Picture in Picture

Edward Semper
Vincent Wu
Sue Zhengs

This project is to create a design for a picture in picture system. The system is design to take in two NTSC video sources and display both of them on a VGA screen at the same time. One of the video sources would be displayed in a small window in the foreground while the other would be full size in the background. The choice of which video image is in the foreground and background can be switched by the user. The size and angle of the smaller image can also be adjusted by user input. This design also features and integration mode where an object in the foreground can be integrated into the background image.

This design is divided into three stages. Stage 1 handles the digitizing the signals and storing those signals to RAM as well as managing the access to the bus. Stage 2 handles the manipulation of the foreground image. Stage 3 handles the final output to the screen which involves combining the two images by position the foreground image onto the background image.

The two video images are converted from analog to digital using a pair of AD775 A/D converter and GS4981 Sync Separator. Depending on user input, one signal is routed to foreground RAM and one signal is routed to the background RAM. Stage 2 has an intermediate RAM that is uses to calculate intermediate steps in it transformation and a ROM to provide the mathematical constants that it needs for it transform calculation. The output final output from Stage 2 is stored on fg_out RAM. Stage3 is constantly accessing background RAM and fg RAM to determine which pixel it should show. User input indicates it moving up and down.

Access across to the databus needs to be managed across the three kits. Based on a 20 MHz clock, Stage 3 needs to output a pixel, every 2 clock cycles. Saving two pixels at time frees up a two clock cycle time period where the other two stages can access the bus. Stage 1 and 2 alternates between bus access times (See Figure 1).

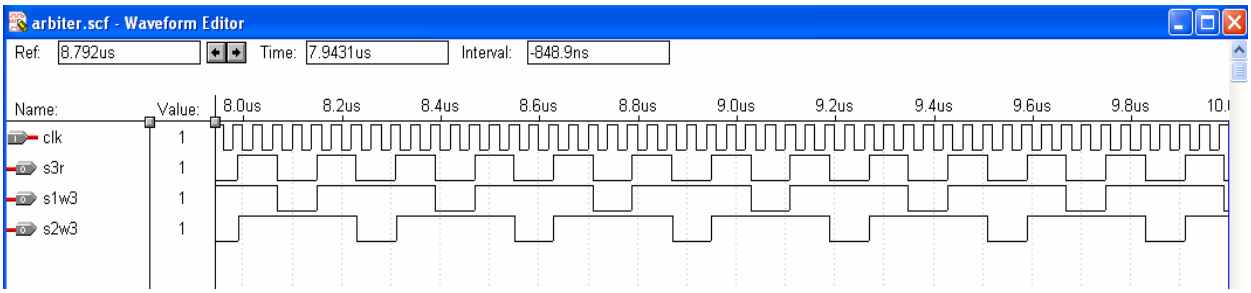
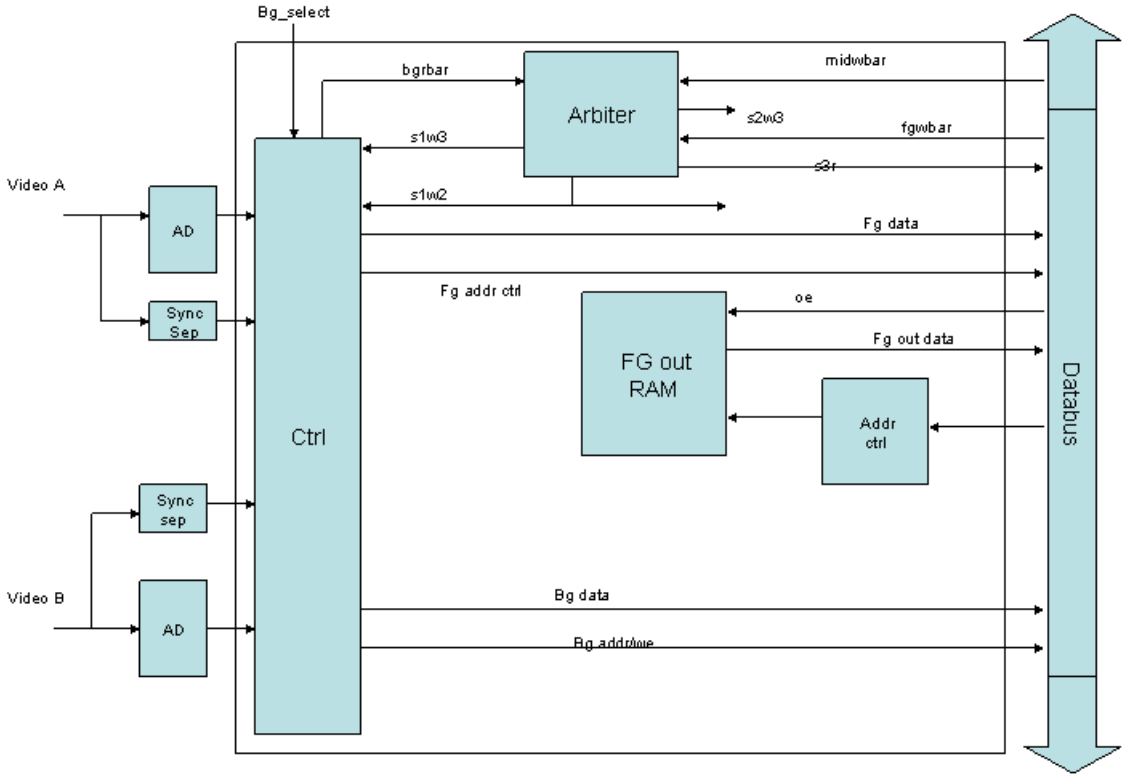


Figure 1: s3r is high during the times when Stage 3 has access to the data bus. s1w3 and s2w3 are low indicating when stage 2 and stage 3 have access to the data bus.

