EECS 2070 02 Digital Design Labs 2020
Lab 7

學號：108062281　　姓名：莊晴雯

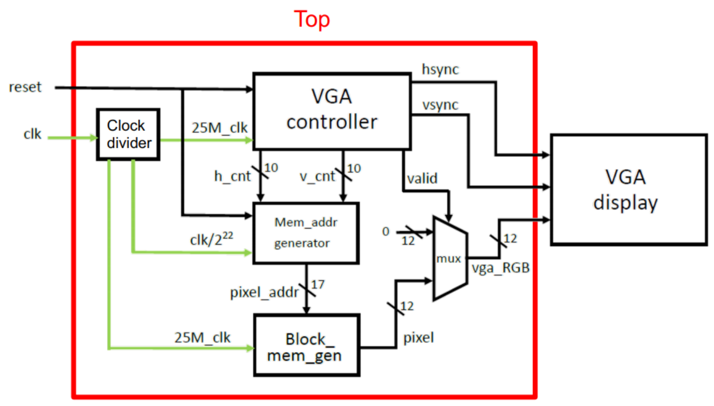
1. 實作過程

In this lab07, we need to implement a digital photo frame with fancy transition effects on the VGA display. Regarding this lab, we need to generate our own *.coe file using the provided PicTrans.exe. PicTrans.exe will convert a *.jpg file to a bit map file, and all we need to do is to first have an image of *.jpg, run PicTrans.exe and enter the width and the height of the output file, also the name of the image file, and then a *.coe file will be generated under the name out.coe. out.coe will then be used when we create the memory to store the image data using memory IP. We will choose the out.coe file when we load the init file in the memory initialization section, as seen on page 39 of lec08 fpga4.

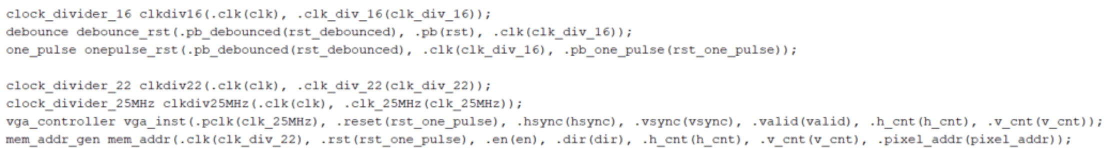
lab7_1

In this lab7_1, we need to design a VGA controller to make our image scroll either upwards or downwards, with the inputs clk, rst, en, dir, and the outputs vgaRed, vgaGreen, vgaBlue, hsync, and vsync. The rst is triggered by the positive edge, and when rst occurs, the VGA display will show the image at the original position. We then have en to determine whether the image will move or not. When en == 1’b0, the image will stay still at its position and when en == 1’b1, we need to see dir. When en == 1’b1 and dir == 1’b0, the image will scroll up at the frequency of 100MHz divided by 2**22, and when en == 1’b1 and dir == 1’b1, the image will scroll down at the frequency of 100MHz divided by 2**22.

I designed lab7_1 using the block diagram provided in the PPT:



The Top module, which is module lab7_1, instantiates several other modules such as clock divider, VGA controller, mem_addr_generator, and block_mem_gen.



```
clock_divider_16 clkdiv16(.clk(clk), .clk_div_16(clk_div_16));
debounce debounce_rst(.pb_debounced(rst_debounced), .pb(rst), .clk(clk_div_16));
one_pulse onepulse_rst(.pb_debounced(rst_debounced), .clk(clk_div_16), .pb_one_pulse(rst_one_pulse));

clock_divider_22 clkdiv22(.clk(clk), .clk_div_22(clk_div_22));
clock_divider_25MHz clkdiv25MHz(.clk(clk), .clk_25MHz(clk_25MHz));
vga_controller vga_inst(.pclk(clk_25MHz), .reset(rst_one_pulse), .hsync(hsync), .vsync(vsync), .valid(valid), .h_cnt(h_cnt), .v_cnt(v_cnt));
mem_addr_gen mem_addr(.clk(clk_div_22), .rst(rst_one_pulse), .en(en), .dir(dir), .h_cnt(h_cnt), .v_cnt(v_cnt), .pixel_addr(pixel_addr));
```

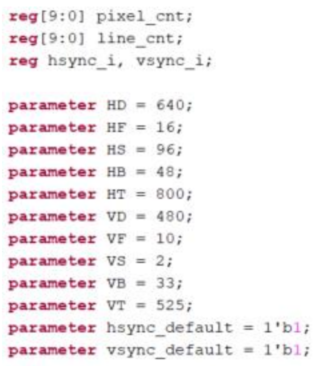
I also have several additional wires, namely:

