Siege, a physics-based game for FPGA
Game report

Felix Bowyer

0. Contents

Initial task

VGA Protocol

2.1. Overview—The VGA protocol controls a screen with 5 outputs - 3 4-bit lines for red, green and blue pixel density, and 2 1-bit pulse control signals for horizontal and vertical sync.

VGA was created for CRT (Cathode ray tube) screens, which work by directing a single cathode ray across the screen, left to right, top to bottom. The intensity of the ray at a point determines the colour of the pixel at that point. Since a beam cannot be instantaneously moved, at the borders of the screen there are some ”blanking” cycles in which no pixels are drawn, but the beam is given time to move across the screen to where the next pixel needs to be drawn.

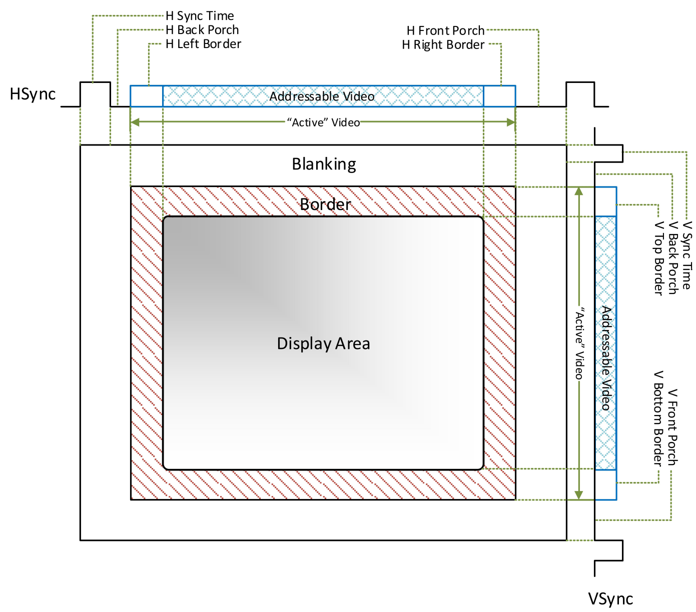

Fig. 1. VGA blanking, border and display regions.[1]

hsync and vsync are put high when the beam should be at the start of their respective blanking intervals. These sync signals tell the screen how to move the beam in order to produce the required resolution. There is also a small border region around the actual displayed area - this is mostly to prevent pixels being written during the blanking region. Although most screens are now LCD and therefore do no require a blanking interval, the VGA protocol is still the same regardless.

2.2. Sync circuit design—To prevent screen tearing, any processing which effects pixels shown on the screen should be done only in the blanking region. This stops the image being updated during drawing, preventing tearing.

For the game, hsync and vsync are controlled by two counters, `hcount_reg` and `vcount_reg`. `hcount_reg` is incremented every clock cycle, and reset to 0 at the end of a row of pixels (at `hcount_reg == 11'd1903`). `vcount_reg` is incremented at the end of each row of pixels, and reset at the end of the last row (where `vcount_reg == 10'd931`).

```verilog
// Assign hsync and vsync bools based on hcount and vcount
assign hsync = ~((hcount_reg >= 0) && (hcount_reg <= 151));
assign vsync = (vcount_reg >= 0) && (vcount_reg <= 2);

// Assign drawing region bool if in the drawing region
assign in_drawing_region = ((hcount_reg >= 384) && (hcount_reg <= 1823) && (vcount_reg >= 31) && (vcount_reg <= 930));
```

Fig. 2. Code assigning hsync, vsync, and in_drawing_region.

Not counting the blanking and border regions, we have a drawing region between virtual pixel 384 and 1823 horizontally, and 31 and 930 vertically. Therefore, the game has an addressable resolution of 1440x900.

The counter-controlled sync system is designed with the counters as D-type flip-flop counters, where hcount is enabled with the clock, and vcount is enabled once hcount reaches 1903 (at which point TC goes high). Comparators are used to assign a boolean value to hsync, vsync and in_drawing_region, based on the value of the counters.

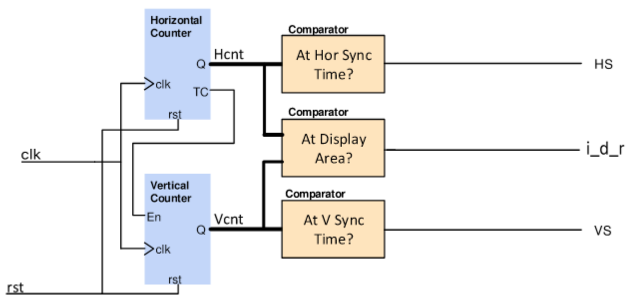

Fig. 3. Design of counter-sync system.

```verilog
always @(posedge clk) begin
    hcount_reg <= hcount_reg + 1;
    if (hcount_reg == 11'd1903) begin
        hcount_reg <= 0;
        vcount_reg <= vcount_reg + 1;
        if (vcount_reg == 10'd931) begin
            vcount_reg <= 0;
        end
    end
end
```

Fig. 4. Verilog counter system.

In Verilog, this logic is implemented using an always statement, as seen in Fig. 4. Each count is a register which is incremented as described above. An assign statement is used with comparators in Fig. 2 to implement the rest of the logic. When this program is synthesised and implemented on the FPGA board, the resulting circuit is optimised to be very similar to the design in Fig. 3.

2.3. Drawing a pixel—In order to actually draw a pixel to the screen, we introduce the 12-bit `pix` register (where `pix[11:8]` is the 4 bits for red, `pix[7:4]` is for green and `pix[3:0]` is for blue). As module output, we have `pix_{r,g,b}`, each of which is 4-bits and corresponds to the relevant pins of the VGA connector in the constraints file. hsync and vsync also directly correspond to a VGA pin in the contraints.

To choose the colour of a pixel to draw, we set pix to the desired 12-bit value. For example, to set it to a pure red, we could use `pix <= 12'hF00`. When drawing, hcount and vcount are effectively used as pointers to the coordinates of the pixel to be drawn this clock cycle. Therefore, we can define a region (using comparators) in which we should draw a colour. For example, `assign in_banner_region = ((hcount >= 384) && (hcount < 1824) &&`

`(vcount >= 31) && (vcount < 131));`* defines the rectangular region in which to draw the info bar at the top of the screen (which is a sprite and will be explained in more detail later - for now assume it is a solid colour). If we want to draw coloured pixels in that region, we assign `pix_{r,g,b}` to be coloured when hcount and vcount are in in the region, and black if not. Since 1 pixel is drawn per clock cycle in the drawing region, this doesn't use up any clock cycles unnecessarily to draw black outside the region.

```
assign pix_r = (draw_en) ? (pix[11:8]) : 4'b0000;
assign pix_g = (draw_en) ? (pix[7:4]) : 4'b0000;
assign pix_b = (draw_en) ? (pix[3:0]) : 4'b0000;
```

Fig. 5. pix{r,g,b} assignment.