Stage 1



Arbiter

The purpose of the Arbiter is to manage access to the bus, upon which are control signals between the different stages and address and databits from the four RAMs and other control signal. The arbiter takes in three inputs and outputs four control signals. The three

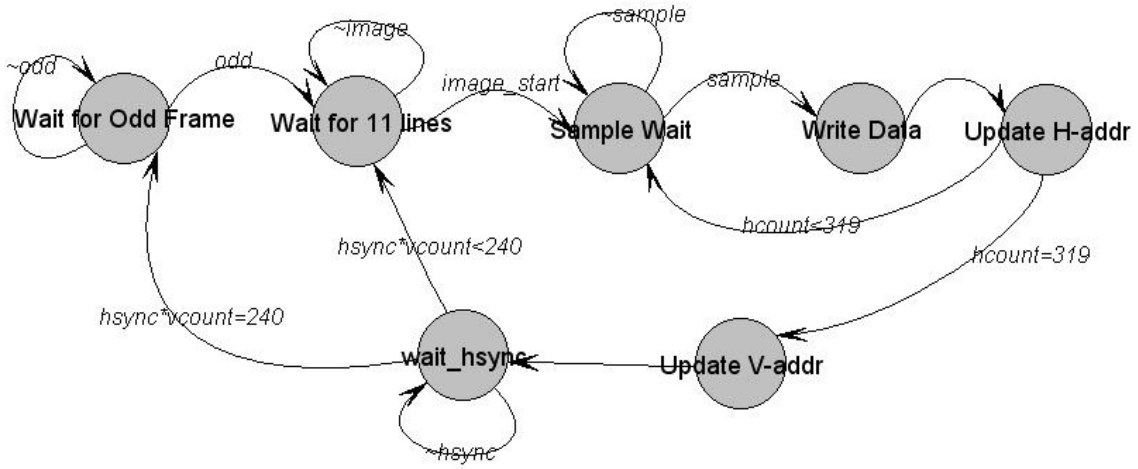
inputs are bgrbar, which indicates that when Stage 3 cannot start reading from the bg RAM, midwbar which indicates when Stage 1 can write the foreground information to Stage 2, and fgwbar, which indicates when Stage 2 cannot start writing to the fg_out RAM. The outputs from the arbiter are timed to meet the timing convention already established for the bus. The four signals are s1w2 (Stage 1 can write to fg RAM), s1w3, (Stage 1 can write to bg RAM), s2w3 (Stage 2 can write fg out RAM) and s3r (Stage 3 can read from the RAMs).

Because of limitations in the design, the inputs to the arbitrator are actually put on the bus. The arbiter only listens to those inputs when it is a valid time that Stage to have access to the bus. During the times when the stage doesn't have access to the bus, the arbiter just holds the particular value from the last time that stage had access to the bus.

Front

The front module combines the digitizing and storing to RAM of the two video sources. It includes a storebg, storefg and sample clk module. There is a three sample clock cycle delay for the output from the AD 775 A/D converter so the appropriately delays the address from its internal so that they are sync correctly. In order to identify line and field information, a GS 4981 Sync separater was used to decode the sync signals. The front module processes these signals to identify odd and even fields. The front module also handles the user input that selects which video source should be used as background and which as foreground.

storebg



The storebg uses seven states to store video to the background RAM. It waits in the idle state until it gets a signal indicating the start of a field. It then goes to the wait-image state where it wait for the off the screen lines to pass. After the wait image state, it then moves to the wait sample state. In the wait sample state, storebg waits for a signal to tell it that the store a sample is available at which point it moves to the write sample state where it passes the data from the AD to one of two pixel registers. Storebg sends a signal indicating when it is writing on the first line of RAM to the Arbiter so it can in sure appropriate spacing between the addressing on the RAM.

storefg

The storefg also uses seven states to store video to the sample RAM. The process is very similar to the storebg FSM except that it stores an entire frame and then signals when it is done. Also before transitioning from the wait image state to the wait sample, it checks for write permission from the Arbiter which indicates Stage 2 is waiting data. If Stage 2 isn't expecting data, it goes back to waiting for the beginning of the next frame.

sampleclk

The purpose of sample clock is to count the appropriate sample interval. It signals a sample interval every four clock cycles.

Sync

The purpose of this module is to sync the asynchronous inputs. The horizontal and vertical sync pulses as well as the reset button need to be sync to the clock.

Stage 2

Overview of Image Processing

Written by Vincent Wu

The second stage of the Picture-in-Picture project involves performing scaling and rotation transforms on the foreground video feed. Conventional digital image processing involves multiplying an image represented as an array of points by a transformation matrix. However, for real-time digital video processing, this method of image processing is too costly for a real-time application.

