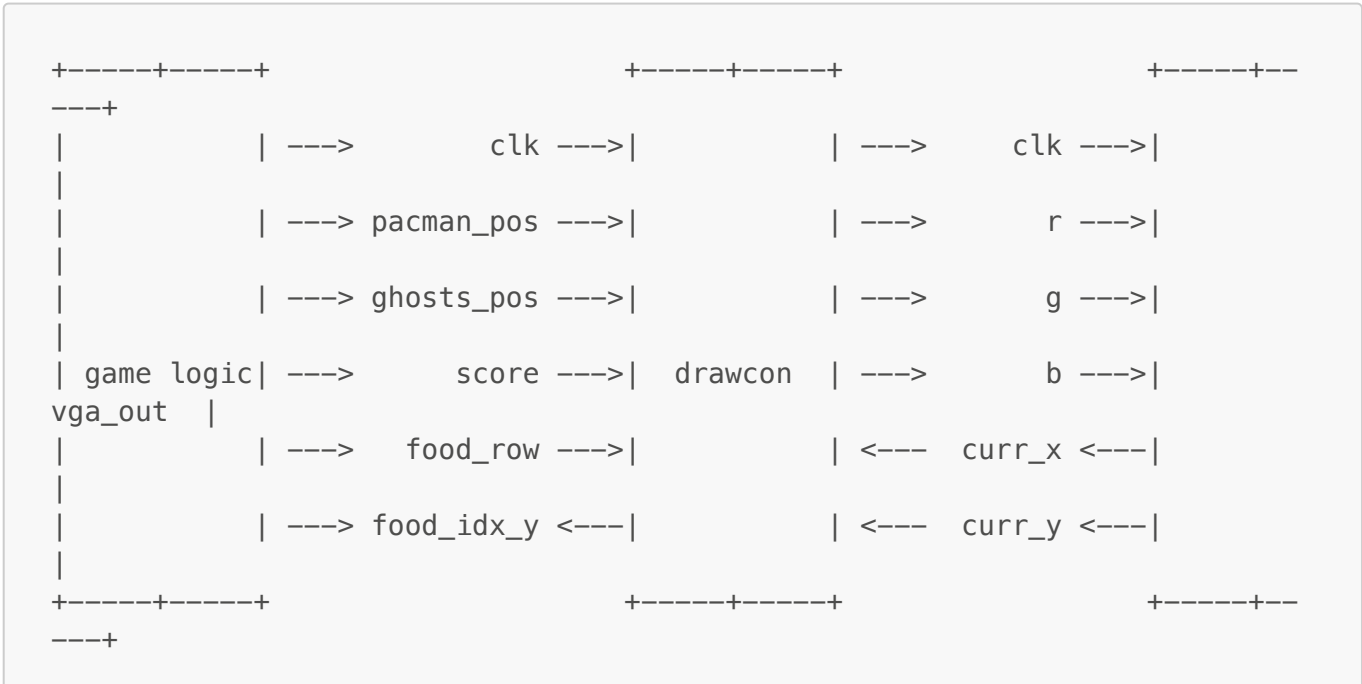
Now it's obvious to have clk, r, g, b and input to the module so that it can set actual r, g, b(pix_r, pix_g, pix_b) signals according to the top module. The output should be the curr_x and curr_y which can be used by the top module to check and range r, g, b if required(more explained later).

Rest of the outputs like pix_r, pix_g, pix_b, hsync and vsync should be mapped to the VGA connector pins(specified in the constraint file).

```
set_property -dict { PACKAGE_PIN A3    IOSTANDARD LVCMOS33 } [get_ports
{pix_r[0]}];
...
set_property -dict { PACKAGE_PIN B11   IOSTANDARD LVCMOS33 } [get_ports {
hsync }];
set_property -dict { PACKAGE_PIN B12   IOSTANDARD LVCMOS33 } [get_ports {
vsync }];
```




Map and Sprites (module: drawcon ; file: drawcon.v)



For drawing a sprite or map, above module should be used and according to the x, y position of the pixel being drawn, r, g, b should be controlled.

**Block Memory**

The map is stored in the block memory. Even though screen resolution is 1280x800 and map should contain value at each pixel but most of the pixels could be replicated and the map could be compacted by a factor 2^N. It's similar to scaling a 20x20 block to 240x240 where each pixel from 20x20 will be replicated by a factor of 12.