Using this system, it becomes easy to define regions and colours to draw in those regions. However, to draw more complex images containing many colours, as well as transparency, we need to use sprites.

Sprites

In Siege, sprites are stored in Block RAM (BRAM), which is implemented through Xilinx IP. Each sprite has a corresponding BRAM IP, which is created using a `.coe` file, which contains 1 hexadecimal digit for each pixel in the sprite. This allows for a maximum of 16 colours per image. Although this number is low, the available BRAM on the FPGA board quite low, and so it is nessecary to decrease file sizes where possible.

To allow for a greater colour depth than the 4 bits of a hex digit, the game uses colour palettes. Each hex digit in a `.coe` file corresponds to a 12-bit (3-hex digit, 1 per colour channel) code in the sprite's `.mem` palette file. When sprites are drawn, the code in the `.coe` file is used as an index to look up the higher-depth colour to draw. This allows the sprite `.coe` files to be 3x smaller whilst retaining high colour depth. For most images, 16 colours is more than enough, and if more are required then most of the time the sprite can be split into separate sprites.

3.1. SPRITE FILE FORMAT—A sprite's coe file consists of two things.[2] Firstly, a `memory_initialization_radix`, which tells Vivado which base the data in the file is. Since we are using 1 hex character per colour, this was set to `16` for all files. Secondly, there is a `memory_initialization_vector`, which contains the actual sprite, in the form of a list of hex characters.

* Note that hcount and vcount values are offset to account for blanking and border regions.

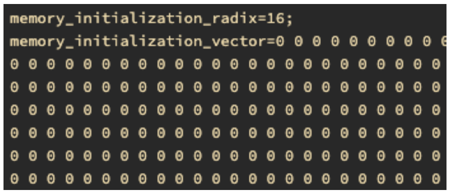

Fig. 6. A sprite .coe file.

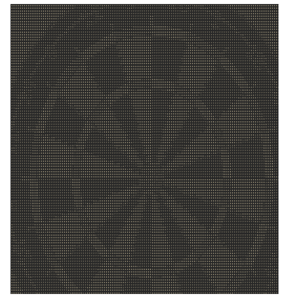

Fig. 7. Zoomed out.

For every coe file, there is a corresponding palette `.mem` file. This consists of a list of up to 16 12-bit hex-formatted colour codes (one for each hex digit in the coe file). When drawing the sprite, the hex digit in the coe file is used as an index to look up the correct colour code for the pixel.

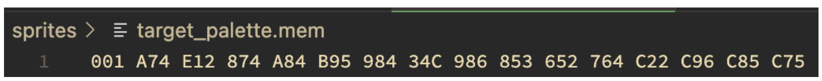

Fig. 8. A palette mem file.

3.2. Generating sprites—A Python script, `image_quantizer_coe.py`, was created from scratch using Python's PIL library to generate coe and palette mem files. A copy is included in the appendix.

It works by first importing the pixels of an image and adjusting the colour depth to have maximum 16 colours in the image. It then searches through the resulting image, making a list of all colours present. Whilst going through all the pixels, the script assembles the coe file based on the colour it finds. Finally, the program assembles the palette file from the list of colours. Using these files, we can create a ROM IP block holding the coe file, and draw the sprite by decoding each hex value with the palette.

3.3. Drawing the info bar—The info bar is a sprite drawn at the top of the screen. In order to draw this sprite, we must set up a few things, starting with instantiating the IP block.

3.3.1. ROM IP block—ROM IP works by taking an address wire `addra`. The data at this address is then output on a wire `douta`. In this case, each input memory address corresponds to a 4-bit pixel value as imported from the coe file - once the ROM is clocked, this value is output on `douta`. In order to run the VGA output, we earlier used a clock, which we can use for our ROM IP.

3.3.2. Selecting a pixel to draw—So far, we have a wire `banner_out`, which contains the pixel value, and `banner_px_count`, containing the index of the pixel to lookup in block memory. Therefore, to access each pixel, we must increment `banner_px_count` by 1 after

```
banner_rom banner (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(banner_px_count),  // input wire [17 : 0] addra
    .douta(banner_out)  // output wire [3 : 0] douta
);
```

Fig. 9. The info bar's IP instantiation.

drawing a pixel, allowing us to draw successive pixels in the order they are in memory.

We only want to draw pixels for a sprite in a certain area. To achieve this, we assign a boolean value representing rectangular region for the sprite to be drawn in, similarly to how `in_drawing_region` was defined. When the program is drawing a pixel in this region, `banner_px_count` should be incremented. This count is reset at the end of drawing each frame so that the 1st pixel is drawn first each time.

```
assign in_banner_region = ((hcount >= 384) && (hcount < 1824) && (vcount >= 31) && (vcount < 131));
if (in_banner_region) begin
  banner_px_count <= banner_px_count + 1;
end
```

Fig. 10. Defining the info banner region, and incrementing the pixel count.

3.3.3. Decoding a pixel value—To draw the correct colour, we must use the value from the ROM's output to look up the colour in the palette. To achieve this, the palette mem file is imported into a 2d register using `$readmemh`. Then, the output from the block memory `banner_px_count` is used as an index into the palette array to obtain the colour of the pixel to draw. This actual colour code to be drawn is then stored in the `pix` 12-bit register. As described in §2, the value in the pix register is assigned to the outputs `pix_{r,g,b}`, in order to draw that colour to the screen.

```
if (in_banner_region) begin
  draw_en <= 1;
  pix <= banner_palette[banner_out];
end
// assign current pixel values to outputs
assign pix_r = (draw_en) ? (pix[11:8]) : 4'b0000;
assign pix_g = (draw_en) ? (pix[7:4]) : 4'b0000;
assign pix_b = (draw_en) ? (pix[3:0]) : 4'b0000;
```

Fig. 11. Decoding a pixel value with the palette, and assigning it to outputs.