To transform our foreground video feed, the Stage 2 manipulates the stored image from Stage 1 with two passes, a vertical transform and a horizontal transform. The vertical transform performs operations on columns of the image, while the horizontal transform performs operations on rows of the image. The image can be processed as two one-dimensional transforms because of the separability of the transformation matrix referred earlier. This separability property applied to image processing was first proposed by Catmull and Smith [1]. Stage 2 applies a similar two-pass approach proposed by Chen and Kaufman[2].

All two-pass rotations involve a vertical and horizontal skew transformation. A skew involves scaling the images and linearly translating the image columns or rows .Chen and Kaufman developed the following two-pass rotation transformation formula:

---

[1] E. Catmull and A.R. Smith 3-D transformation of images in scanline order. *Computer Gaphics (SIGGRAPH '80 Proceedings)*, 14(3): 279-285, July 1980.

[2] B. Chen and A. Kaufman. Two-Pass Image and Volume Rotation. http://www-users.cs.umn.edu/~baoquan/papers/rot2p.pdf

$$R(\alpha) = \begin{bmatrix} 1 & 0 \\ \tan\alpha & \sec\alpha \end{bmatrix} \begin{bmatrix} \cos\alpha & -\sin\alpha \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{read} \\ y_{read} \end{bmatrix} (1)$$

R(alpha) = vertical transform * horizontal transform (1a)

The two formulas for the vertical transform are :

$$y_{write} = y\tan\alpha + y_{read}\sec\alpha \Rightarrow y_{read} = y_{write}\cos\alpha - x\sin\alpha \ (2)$$

$$x_{read} = x_{write} \ (3)$$

The two formulas for the horizontal transform are:

$$x_{write} = x_{read}\cos\alpha - y\cos\alpha \Rightarrow x_{read} = x_{write}\sec\alpha - y\tan\alpha \ (4)$$

$$y_{read} = y_{write} \ (5)$$

To implement Stage 2, the formulas for each respective transform are implemented as two finite state machines.

Module Implementation

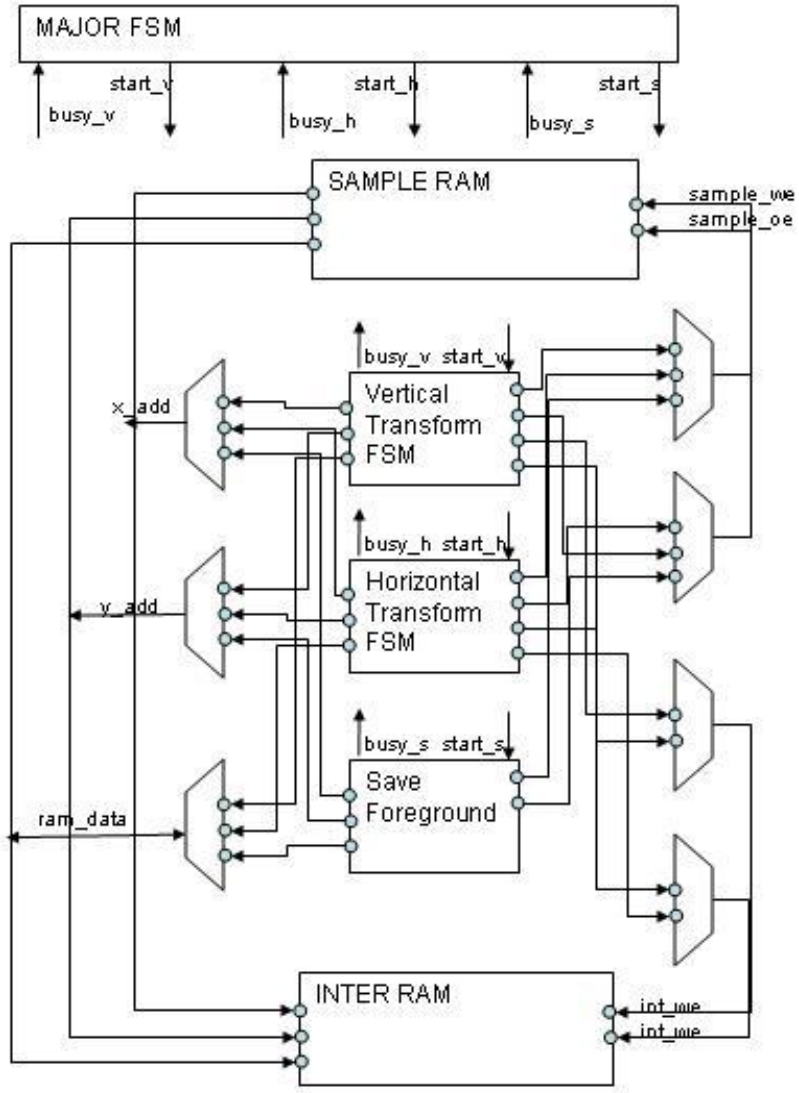


Figure 2a: Overall Block Diagram of Stage 2

The overall block diagram shows how the various modules and hardware are interconnected. The RAMs share an address and data in/out bus, and the various modules gain control of these busses according to the state of the major FSM. The major FSM coordinates the start signals of each module, and the modules operate independently.