Here in this case, the map for the Pacman game was 80x50 so a replication factor of 16 will eventually scale to 1280x800 which is out screeen resolution. This could be further compacted by a matrix with information of edges and boundaries.

Map was only bi-colored(wall: Ocur Yellow ; Free Spce : Black) so at each point in 80x50 grid could be 0 or 1. If it's 1 if could reflect the wall and if it's 0 then free space for pacman and ghosts to move.

Block memory can be read as follows. Where map_idx_y is the vertical indexing i.e whichever vertical index is set, block memory will read the row into map_row which can be output to the VGA accordingly to display.

```
pacman_map_blockmem map(
    .clka(clk),
    .addra($unsigned(map_idx_y)),
    .douta(map_row)
);
```

**Scaling the map**

For scaling, the objective is to replication each x, y value from the map to a 16x16 block. So the code needs to designed to fetch a new map_row[i] element at every 16th curr_x and update map_idx_y at every 16th curr_y.

So all we need to do is to find a trigger at 16th of curr_x and curr_y. curr_x and curr_y is a 11 bit and 10 bit number respectively which are mere counters so curr_x[4] and curr_y[4] bit will toggled at a step of 16th only catch is that it will be 0 and 1 alternatively. To mitigate this a variables called mod_x and mod_y were declared which are xor'ed with curr_x[4] and curr_y[4] and updated when condition is true.

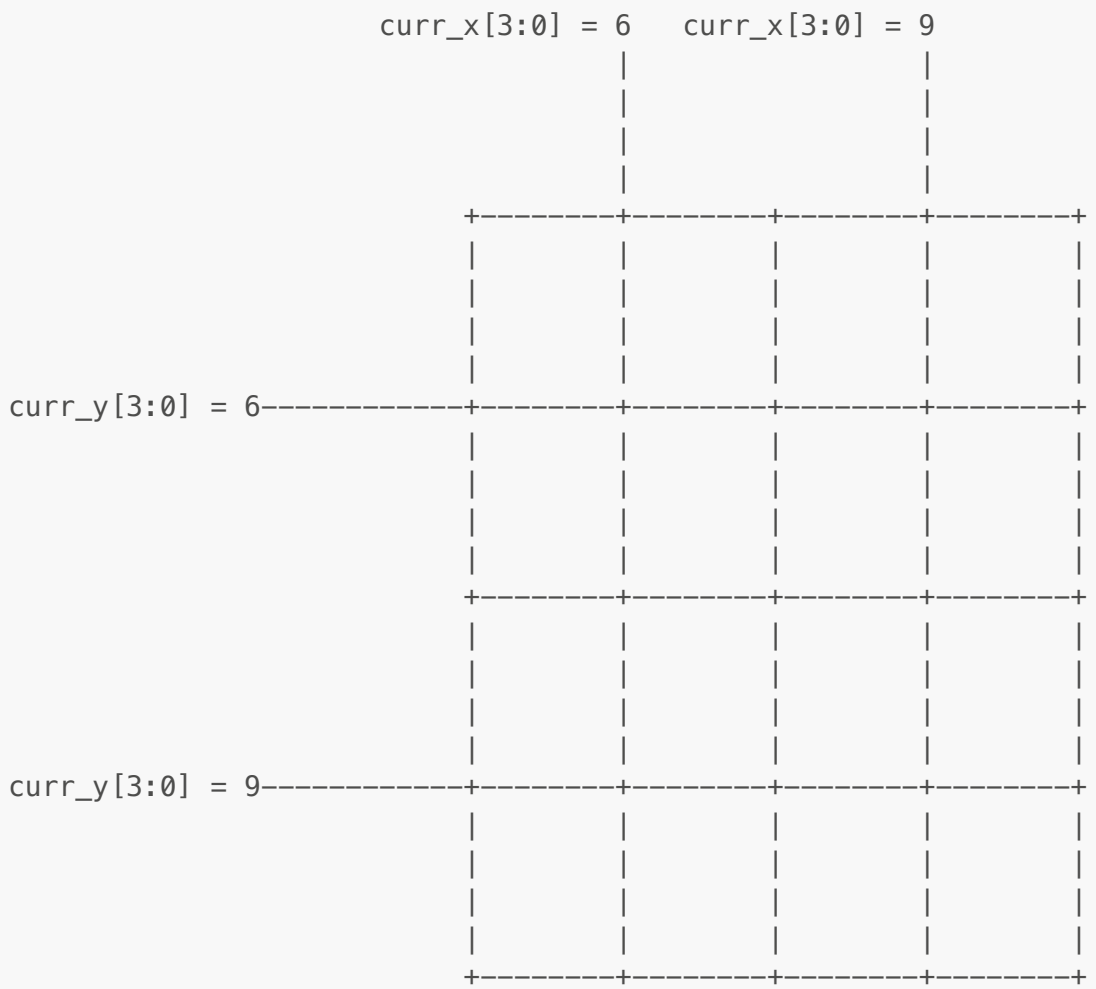
For example let's assume mod_x = 0 initially. At curr_x = 16, curr_x[4] == 1 if xor'ed curr_[4] ^ mod_x condition will pass and mod_x will be update to 1 and then at 32 curr_x[4] = 0 and mod_x = 1. Similarly for y.

```
if(curr_x[4] ^ mod_x) begin
    mod_x <= curr_x[4];
    ...
end
```