This system of using a pixel counter, which is incremented after a pixel from the sprite is drawn, changing the colour of the next pixel to draw is very successful in drawing sprites, and even allows multiple sprites to be drawn on top of each other.

3.4. DRAWING OTHER SPRITES—The info bar is in many ways an ideal sprite to draw - it has a fixed position, is a perfect rectangle, and doesn't require other sprites to be drawn on top of it. However, in the game, there are sprites which aren't so ideal - how do we draw them?

3.4.1. Non-fixed position sprites—The cannonball is a user-controlled moving sprite, which is shot across the screen. Therefore, it's position needs to be able to change.
The easiest way to do this under the current system is to assign the sprite's drawing region to be dependent on registers holding the current position of the cannonball. We assign the region as follows:

```
assign in_cannonball_region = ((hcount >= (384 + cannonball_x)) &&
(hcount < (384 + cannonball_x + cannonball_size)) && (vcount >= (31 +
cannonball_y)) && (vcount < (31 + cannonball_y + cannonball_size)));
```

This causes the region that the cannonball is rendered in to vary with `cannonball_x` and `cannonball_y`, since `in_cannonball_region` is just a combinatorial output wire of it's products.

3.4.2. Non-rectangular sprites—Not all sprites in the game are rectangular. In particular, the cannonball and target sprites are rough circles. Since for both sprites we need to be able to draw behind them, it is necessary to have transparency.

To achieve this, a sprite's palette is edited so that the background colour is `001`. For example, if the image used to generate the script has an `FFF` white background, then the palette can be manually edited to change `FFF` to `001`. Then, in order to make the background pixels render as transparent, there is a check on assigning `pix`.
Checking the value of `target_palette[target_out]` i.e. the colour of the pixel to render, and only assigning pix if it isn't 001, stops transparent areas from affecting the value of pix, allowing objects behind it to also render correctly.

```
if (in_target_region && target_palette[target_out] != 12'h001) begin
  draw_en <= 1;
  pix <= target_palette[target_out];
end
```

Fig. 12. Checking for transparent pixels.

3.4.3. Changing sprites—The timer sprite shows the time passed in the game by increasing in size over time. Similarly to transparency, this is achieved by a check on the assignment. We use `if (in_timer_region && (hcount < (394 + game_time + 1)) ) begin` as a condition (game_time slowly increases). This is a quick way to stop any section past a certain point (394 + game_time+ 1) from rendering. The power bar works in a similar way.

3.4.3. Drawing priority—Some sprites need to be drawn on top of others when drawing within the region of multiple sprites. The order of rendering is resolved by making several assignments to `pix`, of which only the last is still stored in pix when `pix_{r,g,b}` are assigned.

Game design

4.1. Menus—There is a start menu when the game is first started, and an end screen when the game is done. Each one has a sprite directing the player what to do to start the game. During the menu, no other sprites are shown.

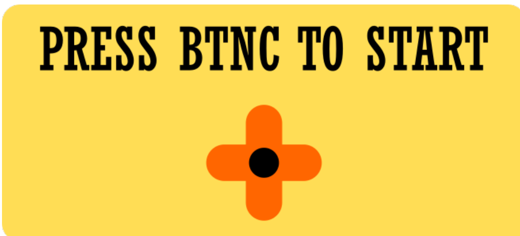

Fig. 13. The start screen.

Fig. 14. The end screen.

To make sure only necessary things are being rendered and processed in the menus, a register `game_state` is used. If `game_state == 0`, the start screen is active. If `game_state == 1`, it is in the actual game-playing state, and if `game_state == 2`, the game over screen displays. This can be thought about as a simple Finite State Machine as follows:

State	State 0 action	State 1 action	State 2 action
0	-	BTNC	-
1	-	-	game_time==230
2	BTNL	BTNR	-

Within the game, there are two separate `always @(posedge clk)` blocks - one for game logic, and one for rendering.
Logic states 0 and 2 only contain very simple code, to detect button pushes, whilst state 1 contains code to do with detecting the firing, simulating the movement, and reacting to collisions of the cannonball.
Similarly, within the rendering block, states 0 and 2 only render the menu sprites. When the game transitions to state 1 (game state), a number of game parameters are reset.

4.2. Physics simulation—The cannonball is shot by the player by holding down btnc, charging the power bar, and letting go. The amount of charge in the power bar affects the movement of the cannonball - higher power causes the ball to go further. This is achieved by decreasing the effect of gravity the higher the power is.

Once per frame (when hcount and vcount == `10'd0` to prevent screen tearing), launch_pwr is incremented. This continues until launch_pwr == 25, at which it resets to 1 - this means that at launch, launch_pwr can take a value between 1 and 24 inclusive.
As previously mentioned, cannonball_x and cannonball_y are used to represent the x and y position of the cannonball. To adjust this, `cannonball_xvel` and `cannonball_yvel` are set to different integer values, and `cannonball_{x,y}` is incremented by `cannonvall_{x,y}vel`

every frame.[†]

```
end else if (BTNC) begin
  if (launch_pwr == 25) begin
    launch_pwr <= 1;
  end else begin
    launch_pwr <= launch_pwr + 1;
  end
end
```

Fig. 15. Increasing launch power.

The cannonball bounces on the top and bottom of the gameplay area - this is done by detecting when it is out of bounds (by seeing if the cannonball's region is out of the gameplay area's region), and inverting the y velocity.[‡]

The strength of gravity, determined by pwr, is controlled by `accel_app_count`, which is set to be equal to launch_pwr at launch. It is then decremented until it reaches 0, at which point cannonball_yvel is decremented.[§] This means that yvel is decreased faster when pwr is less, effectively making gravity stronger with lower power shots.

4.3. 7-SEGMENT DISPLAY—During gameplay, the onboard 7-segment display shows the game score (1 point is gained each time a target is hit). The score is stored in a 32-bit `score` register, which at the start is the game is set to 0. `seginterface.v` and `sevenseg.v` from earlier FPGA labs are used to instantiate a segintergace `seg`, whose outputs are passed through to the relevant board pins via `sg` and `an`.

```
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { sg[0] }];
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { sg[1] }];
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { sg[2] }];
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { sg[3] }];
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { sg[4] }];
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { sg[5] }];
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { sg[6] }];
```

Fig. 16. 7-seg output contraints.