Vertical Transform FSM

For Stage 2, the address generator maps the original image pixel address to the transformation image pixel address. A counter generates the write address of the vertically transformed images, and the corresponding pixel address of the source image is a function of the write address, denoted by Equations 2 and 3. These equations do not account for scaling of the foreground image, but scaling only adds a scaling coefficient factor in addition to the cosine factor, turning our final y_read address generator formula to be:

$$y_{read} = y_{write} * \text{scaling factor}^{-1} * \cos(\text{alpha}) - x_{write} * \tan(\text{alpha}) \quad (6)$$

The scaling factor $^{-1}$ * cosine is grouped together into s_coeff. S_coeff is the scaling coefficient that combines both the shrinking of the image and the scaling factor added when rotating the image. In order to save computation power, these scaling coefficients are pre-calculated and stored in an internal FPGA ROM. The user input module generates the ROM address for the s_coeff according the number of scale and rotate requests from the user.

In calculating these scaling coefficients, the number representation becomes an issue. The number representation is a fixed radix, sign-magnitude binary number with a 4-bit decimal part. In addition, the precision of our angles is also an issue, since only 16 decimal numbers can be represented this way, as shown in Table 1.

Table 1. Decimal Values

a_3	a_2	a_1	a_0	**Value**
0	0	0	0	0
0	0	0	1	0.0625
0	0	1	0	0.125
0	0	1	1	0.1875
0	1	0	0	0.250
0	1	0	1	0.3125
0	1	1	0	0.375
0	1	1	1	0.4375

1	0	0	0	0.5
1	0	0	1	0.5625
1	0	1	0	0.625
1	0	1	1	0.6875
1	1	0	0	0.75
1	1	0	1	0.8125
1	1	1	0	0.875
1	1	1	1	0.9375

The design only performs rotations of 15° and scaling down by 2 to 4 in increments of 0.25. Any more precision would require more decimal bits in the number.

The y_ptr is a 9-bit number. To perform fixed-radix multiplication, the y_ptr is extended with a 4-bit decimal. When y_ptr is multiplied with the 7-bit (3-bit integer, 4-bit decimal) s_coeff at the end of the COMPUTE state, a 20-bit number (12-bit integer part, 8-bit decimal part) is created. The integer bits and the 4 most significant decimal parts are used for the CHECK state.

In the CHECK state, the form of the read address dictates the next state. If the y_read address is negative, which occurs for the first set of read addresses since offset is subtracted from the read address, or if the y_read address is out of the bounds of the image (which is 256 x 240 pixels), then the value to be written into the intermediate RAM will be a blanking signal, noted as 255. The intermediate storage RAM's (int RAM) write enable signal is asserted to setup the RAM for storage and the FSM proceeds to the STORE state.

If the decimal part of the y_read address is 4'b0000, then the y_read address is a whole number, whereby no linear interpolation of pixels is necessary and the FSM proceeds to the READ state. Then the sample storage RAM's (sample RAM) output enable signal is asserted to retrieve data from the sample RAM. If there is a decimal part, the FSM enters the INTER_DATA0 state, where the linear interpolation process begins.

Since the address generator produces addresses with fractional values, such as 2.5, there is no physical 2.5 address location in RAM. To obtain a luminance value for such an address, linear interpolation produces a weighted average of two consecutive pixels, here the integer part of the y_read address and 1-plus that address. The states from INTER_DATA0 to INTER_DATA_MULT retrieves the two pixels from RAM and interpolates those two pixels. The new luminance value will be then stored in the STORE state.

In the READ state, the output enable signal to the sample RAM is held for one more clock cycle in order to receive valid data on the BUS. In the STORE state, the int RAM's write enable signal is asserted to prepare for storage of the luminance data.

The INC_Y state checks to see if the pointer has reached the y-axis resolution of the vertically transformed image. This number is pre-calculated and store in ROM. If it has reached the bottom of the image, it proceeds to increment the column of the image. If the x_ptr has reached the last column of the image, then the FSM returns to its IDLE state until the next frame is finished storing.

Horizontal Transform FSM

The horizontal transform FSM is exactly like the vertical transform FSM, with the exception that x_read is generated with the horizontal transform Equations 4 and 5, and the limits are related to the width of the image. Figures 2b and 2c illustrate the state transitions of the vertical and horizontal FSMs, respectively.

User Interface

The user interface consists of a scale and rotate button. The coeff_chooser module accumulates these button presses and the counters are mapped to addresses in various internal FPGA ROMs holding the various scaling coefficients for scaling and rotation, the x and y resolutions of the final image, and the module itself directly outputs the rotation offset values.