**Pacman Food**

To draw pacman's food which is a mere white square block of size 4x4 pixels there's no requirement for a sprite and a block memory. We just need to detect if it's a free space and a 4x4 block centered inside a 16x16 block. A similar trick like previous can be implemented here, we know that curr_x[3:0] and curr_y[3:0] is a 16x16 block which get's reset at every 16x16 block automatically. Now we just have to find 4x4 block at the center of these 16x16 block which is given by.

```
if(curr_x[3:0] >=6 && curr_x[3:0] <= 9 && curr_y[3:0] >= 6  && curr_y[3:0]
<= 9 && free_space) begin
    draw_food <= 1;
end
else begin
    draw_food <= 0;
end
```




As seen above, now whenever draw_food is set to 1 we can set r, g, b to white color. There's one problem that should be solved first i.e what if there's empty space and we draw food but pacman has already ate that in that case we shouldn't draw food. This a fundamental problem in the map, as it needs to represent which all free space blocks has food. So rather than representing map by binary we are supposed increase to 4 bit representation. We thought it's better to create a copy of a Map but this to RAM memory and not

ROM because game logic will update food once pacman is in the food position. Also after every reset, Food RAM mem should be filled up as well, this feature has not been implemented due to the time constraint.

**Controlling R, G, B**

This section describes how different sprites, food and wall has been displayed in respect to curr_x and curr_y.

**Map**

As explained in above sections, map is represented as a binary value, 0 for wall and 1 for free space. We can set specific variables accordingly.

```
reg [3:0] bg_r, bg_g, bg_b;
...
if(map_pix == 0) begin
    bg_r <= 9;
    ...
end
else begin
    bg_r <= 0;
    ...
end
```

And correspondingly set to the output of the module.

```
always@(posedge clk) begin

    if(pacman)
    ...
    else if(ghosts)
    ...
    else if(score)
    ...
    else if(food)
    ...
    else
        r <= bg_r;
        ...
endmodule
```

**Pacman and Ghosts**

Pacman and ghosts display is quite simple, all we need to do is get the position from the game logic and check if curr_x and curr_y is in the position, if yes then we set output accordingly. Similar to map, sprites are also written to a block mem and read depending y index.

```
if ((draw_x > pacman_blkpos_x
    && draw_x < (pacman_blkpos_x + 16))
    && (draw_y > pacman_blkpos_y
    && draw_y < (pacman_blkpos_y + 16))) begin
    ...
    if(pacman_sprite_row[draw_x - pacman_blkpos_x]) begin
        r <= blk_r;
        g <= blk_g;
        b <= blk_b;
    end
    else begin
        r <= 0;
        g <= 0;
        b <= 0;
    end
end
```

**Food**

From the above section we already discussed how we set food and now we just need to set to white color whenever it's true. The only challenge in displaying food is that, food map needs to be updated the game logic and read by the drawcon so we create a RAM with 2 ports where one port is used by the game logic and another one by drawcon.

```
if(!pacman && !ghosts && !score && food) begin
    r <= 15;
    g <= 15;
    b <= 15;
end
```

**Pose Changing Sprites**

We can conviniently change pose of the sprites as well for example pacman pose form wide mouth to closed mouth which make the user feel that pacman is eating food as it proceeds. Similarly goes for ghosts.