4-bit sections, each representing a hex digit of `score`, are passed to `seg`'s `dig{0..7}`. By default, this lets the score be displayed, but in hexadecimal - to get it to display in decimal, a hacky workaround is used.
As explained before, score is incremented every time the cannonball and target collide.[¶] The target collision area is defined with `in_target_coll`. Every time the score is incremented, a check is performed to see if any digits == `4'd9` i.e. will be incremented to `4'd10` on the next

[†] Note that to allow for negative velocities without converting to/from two's compliment, cannonball_x,yvel == 32 actually represents a velocity of 0. Values below 32 are negative velocity, and above are positive velocity.
[‡] This means adding yvel-32 if moving upwards, and taking away 32-yvel if moving downwards initially.
[§] This does introduce a known bug with yvel overflowing, which in hindsight is extremely easy to fix by enforcing a terminal velocity.
[¶] The target collision area is slightly smaller than the target sprite, since a rectangle around the target would count collisions through the corner of the area, which does not look like part of the round target.

clock cycle and therefore display as a hex A. If this is the case, that digit is reset to 0, and the next digit is incremented, making the 7-segment display look like it's displaying a natural base 10 number.

Testing

Two main types of test were used; agile testbenches and unit tests.
Agile tests took place throughout development, after implementation of a feature. Initially, when developing the base vga output system, we used a testbench to ensure hsync and vsync were updated correctly. With programs running on bare hardware, it is difficult to diagnose logic errors, since there is no interactive terminal for debugging. Therefore, it is very useful to analyse input, output and register traces to see what is going wrong. Test benches were also used successfully to diagnose overflow issues with buffer under/overflows with cannonball velocity.

Several tests were also used towards the end of development, targeting different parts of the program, as follows.

Program section	Test no.	Description	Result	Notes
VGA output	1	hsync and vsync are timed correctly (by lab sheet specification)	PASS	Failed initially, hsync was out of sync by 1 clock cycle
	2	Test object does not go out of display area range	PASS	Test object used only during development
	3	Screen tearing doesn't occur when object moving	PASS	Failed initially, changed processing to happen only on hcount, vcount == 0
	4	Screen resolution is correct (1400x900)	PASS	Measured by drawing a set number of pixels across the screen
Sprites	5	Sprites loaded correctly into BRAM	PASS	Tested together. Tested using black and white sprites without palettes
	6	Sprite pixel counts function		
	7	Sprite palettes work correctly	PASS	Initially failed due to little endian indexing, which shuffled colours.

Program section	Test no.	Description	Result	Notes
Gameplay	8	Power and time bar progressively render correctly	PASS	Power initially failed, since pixel count wasn't being updated on non-render of pixel.
	9	Menus function correctly	PASS	Initially didn't reset game parameters on game restart from end screen.
Physics	10	Power proportionally affects gravity	PASS	Confirmed with testbench, analysing frequency of gravity application.
	11	Cannonball doesn't exhibit unpredictable behaviour	MIXED	Most bugs were fixed - however in the submitted version, a buffer underflow was still present as described previously. This has been fixed since, but is not in the submitted version.
	12	Cannonball stays within bounds	PASS	Cannonball may be partially out of bounds for a single frame, though this is intended and necessary for bouncing to properly occur.

Reflection

I enjoyed this coursework a lot. It allowed me to put into practice what I have learnt about in theory, which is always a good thing. FPGA programming presents unique challenges that come with programming at such a low level, so I value now having experience with that. In particular, detecting logical errors with testbenches, and importing and properly formatting block memory has been unique and challenging.
Creating a game from the ground up has been a good task, because it requires lots of different tasks. Creating reliable VGA output, handling asynchronous inputs, optimising sprites as to not use up all of the block memory, dealing with block memory data lines and endianness, and simulating physics without easy access to floating point numbers have been fun tasks personally to work out.

There are many parts of the game I am proud to have created. Of course, I am happy that the game is complete and fully functioning, but I am particularly happy with how interactive the whole game is. There are some features (on-screen moving timer, multiple cannon angles, light-up 7-seg displayed score system, and realistic physics simulation) which at the start of the project I considered features to implement if there was enough time at the end. I am also very proud of creating the sprite system - generating coe and palette mem files, reading them into memory correctly, using the palette to decode them, implementing transparency, and drawing them to the screen correctly. It is very satisfying to see my design work correctly having spent so much time implementing it.

If I were to revisit project again, firstly I would iron out the few remaining bugs in the physics simulation to do with buffer overflows. Since the realistic physics was implemented towards the end, less thought was given to it's design, and I feel that spending a little longer designing it would have helped. Another thing I would like to change is to do with sprites; it would be useful to be able to scale sprites up, rather than including huge repeated sections of block memory. This would take some time to implement, but it would free up enough memory to add a full background image to the game, as well as introduce more sprites. It would also be good to implement a frame buffer,[3] to allow processing to take place outside of the frame buffer regions. The final change I would make is just to spend some time tidying up the code - whilst it is high quality, it would definitely benefit from parts being moved to different verilog files (as was done with `vga_timing.v` and others). The code around managing block memory could also be greatly reduced, since much of it is repeated.

Over the project, I learnt a lot about how sprite systems work, why palettes are necessary, how and why the VGA protocol works, and how to deal with memory in an extremely low-level way. I also learnt more about controlling external outputs through setting pins, such as with the 7-segment display or LED lights. I also liked the way the project guided me to learning the necessary knowledge through labs.