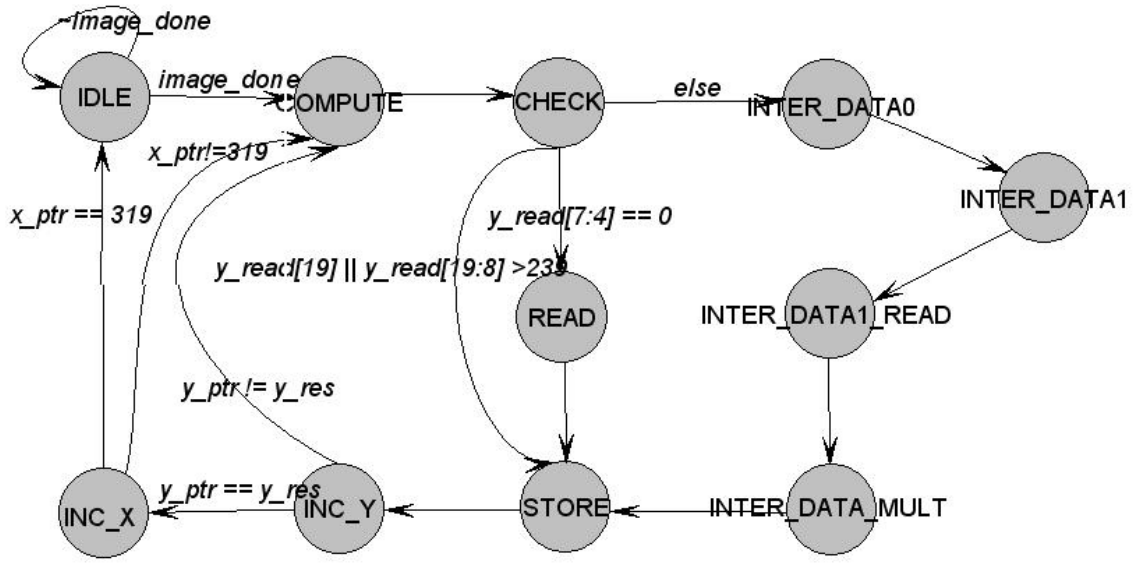


Figure 2b. Vertical Transform FSM

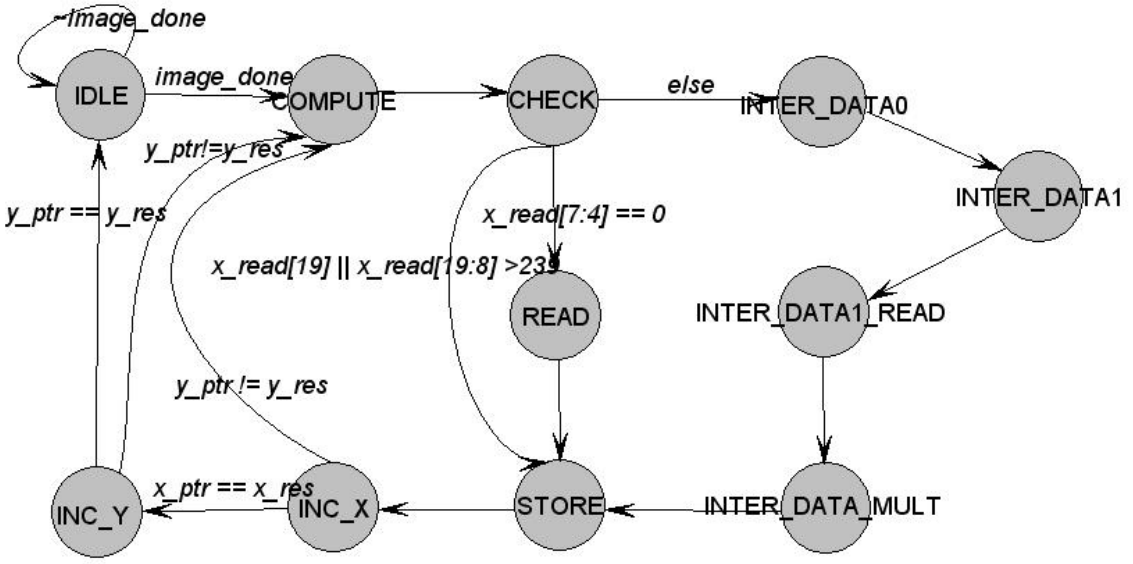
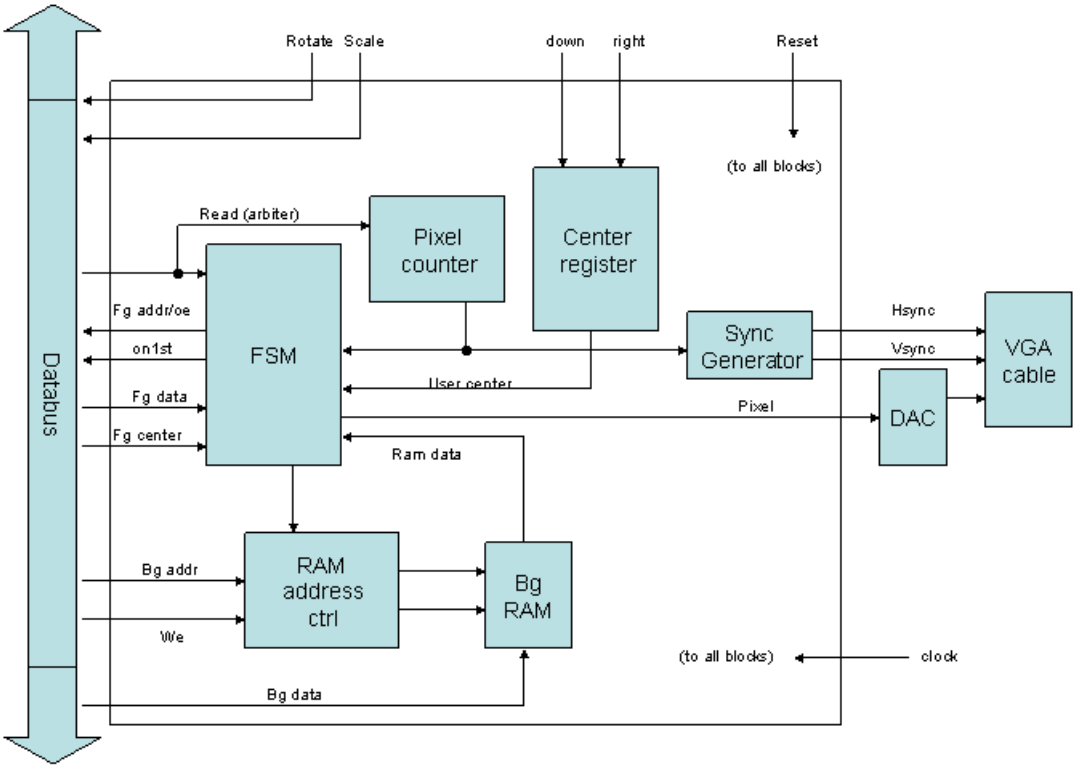