- wire:
  - data: takes 12 bits, takes the value of {vgaRed, vgaGreen, vgaBlue} and then will be used as the input of blk_mem_gen module.
  - clk_div_22: frequency of clk/(2**22).
  - clk_25MHz: frequency of 25MHz. We used 25MHz because for 640 x 480 pixels with 60Hx refresh rate, we can get the clock frequency from 800 x 525 x 60 (frame/sec) = 25M (pixel/sec).
  - pixel_addr: takes 17 bits, generated from mem_addr_gen module and will be used as the input for blk_mem_gen module.
  - pixel: takes 12 bits, its value will be assigned to {vgaRed, vgaGreen, vgaBlue} if valid is equal to 1'b1, and it is the output of blk_mem_gen module.
  - valid: is the output of vga_controller module, and will be used to determine the value assigned to {vgaRed, vgaGreen, vgaBlue}.
  - h_cnt: takes 10 bits, is the output of vga_controller module and the input of mem_addr_gen module.
  - v_cnt: takes 10 bits, is the output of vga_controller module and the input of mem_addr_gen module.
  - rst_debounced: for the rst button.
  - rst_one_pulse: for the rst button.

For the clock divider, we need to generate the frequencies of clk/(2**16) for the pushbutton (rst), clk/(2**22) for the animation of the image, and frequency of 25MHz for the input of VGA controller module. As per usual, to make the pushbuttons work, we need to instantiate debounce and one pulse module that will generate rst_one_pulse in the end to be used.

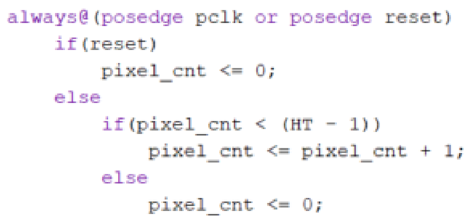
The vga_controller module takes clk_25MHz and rst_one_pulse as the inputs and generate hsync, vsync, valid, 10 bits of h_cnt and 10 bits of v_cnt as the outputs.

We have additional registers provided which are 10 bits of pixel_cnt and 10 bits of line_cnt, also hsync_i and vsync_i. We also have parameters provided along with their values.



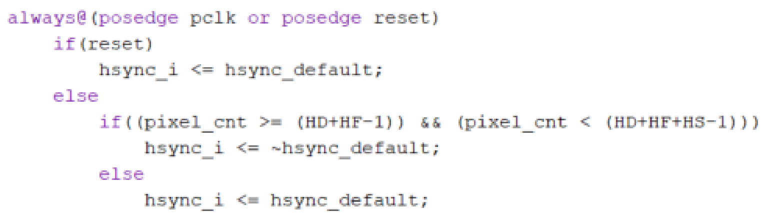
```
reg[9:0] pixel_cnt;
reg[9:0] line_cnt;
reg hsync_i, vsync_i;

parameter HD = 640;
parameter HF = 16;
parameter HS = 96;
parameter HB = 48;
parameter HT = 800;
parameter VD = 480;
parameter VF = 10;
parameter VS = 2;
parameter VB = 33;
parameter VT = 525;
parameter hsync_default = 1'b1;
parameter vsync_default = 1'b1;
```

Furthermore, we also have four always block inside the vga_controller module, and each of them deals with pixel_cnt, hsync_i, line_cnt, and vsync_i respectively. On the first always block, when reset occurs, pixel_cnt will be set to 0, otherwise, we see if pixel_cnt < (HT-1), then pixel_cnt will increase its value by one, otherwise it will become 0.



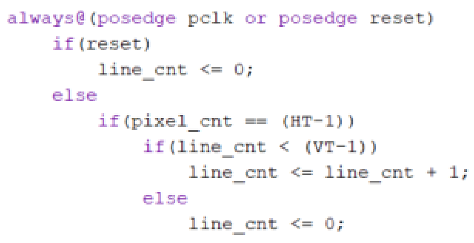
```
always@(posedge pclk or posedge reset)
    if(reset)
        pixel_cnt <= 0;
    else
        if(pixel_cnt < (HT - 1))
            pixel_cnt <= pixel_cnt + 1;
        else
            pixel_cnt <= 0;
```