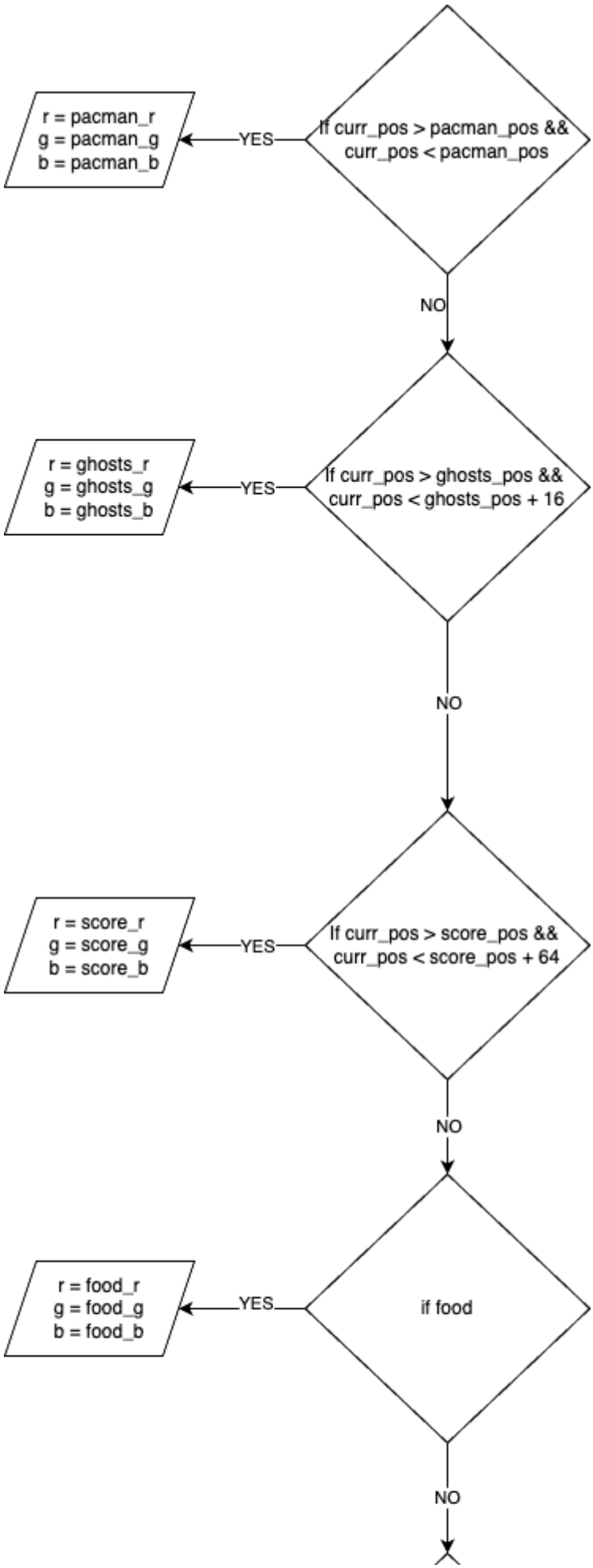
This can be easily, so let's say COE file contains a pose from 0-15 and another one from 15 - 31. We can just add a add multiple of 16 to the index pointing the memory.