Figure 2c. Horizontal Transform FSM

Back End – Data Display and Positioning



Pixel Counter

The pixel counter keeps track of the current pixel being displayed on the screen. The data from the pixel counter is used in generating the hsync and vsync signals. It is also used in the determining the positioning of the foreground video within the background video.

Sync Generator

The Sync Generator creates the hsync and vsync outputs to the FPGA based on the pixel count. It also creates the signals which tell the FSM when to blank the RGB output. The Sync Generator is based on the 20 MHz clock.

RAM

Internal RAM was used for the background video since the amount of stored information could be limited to two lines. The internal RAM was used because it freed up 18 pins on the kits (8 address, we, oe, 8 data). Since the third stage reads the same line twice and the first stage takes twice as long to save a line, a line takes the same amount of time for each stage; as a result, the first and third stage are always one line apart and synchronized so that only two lines need to be saved into RAM. To make sure they weren't working on the same line, the first stage would digitize one line of video then send a start signal to the third stage at the beginning.

RAM Address Control of Background Ram

The address to the ram is controlled by this block. If it is the third stage's turn to read from the RAM, control of the address to the RAM is given to the third stage. If it is the first stage's turn to write to the RAM, access is given to the first stage, along with control of the write-enable. The output data from the RAM when stage 3 is reading is sent to the FSM for determining the RGB output.

Data to the RAM is set up such that the first 4 bits of data are the first pixel and the second 4 bits of data are the second pixel. This results in fewer reads and writes to the RAM and less address bits.

Center

The center block keeps track of where the foreground picture should be centered about. It has user inputs right (switch), go_right (button), down (switch), and go_down (button). If the right switch is high, then the user wants the image to move to the right. If it is low, then the user wants to move it to the left. A change is implemented only if the button is pressed. The down and go_down work similarly. The Center block counts up to a certain amount (approx $2.6/100^{th}$ of a second) before causing an update in the

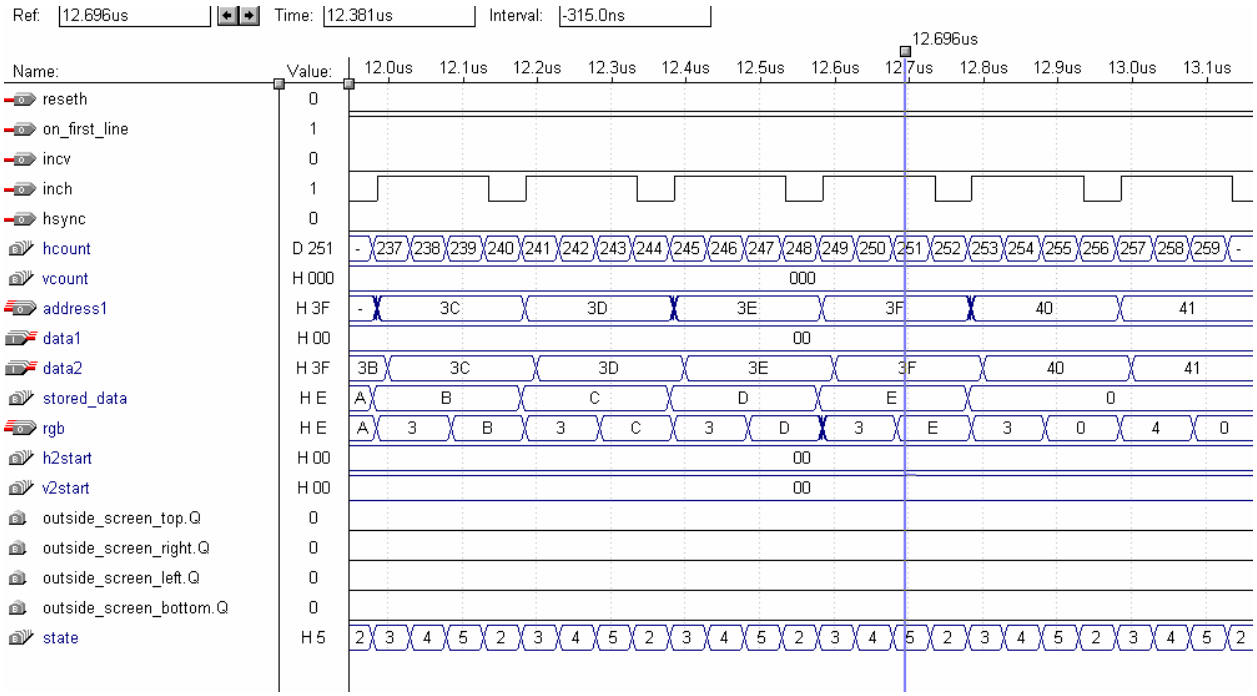
position of the foreground video, which is a shift of two pixels. Holding down the button will cause a continual change in the position of the foreground video. The output of the Center block is sent to the FSM to calculate the positioning of the foreground video.