On the second always block which deals with hsync_i, when reset occurs then hsync_i will be set to hsync_default's value. Otherwise, we see if ((pixel_cnt >= (HD+HF-1)) && (pixel_cnt < (HD+HF+HS-1))), then hsync_i will be assigned to the value of ~hsync_default, otherwise hsync_i will be assigned to the value of hsync_default.



```
always@(posedge pclk or posedge reset)
    if(reset)
        hsync_i <= hsync_default;
    else
        if((pixel_cnt >= (HD+HF-1)) && (pixel_cnt < (HD+HF+HS-1)))
            hsync_i <= ~hsync_default;
        else
            hsync_i <= hsync_default;
```

On the third always block which deals with line_cnt, when reset occurs then line_cnt will be set to 0. Otherwise, if pixel_cnt is equal to (HT-1), then we see if line_cnt < (VT-1), then line_cnt will increase its value by 1, else line_cnt will be set as 0.



```
always@(posedge pclk or posedge reset)
    if(reset)
        line_cnt <= 0;
    else
        if(pixel_cnt == (HT-1))
            if(line_cnt < (VT-1))
                line_cnt <= line_cnt + 1;
            else
                line_cnt <= 0;
```

On the last always block, which deals with vsync_i, when reset occurs then vsync_i will be set to vsync_default's value. Else, if((line_cnt >= (VD+VF-1)) && (line_cnt < (VD+VF+VS-1))), then vsync_i will take the value of ~vsync_default. Else, then vsync_i will take the value of vsync_default.

```
always@(posedge pclk or posedge reset)
    if(reset)
        vsync_i <= vsync_default;
    else
        if((line_cnt >= (VD+VF-1)) && (line_cnt < (VD+VF+VS-1)))
            vsync_i <= ~vsync_default;
        else
            vsync_i <= vsync_default;
```

In the end, hsync will then take the value of hsync_i, vsync will take the value of vsync_i, valid will take the value of ((pixel_cnt < HD) && (line_cnt < VD)), h_cnt will take the value of pixel_cnt if pixel_cnt < HD, else it will take the value of 10'd0, and v_cnt will take the value of line_cnt if line_cnt < VD, otherwise it will be assigned to 10'd0.

```
assign hsync = hsync_i;
assign vsync = vsync_i;
assign valid = ((pixel_cnt < HD) && (line_cnt < VD));
assign h_cnt = (pixel_cnt < HD) ? pixel_cnt : 10'd0;
assign v_cnt = (line_cnt < VD) ? line_cnt : 10'd0;
```

Next, we take a look at the mem_addr_gen module. This is where I modifies the code in order to make the picture slide up and down with the frequency of clk/(2**22). We have inputs clk, rst, en, dir, 10 bits of h_cnt and 10 bits of v_cnt, and lastly the output of 17 bits pixel_addr. We also have an additional 8 bits register of position. We first need to assign pixel_addr to the value of ((h_cnt >> 1)+320*(v_cnt >> 1)+position*320)%76800 in order to make the picture move up and down.

```
assign pixel_addr = ((h_cnt >> 1)+320*(v_cnt >> 1)+position*320)%76800;
```

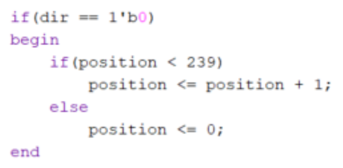
We then have an always block that will handle the position. When rst occurs, position will be set to 0 and the picture displayed on the screen will be set back to its original position.

```
if(rst)
    position <= 0;
```

Else, we first need to check the value of en. If en == 1'b0, then the picture will stay at its position and not move, which is why I assign the value of position as position itself.

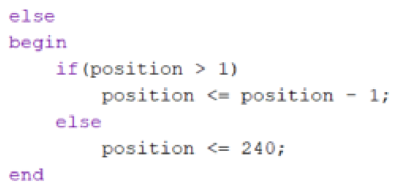
```
if(en == 1'b0)
    position <= position;
```

Otherwise, if the value of en is 1’b1, we need to check the value of dir. If dir == 1’b0, then the picture displayed should slide upwards, and we have the boundaries 239 and 0. So until the value of position has not reached 239, we increase the value of position by 1 at frequency of clk/(2**22). When it reaches 239, the value of position will be set to 0 and the process repeats.