Appendix

Main file, managing rendering sprites and game logic: `vga_out.v`.

```
`timescale 1ns / 1ps

module vga_out (
    input clk,
    //input sw[2:0],
    input BTNU,
    input BTND,
    input BTNL,
    input BTNR,
    input BTNC,
    input RST,
    output [3:0] pix_r,
    output [3:0] pix_g,
    output [3:0] pix_b,
    output hsync,
    output vsync,
    output [6:0] sg,
    output [7:0] an,
    output [15:0] LED
);

  // Declared as wires - assigned to in combinatorial statement
  wire [10:0] hcount;
  wire [9:0] vcount;
```

```
wire in_drawing_region;

reg [31:0] score = 0;
reg [15:0] led_reg = 16'h0000;
assign LED = led_reg;

seginterface seg (
    .clk(clk),
    .dig7(score[31:28]),
    .dig6(score[27:24]),
    .dig5(score[23:20]),
    .dig4(score[19:16]),
    .dig3(score[15:12]),
    .dig2(score[11:8]),
    .dig1(score[7:4]),
    .dig0(score[3:0]),
    .a(sg[0]),
    .b(sg[1]),
    .c(sg[2]),
    .d(sg[3]),
    .e(sg[4]),
    .f(sg[5]),
    .g(sg[6]),
    .an(an)
);

// set up vga timings
vga_timing vga_timer (
    .clk(clk),
    .hsync(hsync),
    .vsync(vsync),
    .hcount_reg(hcount),
    .vcount_reg(vcount),
    .in_drawing_region(in_drawing_region)
);

reg width = 1440;
reg height = 900;

reg [6:0] squ_size_x = 49;
reg [6:0] squ_size_y = 49;
reg [10:0] squ_x = 200;
reg [9:0] squ_y = 200;
reg lr = 1;
reg ud = 1;

wire blk_en = 1;

// setup banner image
reg  [17:0] banner_px_count = 0;
wire [ 3:0] banner_out;
banner_rom banner (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(banner_px_count),  // input wire [17 : 0] addra
    .douta(banner_out)  // output wire [3 : 0] douta
);
wire in_banner_region;
assign in_banner_region = ((hcount >= 384) && (hcount < 1824) && (vcount >= 31) && (
  vcount < 131)); // may need adjusting
reg [11:0] banner_palette[0:15];

// setup brick wall image
reg [17:0] wall_px_count = 0;
wire [3:0] wall_out;
wall_rom wall (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(wall_px_count),  // input wire [17 : 0] addra
    .douta(wall_out)  // output wire [3 : 0] douta
);
wire in_wall_region;
```

```
assign in_wall_region = ((hcount >= 1624) && (hcount < 1824) && (vcount >= 131) && (vcount < 931));
reg [11:0] wall_palette[0:15];

// setup target image
reg [14:0] target_px_count = 0;
wire [3:0] target_out;
target_rom target (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(target_px_count),  // input wire [14 : 0] addra
    .douta(target_out)  // output wire [3 : 0] douta
);
wire in_target_region;
assign in_target_region = ((hcount >= 1649) && (hcount < 1799) && (vcount >= 156) && (vcount < 306));
reg [11:0] target_palette[0:15];

// setup target image 2
reg [14:0] target2_px_count = 0;
wire [3:0] target2_out;
target_rom target2 (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(target2_px_count),  // input wire [14 : 0] addra
    .douta(target2_out)  // output wire [3 : 0] douta
);
wire in_target2_region;
assign in_target2_region = ((hcount >= 1649) && (hcount < 1799) && (vcount >= 456) && (vcount < 606));

// setup cannons
reg  [15:0] cannon_px_count = 0;
wire [ 3:0] cannon_30_out;
wire [ 3:0] cannon_45_out;
wire [ 3:0] cannon_60_out;
cannon_30_rom cannon_30 (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(cannon_px_count),  // input wire [15 : 0] addra
    .douta(cannon_30_out)  // output wire [3 : 0] douta
);
cannon_45_rom cannon_45 (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(cannon_px_count),  // input wire [15 : 0] addra
    .douta(cannon_45_out)  // output wire [3 : 0] douta
);
cannon_60_rom cannon_60 (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(cannon_px_count),  // input wire [15 : 0] addra
    .douta(cannon_60_out)  // output wire [3 : 0] douta
);
wire in_cannon_region;
assign in_cannon_region = ((hcount >= 384) && (hcount < 584) && (vcount >= 731) && (vcount < 931));
reg [11:0] cannon_palette[0:15];

// setup cannonball
reg [10:0] cannonball_x = 1;
reg [9:0] cannonball_y = 859;
reg [6:0] cannonball_size = 7'd40;
reg [5:0] cannonball_xvel = 0;
reg [6:0] cannonball_yvel = 32;  // essentially 0
reg cannonball_shot = 0;
reg [5:0] launch_pwr = 0;
reg [5:0] accel_app_count = 0;

reg [14:0] cannonball_px_count = 0;
wire [3:0] cannonball_out;
cannonball_rom cannonball (
    .clka (clk),  // input wire clka
```

```
    .ena  (blk_en),  // input wire ena
    .addra(cannonball_px_count),  // input wire [10 : 0] addra
    .douta(cannonball_out)  // output wire [3 : 0] douta
);
wire in_cannonball_region;
assign in_cannonball_region = ((hcount >= (384 + cannonball_x)) && (hcount < (384 + cannonball_x + cannonball_size)) && (vcount >= (31 + cannonball_y)) && (vcount < (31 + cannonball_y + cannonball_size)));
reg  [11:0] cannonball_palette[0:15];


// setup power bar
reg  [13:0] pwr_px_count = 0;
wire [ 3:0] pwr_out;
pwr_rom pwr (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(pwr_px_count),  // input wire [13 : 0] addra
    .douta(pwr_out)  // output wire [3 : 0] douta
);
wire in_pwr_region;
assign in_pwr_region = ((hcount >= 384) && (hcount < 634) && (vcount >= 181) && (vcount < 231));
reg [11:0] pwr_palette[0:15];

// setup timer bar frame
reg [13:0] timerframe_px_count = 0;
wire [3:0] timerframe_out;
timerframe_rom timerframe (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(timerframe_px_count),  // input wire [13 : 0] addra
    .douta(timerframe_out)  // output wire [3 : 0] douta
);
wire in_timerframe_region;
assign in_timerframe_region = ((hcount >= 384) && (hcount < 634) && (vcount >= 131) && (vcount < 181));
reg  [11:0] timerframe_palette [0:15];

// setup timer bar fill
reg  [13:0] timer_px_count = 0;
wire [ 3:0] timer_out;
timer_rom timer (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(timer_px_count),  // input wire [13 : 0] addra