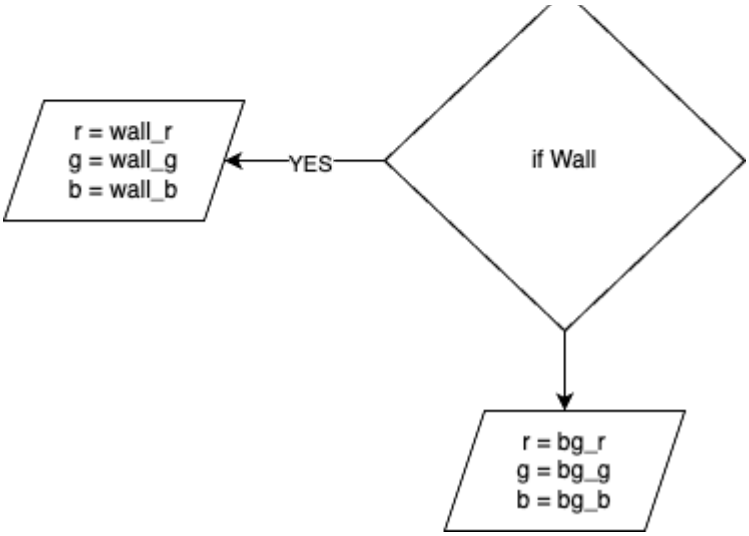
```
pacman_sprite_idx <= (draw_y - pacman_blkpos_y) + pacman_pos * 16;
```

**Score**

The score needs be 4 digit where each digit goes from 0-9 which can be represented as 4 bit. The digits can be sprites in the memory and selected based on the y indexing by a factor of 16 as mentioned above. To convert an 4 digit integer to 4 bit individual binary numbers will require a decimal to BCD converter.