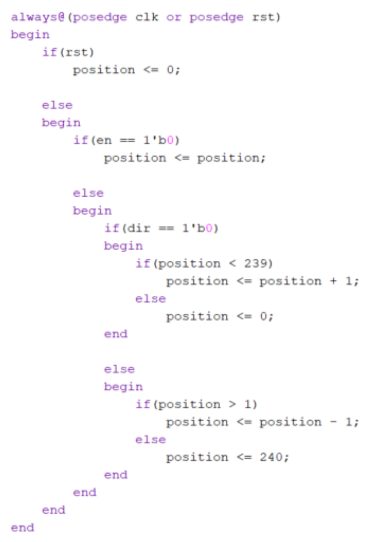
```
if(dir == 1'b0)
begin
    if(position < 239)
        position <= position + 1;
    else
        position <= 0;
end
```

Otherwise, when dir == 1’b1, which means the image will scroll down, we have the boundaries of 1 and 240. When position is till greater than 1, position will decrease its value by 1 at frequency of clk/(2**22). When it reaches 1, the value of position will be set back to 240 and the process repeats.



```
else
begin
    if(position > 1)
        position <= position - 1;
    else
        position <= 240;
end
```

Put together, the whole code regarding the position will be like this:

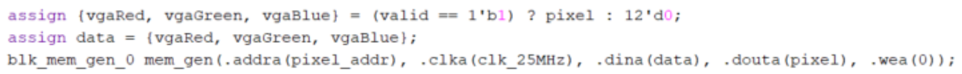


```
always@(posedge clk or posedge rst)
begin
    if(rst)
        position <= 0;

    else
    begin
        if(en == 1'b0)
            position <= position;

        else
        begin
            if(dir == 1'b0)
            begin
                if(position < 239)
                    position <= position + 1;
                else
                    position <= 0;
            end

            else
            begin
                if(position > 1)
                    position <= position - 1;
                else
                    position <= 240;
            end
        end
    end
end
```

I used clk/(2**2) frequency in this always block.

We then need to assign the value of {vgaRed, vgaGreen, vgaBlue} as the value of pixel if valid == 1’b1, otherwise its value will be set to 12’d0. The value of {vgaRed, vgaGreen, vgaBlue} will then be assigned to data, which will be the input for blk_mem_gen module in the dina input port.

After going through the mem_addr_gen module which generates the output pixel_addr of 17 bits, we can assign the value of pixel_addr as the input of the blk_mem_gen module in the addra input port. Other inputs of blk_mem_gen module includes clk_25MHz which will be assigned to the input port clka, and 0 to the input port wea. blk_mem_gen will then generate the output pixel.



```
assign {vgaRed, vgaGreen, vgaBlue} = (valid == 1'b1) ? pixel : 12'd0;
assign data = {vgaRed, vgaGreen, vgaBlue};
blk_mem_gen_0 mem_gen(.addra(pixel_addr), .clka(clk_25MHz), .dina(data), .douta(pixel), .wea(0));
```

lab7_2

In lab7_2, we need to design a VGA controller that can split an image into two parts (left and right) when pressing the button split, and those two parts will slide towards their own direction (left part slides towards the left direction, and right part slides towards the right direction, both exiting the screen while sliding). Then after the two parts disappear entirely, the top half and the bottom half will enter the screen simultaneously and merge back into one complete image. After merging into a complete image, the image will split again and will continue the above process until the user presses the rst button. We are provided with the inputs clk, rst, and split, and the outputs vgaRed, vgaGreen, vgaBlue which were 4 bits each, hsync, and vsync.

To deisgn this lab7_2, I had the thought of making three states each for INITIAL state, SPLIT state, and MERGE state, which I defined at the top of the file.

1. INITIAL: The original image after rst or after first programmed. Basically just a complete image.
2. SPLIT: When the image splits into two parts (left and right) and exits the screen simultaneously until it disappears completely.
3. MERGE: When the image appears from the top and bottom of the screen, simultaneously sliding upwards and downwards until it merged into a complete full image.

I have several additional wires and registers, namely:

- wires:
    - data: takes 12 bits, takes the value of {vgaRed, vgaGreen, vgaBlue} and then will be used as the input of blk_mem_gen module.
    - pixel: takes 12 bits, its value will be assigned to {vgaRed, vgaGreen, vgaBlue}.
    - clk_div_22: frequency of clk/(2**22).

- clk_div_16: frequency of clk/(2**16).
- clk_25MHz: frequency of 25MHz. We used 25MHz because for 640 x 480 pixels with 60Hx refresh rate, we can get the clock frequency from 800 x 525 x 60 (frame/sec) = 25M (pixel/sec).
- valid: is the output of vga_controller module, and will be used to determine the value assigned to {vgaRed, vgaGreen, vgaBlue}.
- h_cnt: takes 10 bits, is the output of vga_controller module.
- v_cnt: takes 10 bits, is the output of vga_controller module.
- rst_debounced: for the rst button.
- rst_one_pulse: for the rst button.
- split_debounced: for the split button.
- split_one_pulse: for the split button.