    .douta(timer_out)  // output wire [3 : 0] douta
);
wire in_timer_region;
assign in_timer_region = ((hcount >= 394) && (hcount < 624) && (vcount >= 141) && (vcount < 171));
reg [11:0] timer_palette[0:15];

// setup start screen
reg [15:0] start_px_count = 0;
wire [3:0] start_out;
start_rom startscr (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
    .addra(start_px_count),  // input wire [15 : 0] addra
    .douta(start_out)  // output wire [3 : 0] douta
);
wire in_start_region;
assign in_start_region = ((hcount >= 939) && (hcount < 1269) && (vcount >= 425) && (vcount < 575));
reg [11:0] start_palette[0:15];

// setup end screen
reg [15:0] end_px_count = 0;
wire [3:0] end_out;
end_rom endscr (
    .clka (clk),  // input wire clka
    .ena  (blk_en),  // input wire ena
```

```
    .addra(end_px_count),  // input wire [15 : 0] addra
    .douta(end_out)  // output wire [3 : 0] douta
);
wire in_end_region;
assign in_end_region = ((hcount >= 939) && (hcount < 1269) && (vcount >= 402) && (vcount
   < 597));
reg [11:0] end_palette[0:15];

// setup target collision boxes
wire in_target_coll; // covers target1 and 2: may need to seperate if you want different
   scores for each target
assign in_target_coll = (((cannonball_x + cannonball_size) >= 1288) && (cannonball_x <
  1392) && ( (((cannonball_y + cannonball_size) >= 148) && (cannonball_y < 252)) || (((
  cannonball_y + cannonball_size) >= 448) && (cannonball_y < 552))) );

reg [7:0] game_time = 0;
reg [4:0] time_inc = 0;

// import sprite palettes
initial begin
  $readmemh("img_palette.mem", palette);
  reg [11:0] banner_palette[0:15];
  $readmemh("banner_palette.mem", banner_palette);
  $readmemh("wall_palette.mem", wall_palette);
  $readmemh("target_palette.mem", target_palette);
  $readmemh("cannon_palette.mem", cannon_palette);
  $readmemh("cannonball_palette.mem", cannonball_palette);
  $readmemh("pwr_palette.mem", pwr_palette);
  $readmemh("timerframe_palette.mem", timerframe_palette);
  $readmemh("timer_palette.mem", timer_palette);
  $readmemh("start_palette.mem", start_palette);
  $readmemh("end_palette.mem", end_palette);
end


always @(posedge clk) begin
  // on each pixel increment pixel counts for each image
  if (in_banner_region) begin
    banner_px_count <= banner_px_count + 1;
  end
  if (in_wall_region) begin
    wall_px_count <= wall_px_count + 1;
  end
  if (in_target_region) begin
    target_px_count <= target_px_count + 1;
  end
  if (in_target2_region) begin
    target2_px_count <= target2_px_count + 1;
  end
  if (in_cannon_region) begin
    cannon_px_count <= cannon_px_count + 1;
  end
  if (in_cannonball_region) begin
    cannonball_px_count <= cannonball_px_count + 1;
  end
  if (in_pwr_region) begin
    pwr_px_count <= pwr_px_count + 1;
  end
  if (in_timerframe_region) begin
    timerframe_px_count <= timerframe_px_count + 1;
  end
  if (in_timer_region) begin
    timer_px_count <= timer_px_count + 1;
  end
  if (in_start_region) begin
    start_px_count <= start_px_count + 1;
  end
  if (in_end_region) begin
    end_px_count <= end_px_count + 1;
  end


  // reset pixel counts at end of frame
```

```
    if (hcount == 11'd0 && vcount == 10'd0) begin
      banner_px_count <= 0;
      wall_px_count <= 0;
      target_px_count <= 0;
      target2_px_count <= 0;
      cannon_px_count <= 0;
      cannonball_px_count <= 0;
      pwr_px_count <= 0;
      timerframe_px_count <= 0;
      timer_px_count <= 0;
      start_px_count <= 0;
      end_px_count <= 0;
    end
  end


  reg [1:0] cannon_num = 0;

  reg last_btnr = 0;
  reg last_btnc = 0;

  reg [1:0] game_state = 0;
  // 0 = start screen
  // 1 = playing
  // 2 = game over

  reg left_start = 0;

  wire cannonball_yvel_neg;
  assign cannonball_yvel_neg = cannonball_yvel < 32;

  always @(posedge clk) begin

    if (RST) begin
        game_state <= 0;
        game_time <= 0;
        score <= 32'h00000000;
        cannonball_shot <= 0;
        cannonball_x <= 1;
        cannonball_y <= 859;
        cannonball_xvel <= 0;
        cannonball_yvel <= 0;
    end

    if (game_state == 0) begin
      // start screen
      if (!BTNC && last_btnc) begin
        game_state <= 1;
      end
      last_btnc <= BTNC;

    end else if (game_state == 1) begin

      /*
      === GAME LOGIC BLOCK ===
      */

      // change cannon number when btnr is pressed
      if (BTNR == 1'b1 && last_btnr == 1'b0) begin
        if (cannon_num != 2'd2) begin
          cannon_num <= cannon_num + 1;
        end else begin
          cannon_num <= 0;
        end
      end

      // between each frame (so we avoid tearing!!)
      if (hcount == 11'd0 && vcount == 10'd0) begin

        // set LEDs off when needed
        if (cannonball_shot) begin
          led_reg <= 16'h0000;
        end
```

```

        // increment game time
        time_inc <= time_inc + 1;
        if (time_inc == 0) begin
          game_time <= game_time + 1;
        end

        // end game is applicable
        if (game_time == 230) begin
          game_state <= 2;
        end

        if (accel_app_count == (6'd2 + launch_pwr[5:1])) begin  //maybe not 2
          // every the acceleration pause reaches 1/2 of launch power, reset the
          // count (and apply the acceleration)
          accel_app_count <= 0;
        end else begin
          accel_app_count <= accel_app_count + 1;
        end

        // Check if off screen, hasn't been shot, or on target and reset needed
        if (!cannonball_shot || cannonball_x > 1440 || in_target_coll) begin
          if (cannonball_x > 1440 || in_target_coll) begin
            // if off screen or on target, reset
            launch_pwr <= 0;

            // If target just hit, light up and increase score
            if (in_target_coll) begin