The user is limited to values that are on the screen (cannot move the image off the screen). This design choice was made because if a user pushes an image off the screen, it could be quite hard to bring the image back on.

FSM

The FSM continuously reads from the background video and displays it. It calculates the position of the foreground video and determines when to start reading from the foreground RAM, since the image from the foreground is always at the upper left corner. At the start of each frame, it checks (based on the center position from the center block and the size of foreground image) whether the foreground image will end up off the screen. (The size of the foreground image is input from the middle stage.) If it ends up off the screen, it gets repositioned within the screen. This was decided because if a user decides to scale up the foreground image, it might result in part of the image being pushed off screen.

The FSM receives both the ram data for the foreground and the background when it is stage 3's cycle to read. Using that data, it determines the current pixel output as well as the next pixel output. The foreground pixel data contains a blanking signal 4'b1111, which tells the FSM to take the background pixel. This is for the event that the foreground image is not a perfect square (i.e. rotated); the foreground then becomes transparent. It also checks that the foreground image has started at the current pixel position, because if the foreground image starts at pixel (300, 159), and the output is currently at (200, 100), it should not display the data from the foreground RAM.



As the above diagram shows, h2start and v2start are 0, so the FSM should look at the data from the foreground image. h2start and v2start give the pixel location where the foreground should start, where the horizontal range is from 0 to 255 and the vertical range is from 0 to 239. This number is scaled up by a factor of 2 during output since the VGA display is 512 x 480. Notice that the RGB output takes on data2's first 4 bits then its second 4 bits unless data2's value is F, which is the blanking signal. In that event, it takes the RGB value from data1. Also notice that the foreground's addresses are being cycled through. The background's address is also being cycled through since the increment horizontal (inch) is high (except when the end of the line is reached).

The data that is received by the FSM is 8 bits. The FSM stores the portion of the pixel data which corresponds to the next pixel and outputs it when it reaches the next pixel location. The FSM also addresses the foreground and background appropriately. The addressing for the foreground is based on 4 signals – increment the vertical count, increment the horizontal count, reset the horizontal count, and reset the vertical count. This is another attempt to save pins, since the foreground RAM has a total of 16 address bits. Also, the FSM reads the addresses from the foreground sequentially, which allows this implementation.

The states of the FSM (ignoring the initial synchronization with the front end) are as follows: There are two clocks cycles, in which the third stage gains control of the foreground and background RAMs. It takes one of the clock cycles to set up the address to the RAMs (the first state) and the other clock cycle to wait for the data to become valid (the second state). It takes these two clock cycles to read 8 bits from each (8 bits = 2 pixels). It displays the appropriate pixel based on its calculations.

For the next two clock cycles, it does not have control of the RAMs. Because it has saved the next pixel to display, it does not need access to the RAMs. It has two states in which it waits for its turn to access the RAM again, during which it updates the RAM address to be sent to RAM on it's next turn. Then the FSM regains control of the bus and it repeats through the cycle. The FSM calculates and updates the positioning of the foreground image only at the start of a new frame, so that the foreground image would not appear disjoint.

Debugging

The datasheet for the GS 4981 states that one needs quite a complicated circuit with a 75 ohm resistor, a low pass filter, a Common collector amplifier and a high pass filter. After building that circuit and testing it out, the video signal that entered the GS 4981 seemed like noise and the chip was properly reporting Vsync signals. Through experimenting it was should that all was necessary to generate a Vsync and Hsync signal was a 75 ohm terminating resistor and a .01 pF coupling capacitor.

There seemed to be a lot of noise on the signal still. Moving the analog circuit off the kit and onto a protoboard didn't help to remove the noise and made it difficult to wire to the FPGA so it was just put back on.

Testing the effect of this noise was difficult since the displaying the signal part of the design wasn't completed yet.

The individual modules were tested and simulated using Max Plus II.

Testing and Debugging

The primary method for debugging the Verilog code was using the MAX+plusII simulator. The simulations helped discover any problems with address generation and RAM control signal timing. Figure 2d below shows a simulator of the y_add being generated by the Vertical Transform FSM. For this simulation, a scaling down by 2 and rotation of 45° (0 offset for the 0th column) should generate y read addresses that are 1.4375-times the y pointer addresses. As the generator sends the integer part of the y_read address, the screenshot demonstrates how the pixel values at address locations 1 and 2 are being read from the sample RAM, filtered and stored into address location 1 in the intermediate RAM. In addition, the sample_oe_bar was asserted low for both read cycles, and the int_we_bar was asserted low for the write cycle, correctly producing the needed RAM control signals.