- registers:
    - pixel_addr: takes 17 bits, generated from mem_addr_gen module and will be used as the input for blk_mem_gen module.
    - position: takes 8 bits, will be used for the SPLIT state (split and slide towards left and right direction).
    - position_v: takes 8 bits, will be used for the MERGE state, where the picture will slide up and down simultaneously. Initial value is 120 because the picture does not begin from the middle but begin to slide from outside the screen instead.
    - position_v1: takes 8 bits, will be used for the MERGE state, where the picture will slide up and down simultaneously. Initial value is 120 because the picture does not begin from the middle but begin to slide from outside the screen instead.
    - state: takes 2 bits, represents the state of the VGA controller. Initial value is `INITIAL.
    - border_right: takes 8 bits, used to determine when the split image has reached the end of the screen, indicating that the image has disappeared completely, and that the VGA controller can go from SPLIT state to MERGE state.
    - border_up: takes 8 bits, used to determine when the image has merged completely, forming a full image, indicating that the VGA controller can go from MERGE state to SPLIT state.
    - pixel_addr: takes 17 bits, used for the transition of the image.
    - start_split: flag to indicate that the split button was pressed.

We first need to instantiate the clock divider modules each for different purposes. We use clk/(2**16) for the split and rst buttons, clk/(2**22) for the movement of the image, and clk_25MHz for the input of the vga_controller module.

```
clock_divider_16 clkdiv16_1(.clk(clk), .clk_div_16(clk_div_16));
clock_divider_22 clkdiv22_1(.clk(clk), .clk_div_22(clk_div_22));
clock_divider_25MHz clkdiv25MHz_1(.clk(clk), .clk_25MHz(clk_25MHz));
```

Then, we need to instantiate debounce and one pulse modules for the split and rst buttons.

```
debounce debounce_rst_1(.pb_debounced(rst_debounced), .pb(rst), .clk(clk_div_16));
one_pulse onepulse_rst_1(.pb_debounced(rst_debounced), .clk(clk_div_16), .pb_one_pulse(rst_one_pulse));
debounce debounce_split(.pb_debounced(split_debounced), .pb(split), .clk(clk_div_16));
one_pulse onepulse_split(.pb_debounced(split_debounced), .clk(clk_div_16), .pb_one_pulse(split_one_pulse));
```

Then we instantiate the vga_controller module. The explanation regarding the vga_controller module was explained previously on page 2-4 of this report file.

```
vga_controller_1 vga_inst_1(.pclk(clk_25MHz), .reset(rst_one_pulse), .hsync(hsync), .vsync(vsync), .valid(valid), .h_cnt(h_cnt), .v_cnt(v_cnt));
```

Next, I have an always block that deals with start_split. In the always@(posedge clk_div_16 or posedge rst_one_pulse) block, if rst_one_pulse == 1’b1, which means that the rst button was pressed, then start_split will be assigned to the value 1’b0. Else, if split_one_pulse == 1’b1, which means that the split button was pressed, then start_split will negate its value itself. Otherwise, when split_one_pulse == 1’b0, start_split will still hold the same value as its previous value.

```
always@(posedge clk_div_16 or posedge rst_one_pulse)
begin
    if(rst_one_pulse == 1'b1)
        start_split <= 1'b0;
    else
    begin
        if(split_one_pulse == 1'b1)
            start_split <= ~start_split;
        else
            start_split <= start_split;
    end
end
```

Moving on, we can take a look at the always block that is used to handle the positions and borders. First if statement is when rst_one_pulse == 1’b1. Everything will be set back to its initial values. position will be set to 0, position_v and position_v1 will be set to 120, both border_right and border_up will be set to 0, and state will be set to `INITIAL.

```
if(rst_one_pulse == 1'b1)
begin
    position <= 0;
    position_v <= 120;
    position_v1 <= 120;
    border_right <= 0;
    border_up <= 0;
    state <= `INITIAL;
end
```

Else, if rst_one_pulse is not 1’b1, then inside the else statement there will be another if – else if – else if statement to handle the states.