              led_reg <= 16'hFFFF;
              score   <= score + 1;

              // Correct score for display
              if (score[3:0] == 4'd9) begin
                score[3:0] <= 4'd0;
                score[7:4] <= score[7:4] + 4'd1;
                if (score[7:4] == 4'd9) begin
                  score[7:4]  <= 4'd0;
                  score[11:8] <= score[11:8] + 4'd1;
                  if (score[11:8] == 4'd9) begin
                    score[11:8]  <= 4'd0;
                    score[15:12] <= score[15:12] + 4'd1;
                    // impossible to score higher than this - no further correction
                  end
                end
              end
            end

          end else if (BTNC) begin
            // if BTNC is held, increase launch power until it overflows
            if (launch_pwr == 25) begin
              launch_pwr <= 1;
            end else begin
              launch_pwr <= launch_pwr + 1;
            end
          end

          // reset the cannonball
          cannonball_shot <= 0;
          cannonball_x <= 1;
          cannonball_y <= 859;
          cannonball_xvel <= 0;
          cannonball_yvel <= 0;

          // if canonnball has been shot and is on screen
          // change velocity and animate cannonball
        end else if (cannonball_shot) begin

          // Change x pos
          cannonball_x <= cannonball_x + (cannonball_xvel - 32);


          // handle bounds, swap velocity
          // if bouncing on top of screen
```

```
          if (cannonball_y < 101) begin
            cannonball_y <= 101;
            // invert velocity if positive
            if (!cannonball_yvel_neg) begin
              cannonball_yvel <= 32 - ((cannonball_yvel - 32) / 2);
            end

            // if bouncing on bottom of screen
          end else if (cannonball_y > (900 - cannonball_size)) begin
            cannonball_y = (900 - cannonball_size);
            // invert velocity if negative
            if (cannonball_yvel_neg) begin
              cannonball_yvel <= 32 + ((32 - cannonball_yvel) / 2);
            end

            // if not bouncing or at terminal velocity
          end else if (cannonball_yvel != 0) begin
            if (!cannonball_yvel_neg) begin
              // if moving up, add yvel-32
              cannonball_y <= cannonball_y - (cannonball_yvel - 32);
            end else begin
              // if moving down, remove 32-yvel
              cannonball_y <= cannonball_y + (32 - cannonball_yvel);
            end
            //cannonball_y <= cannonball_y - (cannonball_yvel - 32); // MAYBE + not -? not sure
          end

          // Gravity
          if (accel_app_count == 0) begin
            cannonball_yvel <= cannonball_yvel - 1;

          end
        end

      end else if (!cannonball_shot && BTNC == 1'b0 && last_btnc == 1'b1) begin
        // BTNC let go
        // Stop shooting on first press of BTNC (leaving start screen)
        if (left_start) begin
          cannonball_shot <= 1;
          if (cannon_num == 2'd0) begin
            cannonball_xvel <= 39;
            cannonball_yvel <= 44;
          end else if (cannon_num == 2'd1) begin
            cannonball_xvel <= 42;
            cannonball_yvel <= 42;
          end else begin
            cannonball_xvel <= 44;
            cannonball_yvel <= 39;
          end
        end

      end else begin
        left_start <= 1;
      end

      last_btnr <= BTNR;
      last_btnc <= BTNC;

      // end screen logic
    end else if (game_state == 2) begin
      if (BTNL) begin
        game_state <= 0;
        game_time <= 0;
        score <= 32'h00000000;
        cannonball_shot <= 0;
        cannonball_x <= 1;
        cannonball_y <= 859;
        cannonball_xvel <= 0;
        cannonball_yvel <= 0;
      end else if (BTNR) begin
        game_state <= 1;
        game_time <= 0;
```

```
      left_start <= 0;
      score <= 32'h00000000;
      cannonball_shot <= 0;
      cannonball_x <= 1;
      cannonball_y <= 859;
      cannonball_xvel <= 0;
      cannonball_yvel <= 0;
    end
  end

end

reg [11:0] pix = 0;
reg draw_en;

always @(posedge clk) begin

  draw_en <= 0;

  if (game_state == 0) begin
    if (in_start_region) begin
      draw_en <= 1;
      pix <= start_palette[start_out];
    end
    if (in_banner_region) begin
      draw_en <= 1;
      pix <= banner_palette[banner_out];
    end

  end else if (game_state == 1) begin
    // game play

    // Drawing logic block
    // for each pixel in the frame, check if it is in a region and draw it

    if (in_wall_region) begin
      draw_en <= 1;
      pix <= wall_palette[wall_out];
    end

    if (in_cannon_region) begin
      if (cannon_num == 2'd0 && cannon_palette[cannon_30_out] != 12'h001) begin
        draw_en <= 1;
        pix <= cannon_palette[cannon_30_out];
      end else if (cannon_num == 2'd1 && cannon_palette[cannon_45_out] != 12'h001) begin
        draw_en <= 1;
        pix <= cannon_palette[cannon_45_out];
      end else if (cannon_num == 2'd2 && cannon_palette[cannon_60_out] != 12'h001) begin
        draw_en <= 1;
        pix <= cannon_palette[cannon_60_out];
      end
    end

    if (in_target_region && target_palette[target_out] != 12'h001) begin
      draw_en <= 1;
      pix <= target_palette[target_out];
    end

    if (in_target2_region && target_palette[target2_out] != 12'h001) begin
      draw_en <= 1;
      pix <= target_palette[target2_out];
    end

    if (in_pwr_region && (hcount < (384 + (10 * launch_pwr)))) begin  // extendy bar
      draw_en <= 1;
      pix <= pwr_palette[pwr_out];
    end

    if (in_timerframe_region && timerframe_palette[timerframe_out] != 12'h001) begin
      draw_en <= 1;
      pix <= timerframe_palette[timerframe_out];
    end