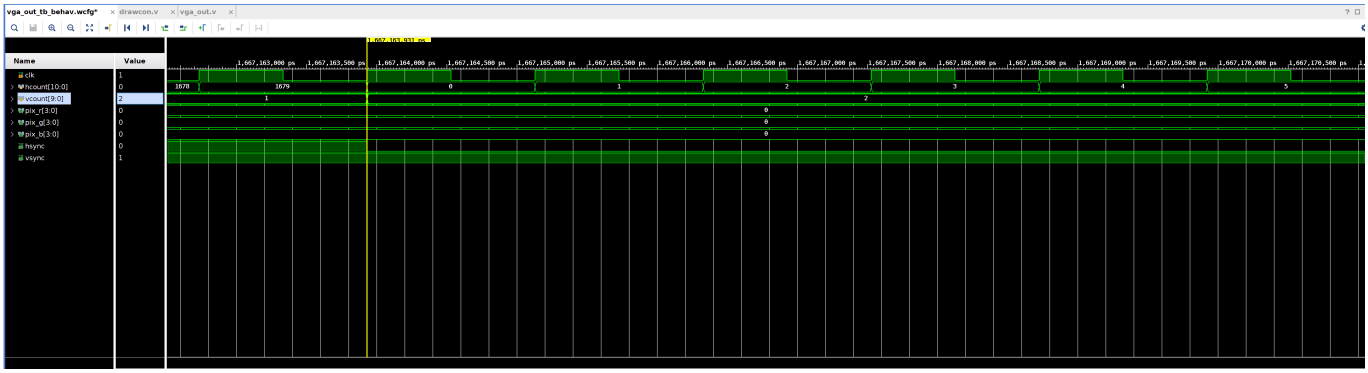
### Flow Chart

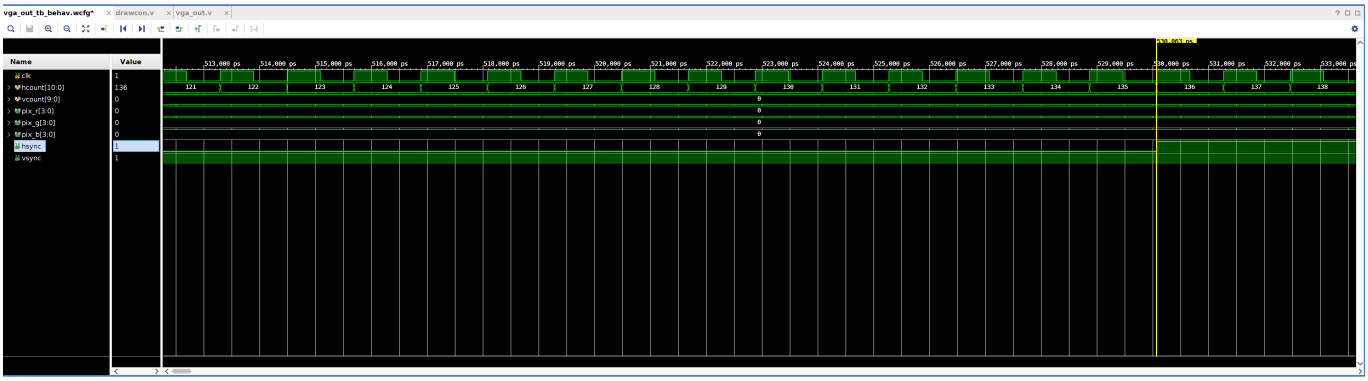

If curr_pos > pacman_pos &&
curr_pos < pacman_pos
YES
r = pacman_r
g = pacman_g
b = pacman_b
NO
If curr_pos > ghosts_pos &&
curr_pos < ghosts_pos + 16
YES
r = ghosts_r
g = ghosts_g
b = ghosts_b
NO
If curr_pos > score_pos &&
curr_pos < score_pos + 64
YES
r = score_r
g = score_g
b = score_b
NO
if food
YES
r = food_r
g = food_g
b = food_b
NO



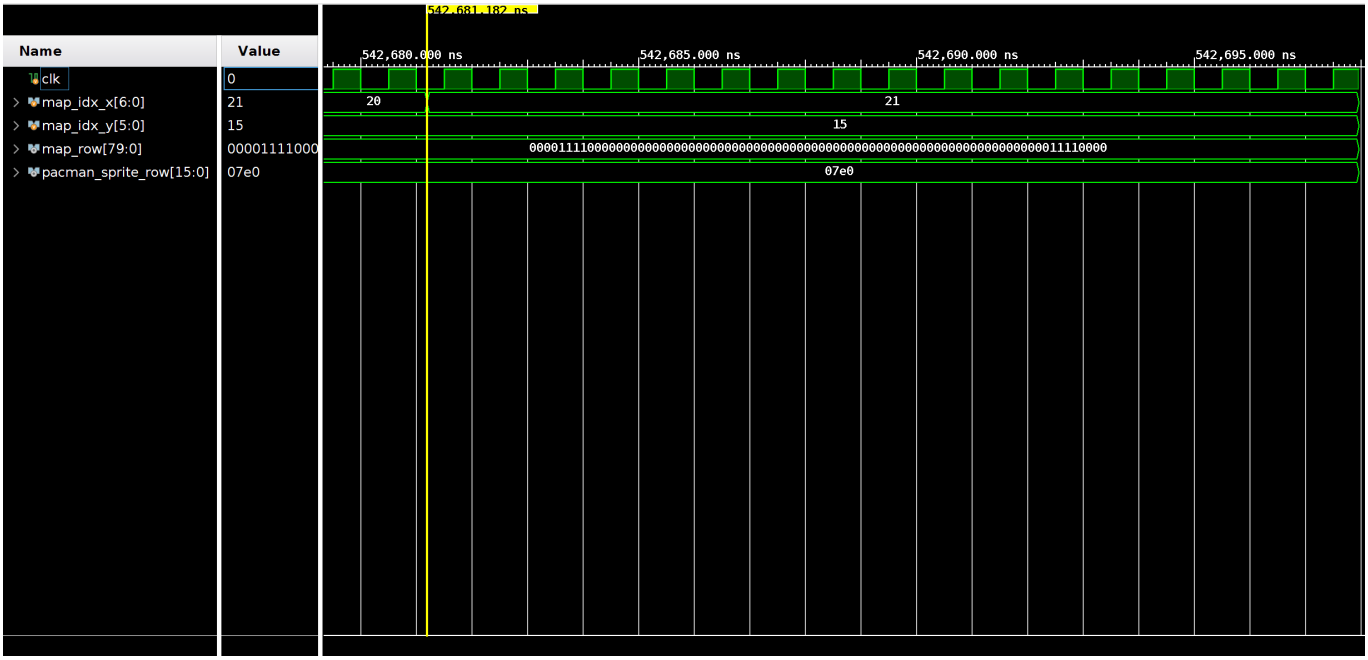

## Testing

Testing was done by simply by running a clk and fixing the pacman position. For VGA vsync, hsync values were monitored. It was also verified if hcount and vcount reaches max values and get reset immediately after that as seen in below graphs.





It can be observed in the below screenshot that




## Future Work

1. Optimizing food map
2. Reset food while resetting the game
3. Change pose only while moving

## References

1. Decimal to BCD Converter