The first if is to handle the INITIAL state. In the INITIAL state, position, position_v, position_v1, border_right, and border_up will be set to their own initial values. In the INITIAL state, user can press the split button. Hence, if start_split == 1’b1, state will be assigned to `SPLIT, making the VGA controller go from INITIAL state to SPLIT state. If the button was not pressed, then the state remains as `INITIAL.

```
if(state == `INITIAL)
begin
    position <= 0;
    position_v <= 120;
    position_v1 <= 120;
    border_right <= 0;
    border_up <= 0;

    if(start_split == 1'b1)
        state <= `SPLIT;
    else
        state <= `INITIAL;
end
```

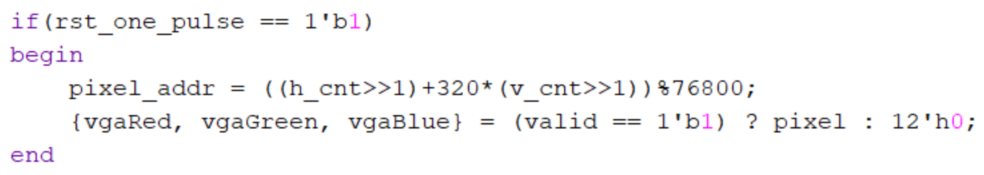
The next else if statement is used to handle the state SPLIT. Here, position_v and position_v1 can be set to 120 since these two are going to be used in the MERGE state. Hence, we can set their values back to their initial values at the SPLIT state. The same thing goes for border_up, which is set back to 0. In this state, while position’s value is still smaller than 320, position will keep increasing its value by one until it reaches 320, the borderline of the left and right of the screen. Thus, by increasing position’s value by one, we ca make the image move towards the left and right direction simultaneously. When position’s value reaches 320, its value will remain the same. When position increases its value by one, border_right also increases its value by one the same time as position. Until border_right reaches 160, it indicates that the image has disappeared completely from the screen and the VGA controller will go to the MERGE state. The number 160 came from 320/2, since the image was divided by half too when moving towards the left and the right direction. When border_right has not reached 160, the controller will stay at SPLIT state and border_right will keep increasing by one.

```
else if(state == `SPLIT)
begin
    position_v <= 120;
    position_v1 <= 120;
    border_up <= 0;

    if(start_split == 1'b1)
    begin
        if(border_right == 160)
            state <= `MERGE;

        else
        begin
            border_right <= border_right + 1;
            state <= `SPLIT;
        end

        if(position < 320)
            position <= position + 1;
        else
            position <= position;
    end
```

Next up is the else if for MERGE state. In the MERGE state, position and border_right will be reset to 0 since both of them will be reused in SPLIT state. Here, we have position_v and position_v1 which deals with the image scrolling upwards and downwards. The boundaries for position_v1 is 120 and 239. While position_v1 has not reached 239, position_v1 will keep increasing its value. Otherwise, position_v1 will be set back to 120. For position_v, the boundaries are 1 and 120. position_v will decrease its value by one until it reaches 1, and then its value will be set back to 120. border_up will also start counting the same time and at the same pace as position_v and position_v1. Until border_up reaches 120, which is the end of the screen, the VGA controller will go to the SPLIT state again from the MERGE state. Otherwise, it will stay at the MERGE state and border_up will still be increasing by one. I set it to 120 since when merging, we take the top half of the picture and the bottom half of the picture and let them meet in the middle. Since it is two picture being halved, then it is 240/2 = 120, which is also the center of the screen vertically.

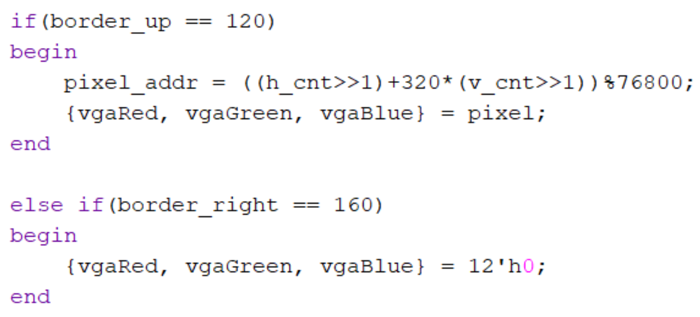
```
else if(state == `MERGE)
begin
    position <= 0;
    border_right <= 0;

    if(start_split == 1'b1)
    begin
        if(border_up == 120)
            state <= `SPLIT;

        else
        begin
            state <= `MERGE;
            border_up <= border_up + 1;
        end

        if(position_v1 < 239)
            position_v1 <= position_v1 + 1;
        else
            position_v1 <= 120;

        if(position_v > 1)
            position_v <= position_v - 1;
        else
            position_v <= 120;
    end
```

The next always block deals with displaying the image on the screen. It used posedge clk or posedge rst_one_pulse. Inside the always block, the first thing we can see is the if statement if(rst_one_pulse == 1'b1), that is, when the rst button was pressed, then the original image was shown on screen, and we know that to show the original image on the screen, we need to assign pixel_addr as ((h_cnt>>1)+320*(v_cnt>>1))%76800 and {vgaRed, vgaGreen, vgaBlue} ad pixel if valid is equal to 1'b1. Otherwise, the value of {vgaRed, vgaGreen, vgaBlue} will be 12'h0 which is blank.