```

```
      if (in_timer_region && (hcount < (394 + game_time + 1)) ) begin // doesn't draw
    right of hcount = 394 + game_time
        draw_en <= 1;
        pix <= timer_palette[timer_out];
      end

      if (in_cannonball_region && cannonball_shot && cannonball_palette[cannonball_out] !=
     12'h001) begin
        draw_en <= 1;
        pix <= cannonball_palette[cannonball_out];
      end

      if (in_banner_region) begin
        draw_en <= 1;
        pix <= banner_palette[banner_out];
      end
    end else begin
      if (in_end_region) begin
        draw_en <= 1;
        pix <= end_palette[end_out];
      end
      if (in_banner_region) begin
        draw_en <= 1;
        pix <= banner_palette[banner_out];
      end
    end

  end

  // assign current pixel values to outputs
  assign pix_r = (draw_en) ? (pix[11:8]) : 4'b0000;  //pix[3:0];
  assign pix_g = (draw_en) ? (pix[7:4]) : 4'b0000;
  assign pix_b = (draw_en) ? (pix[3:0]) : 4'b0000;

endmodule
```

Timing module for vga control: `vga_timing.v`.

```
`timescale 1ns / 1ps

module vga_timing(
    input clk, // note: add width and height as input for dynamic adjustments
    output hsync,
    output vsync,
    output reg [10:0] hcount_reg,
    output reg [9:0] vcount_reg,
    output in_drawing_region
    );

    // hcount and vcount are initially 0
    initial begin
        hcount_reg <= 0;
        vcount_reg <= 0;
    end

    // On each clock cycle, iterate hcount and vcount correctly
    always @(posedge clk) begin
        hcount_reg <= hcount_reg + 1;
        if (hcount_reg == 11'd1903) begin
            hcount_reg <= 0;
            vcount_reg <= vcount_reg + 1;
            if (vcount_reg == 10'd931) begin
                vcount_reg <= 0;
            end
        end
    end

    // Assign hsync and vsync bools based on hcount and vcount
    assign hsync = ~((hcount_reg >= 0) && (hcount_reg <= 151));
    assign vsync = (vcount_reg >= 0) && (vcount_reg <= 2);

    // Assign drawing region bool if in the drawing region
    assign in_drawing_region = ((hcount_reg >= 384) && (hcount_reg <= 1823) && (vcount_reg
     >= 31) && (vcount_reg <= 930));
```

```

    // Not sure if needed?
    wire pixclk;
    clk_wiz_0 clk_ip (
        // Clock out ports
        .clk_out1(pixclk),     // output clk_out1
        // Clock in ports
        .clk_in1(clk) );       // input clk_in1

endmodule
```

Lab-created logic for outputting hex digits to 7-seg display: `sevenseg.v`.

```
`timescale 1ns / 1ps

module sevenseg(
    input [3:0] sw,
    output a,
    output b,
    output c,
    output d,
    output e,
    output f,
    output g
    );

    assign a = (sw[0]&!sw[1]&!sw[2]&!sw[3])|(!sw[0]&!sw[1]&sw[2]&!sw[3])|(sw[0]&sw[1]&!sw[2]&sw[3])|(sw[0]&!sw[1]&sw[2]&sw[3]);
    assign b = (sw[0]&!sw[1]&sw[2]&!sw[3])|(!sw[0]&sw[1]&sw[2]&!sw[3])|(sw[0]&sw[1]&!sw[2]&sw[3])|(!sw[0]&!sw[1]&sw[2]&sw[3])|(sw[0]&sw[1]&sw[2]&sw[3]);
    assign c = (!sw[0]&sw[1]&!sw[2]&!sw[3])|(!sw[0]&!sw[1]&sw[2]&sw[3])|(!sw[0]&sw[1]&sw[2]&sw[3])|(sw[0]&sw[1]&sw[2]&sw[3]);
    assign d = (sw[0]&!sw[1]&!sw[2]&!sw[3])|(!sw[0]&!sw[1]&sw[2]&!sw[3])|(sw[0]&sw[1]&sw[2]&!sw[3])|(sw[0]&!sw[1]&!sw[2]&sw[3])|(!sw[0]&sw[1]&!sw[2]&sw[3])|(sw[0]&sw[1]&sw[2]&sw[3]);
    assign e = (sw[0]&!sw[1]&!sw[2]&!sw[3])|(sw[0]&sw[1]&!sw[2]&!sw[3])|(!sw[0]&!sw[1]&sw[2]&!sw[3])|(sw[0]&!sw[1]&sw[2]&!sw[3])|(sw[0]&sw[1]&sw[2]&!sw[3])|(sw[0]&!sw[1]&!sw[2]&sw[3]);
    assign f = (sw[0]&!sw[1]&!sw[2]&!sw[3])|(!sw[0]&sw[1]&!sw[2]&!sw[3])|(sw[0]&sw[1]&!sw[2]&!sw[3])|(sw[0]&sw[1]&sw[2]&!sw[3])|(sw[0]&!sw[1]&sw[2]&sw[3]);
    assign g = (!sw[0]&!sw[1]&!sw[2]&!sw[3])|(sw[0]&!sw[1]&!sw[2]&!sw[3])|(sw[0]&sw[1]&sw[2]&!sw[3])|(!sw[0]&!sw[1]&sw[2]&sw[3]);


endmodule
```

Lab-created module translating between memory and 7-seg display: `seginterface.v`.

```
`timescale 1ns / 1ps

module seginterface(
        input clk, rst,
        input [3:0] dig7, dig6, dig5, dig4, dig3, dig2, dig1, dig0,
        output div_clk,
        output a, b, c, d, e, f, g,
        output [7:0] an
    );

    wire led_clk;
    reg [3:0] dig_sel;

    reg [28:0] clk_count = 11'd0;

    always @(posedge clk)
        clk_count <= clk_count + 1'b1;

    assign led_clk = clk_count[16];
    assign div_clk = clk_count[25];

    reg [7:0] led_strobe = 8'b11111110;
    always @(posedge led_clk)
        led_strobe <= {led_strobe[6:0],led_strobe[7]};
    assign an = led_strobe;

```

```
    reg [2:0] led_index = 3'd0;
    always @(posedge led_clk)
        led_index <= led_index + 1'b1;

    always@*
        case (led_index)
            3'd0: dig_sel = dig0;
            3'd1: dig_sel = dig1;
            3'd2: dig_sel = dig2;
            3'd3: dig_sel = dig3;
            3'd4: dig_sel = dig4;
            3'd5: dig_sel = dig5;
            3'd6: dig_sel = dig6;
            3'd7: dig_sel = dig7;
        endcase

    sevenseg inst_1 (.sw(dig_sel), .a(a), .b(b), .c(c), .d(d), .e(e), .f(f), .g(g));

endmodule
```

Notes and References

[1] Digilent. VGA Display Controller. https://learn.digilentinc.com/Documents/269, 2014. [Online; accessed 12/21].

[2] Xilinx. COE File Syntax. https://web.archive.org/web/20180309085520/https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/cgn_r_coe_file_syntax.htm, 2018. [Online; accessed 01/22, thorugh wayback machine].

[3] Will Green. Framebuffers. https://projectf.io/posts/framebuffers/, 2020. [Online; accessed 12/21].

[4] Will Green. Intro to FPGA graphics. https://projectf.io/posts/fpga-graphics/, 2020. [Online; accessed 12/21].

[5] Will Green. Hardware Sprites. https://projectf.io/posts/hardware-sprites/, 2020. [Online; accessed 12/21].

[6] Philip Moorby Donald Thomas. The Verilog Hardware Description Language. Springer Science Business Media, 2008.

[7] Min-Kuang Wu, Ying-Shieh Kung, Yi-Hsien Huang, and Tz-Han Jung. Fixed-point computation of robot kinematics in fpga. In *2014 International Conference on Advanced Robotics and Intelligent Systems (ARIS)*, pages 35–40, 2014.

[8] Image-Processing-Toolbox. https://github.com/Gowtham1729/Image-Processing/blob/master/README.md, 2020. [Online; accessed 01/22], .coe file description solely.