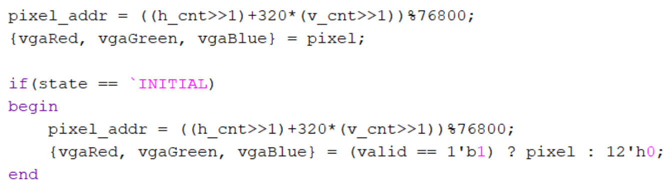
```
if(rst_one_pulse == 1'b1)
begin
    pixel_addr = ((h_cnt>>1)+320*(v_cnt>>1))%76800;
    {vgaRed, vgaGreen, vgaBlue} = (valid == 1'b1) ? pixel : 12'h0;
end
```

Next, if rst_one_pulse == 1'b0 and valid == 1'b1, then we see if border_up is equal to 120, then we need to assign pixel_addr to ((h_cnt>>1)+320*(v_cnt>>1))%76800 and {vgaRed, vgaGreen, vgaBlue} to pixel. Also when border_right, I directly assign {vgaRed, vgaGreen, vgaBlue} to 12'h0 which is blank. I did this because previously there was a slight blink during transition from SPLIT to MERGE state and vice versa.



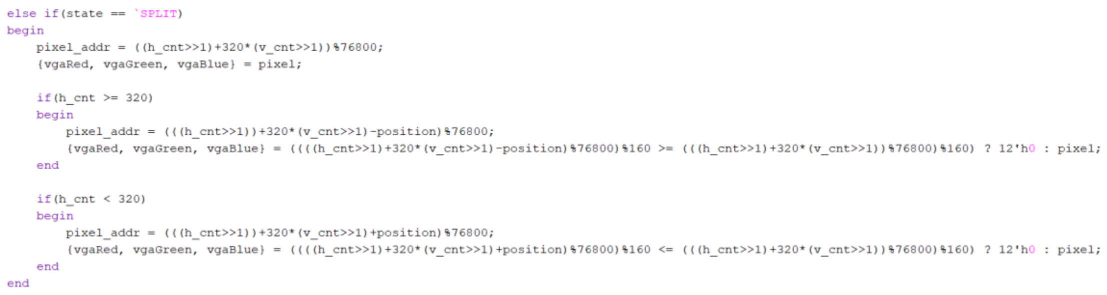
```
if(border_up == 120)
begin
    pixel_addr = ((h_cnt>>1)+320*(v_cnt>>1))%76800;
    {vgaRed, vgaGreen, vgaBlue} = pixel;
end

else if(border_right == 160)
begin
    {vgaRed, vgaGreen, vgaBlue} = 12'h0;
end
```

Else, I set pixel_addr to ((h_cnt>>1)+320*(v_cnt>>1))%76800 and {vgaRed, vgaGreen, vgaBlue} to pixel to avoid a slight blink after pressing the split button after reset, which actually failed (will be explained in number 2 of this report). Then, we can start working on the display based on the states. If the state is INITIAL, I can directly assign pixel_addr to ((h_cnt>>1)+320*(v_cnt>>1))%76800 and {vgaRed, vgaGreen, vgaBlue} to pixel if valid == 1'b1, else it will be set to 12'h0. I did this to show the original image at the original position since it is the INITIAL state.



```
pixel_addr = ((h_cnt>>1)+320*(v_cnt>>1))%76800;
{vgaRed, vgaGreen, vgaBlue} = pixel;

if(state == `INITIAL)
begin
    pixel_addr = ((h_cnt>>1)+320*(v_cnt>>1))%76800;
    {vgaRed, vgaGreen, vgaBlue} = (valid == 1'b1) ? pixel : 12'h0;
end
```

If the VGA controller is at SPLIT state, I also assign pixel_addr to ((h_cnt>>1)+320*(v_cnt>>1))%76800 and {vgaRed, vgaGreen, vgaBlue} to pixel to avoid a slight blink after pressing the split button after reset, (which failed, refer to the same statement I mentioned in the previous page). Apart from this, we need to see h_cnt and its boundary which is 320. We deal with two cases, if h_cnt >= 320 and if h_cnt < 320 to determine the half left of the image to slide left and the half right of the image to slide right. To make the image slide left or right, we just need to set the formula of pixel_addr to either –position or +position. We also need to modify the condition for the value that will be assigned to {vgaRed, vgaGreen, vgaBlue}. For h_cnt >= 320, we used –position in ((((h_cnt>>1)+320*(v_cnt>>1)-position)%76800)%160 >= (((h_cnt>>1)+320*(v_cnt>>1))%76800)%160) and for h_cnt < 320 we need to use +position in ((((h_cnt>>1)+320*(v_cnt>>1)+position)%76800)%160 <= (((h_cnt>>1)+320*(v_cnt>>1))%76800)%160). Notice that I also used %160. The 160 here came from 320/2 since we only take half of the screen for each of the split image. This can be seen on the screenshot below.



```
else if(state == `SPLIT)
begin
    pixel_addr = ((h_cnt>>1)+320*(v_cnt>>1))%76800;
    {vgaRed, vgaGreen, vgaBlue} = pixel;

    if(h_cnt >= 320)
    begin
        pixel_addr = (((h_cnt>>1))+320*(v_cnt>>1)-position)%76800;
        {vgaRed, vgaGreen, vgaBlue} = ((((h_cnt>>1)+320*(v_cnt>>1)-position)%76800)%160 >= (((h_cnt>>1)+320*(v_cnt>>1))%76800)%160) ? 12'h0 : pixel;
    end

    if(h_cnt < 320)
    begin
        pixel_addr = (((h_cnt>>1))+320*(v_cnt>>1)+position)%76800;
        {vgaRed, vgaGreen, vgaBlue} = ((((h_cnt>>1)+320*(v_cnt>>1)+position)%76800)%160 <= (((h_cnt>>1)+320*(v_cnt>>1))%76800)%160) ? 12'h0 : pixel;
    end
end
```

For the MERGE state, we deal with the image scrolling up and down. This time, we use v_cnt and 240 as the boundary. We deal with two cases, if v_cnt >= 240 and if v_cnt < 240 to determine the half top of the image to slide downwards and the half bottom of the image to slide upwards. For v_cnt >= 240, I used position_v1 to deal with the formula. Here, pixel_addr will be assigned to (((h_cnt>>1))+320*(v_cnt>>1)+320*position_v1)%76800, and {vgaRed, vgaGreen, vgaBlue} will consider the condition ((((h_cnt>>1)+320*(v_cnt>>1)-(position_v1-120))%76800) >= (((h_cnt>>1)+320*(v_cnt>>1)+320*(position_v1-120))%76800)) for {vgaRed, vgaGreen, vgaBlue} to take the value of pixel. Otherwise it will take the value 12'h0 which is blank. As for v_cnt < 240, I assign pixel_addr as (((h_cnt>>1))+320*(v_cnt>>1)+320*position_v)%76800 and {vgaRed, vgaGreen, vgaBlue} will consider the condition ((((h_cnt>>1)+320*(v_cnt>>1)+(position_v+120))%76800) <= (((h_cnt>>1)+320*(v_cnt>>1)+320*(position_v+120))%76800)) for {vgaRed, vgaGreen, vgaBlue} to take the value of pixel. Otherwise, it will take the value 12’h0 which is blank. I used position_v1-120 and position_v+120 because 120 is the screen being halved, from 240 to 120 since we only used the top half for the picture to slide downwards and bottom half for the picture to slide up, before merging into one whole image.

Next up, we just need to assign the value of data to {vgaRed, vgaGreen, vgaBlue}, and then instantiate blk_mem_gen module and pass the inputs to the input ports, and let it generate the output from the inputs.

2. 學到的東西與遇到的困難

   I was pretty much confused on what to do, most particularly at lab7_2. I had a hard time figuring out what to do in order to make the picture slide to left and right. I tried countless possible calculations before I ended up with the simple –position in (((h_cnt>>1))+320*(v_cnt>>1)-position)%76800 and +position in (((h_cnt>>1))+320*(v_cnt>>1)+position)%76800. I then also got confused on when to merge and when to split before I ended up using border_up and border_right that counts alongside with the positions simultaneously, to indicate when the borders has reached the end of the screen. Therefore, I have decided to use states to determine the transition between SPLIT and MERGE. I also encountered one problem which was pointed out during the demo day, in which my image blinked for a split second after pressing the split button, and before the picture splits into two parts. I have not had the chance to fix it during the demo day, since I figured that the blink was caused by the delay since I used states. So I concluded that in order to fix the blink, I need to re-do my design all over again and think of another way instead of using states.

3. 想對老師或助教說的話

   I feel like this lab has been one of the most challenging labs for me among all other labs. Figuring out what to do and figuring out the calculations and the borders and boundaries has been quite a stressful journey for me while doing this lab. I do hope that more explanations will be given regarding this lab since I am very new to this and have a very limited knowledge on this section.