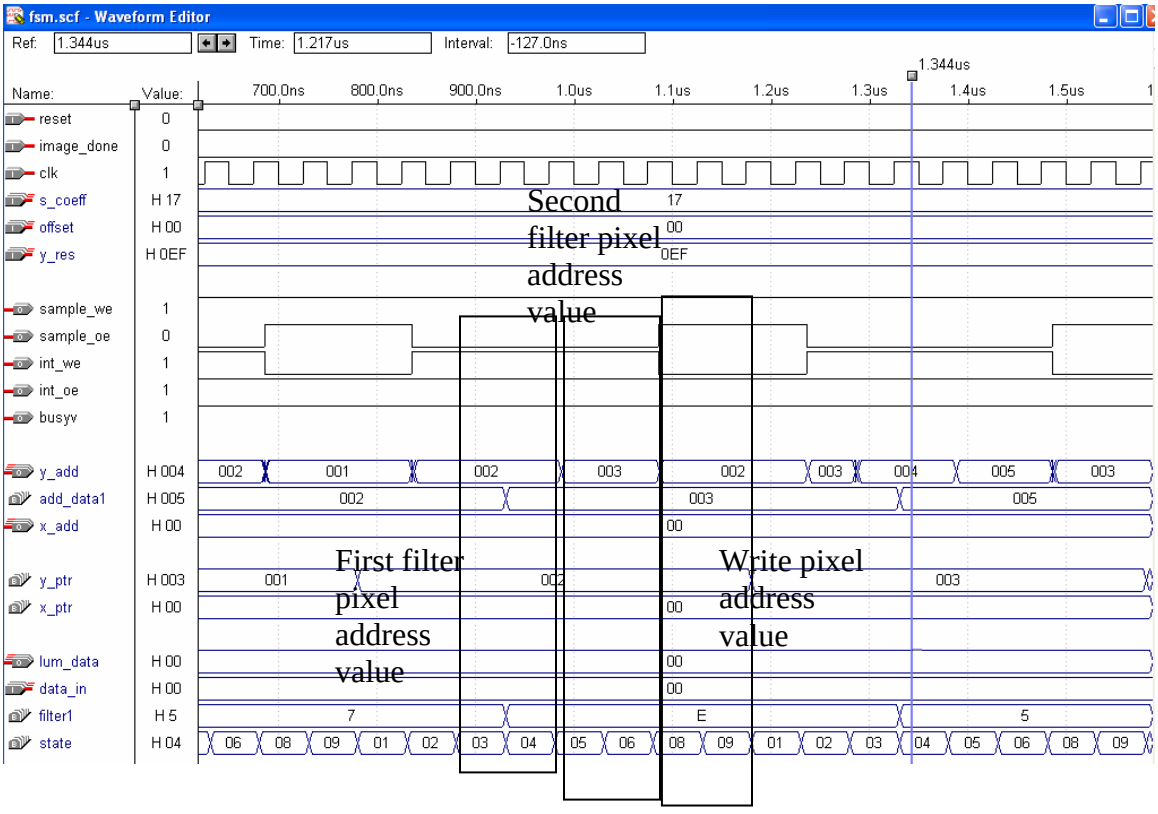


Figure 2d. Simulation of Address Generator

Since there was no stored sample image for Stage 2 to test its transformations, I wrote code to store to RAM a solid colored block, performed the operations on that image, and used a simplified version of the back-end code to display it. Ultimately, I was unable to see a rotated image on the screen, but have on various occasions generated parts of a rotated rectangle. The reason behind the difficult in generating an image is the high peak-to-peak ringing of the RAMs control signals generated from the FPGA. These voltages do in fact cross below the V_{IH} of 2.2 V for the RAM, and this will have adverse effects on image storage.

Debugging of Third Stage

To debug the output stage, internal RAMs were created for foreground and background outputs. Writing the RAM with known values, the third stage could then output what was in RAMs and do the appropriate calculations to position the foreground image. Internal RAM was chosen over external RAM because external RAM was much less reliable, which would add more possible problems. Limited to small RAM sizes, a small foreground image was created (with blanking signals) and addressing was controlled such that if it overflowed the actual RAM address (which was 9 bits instead of the real 16 bits), it would read from an address which was known to contain the blanking signal. This was to simulate the blanking signals that would be contained in the foreground RAM at locations outside of the foreground image.

Simulations were used extensively in the debugging process. However, with output that depended on such a huge range in time (frame output at $1/60^{th}$ second), simulations were sometimes difficult to parse and took a long amount of time to simulate. In simulating modules, backup modules with all important signals were created as output to verify their operation. Simulations of output were often difficult to understand since the actual image being produced could not be seen. On the other hand, the third stage dealt with output, so it was easier to verify the operation of the modules based on the output. Since the first and second stages did not have a direct output to look at, it was more difficult for those stages to verify their operation.

Conclusions

Hardware limitations presented a much larger problem than we had anticipated. It is very difficult (near impossible) to connect more than two lab kits together and still have enough pins. The data and control signals from one FPGA to another did not travel well at 20 MHz. Even in connecting only two lab kits, the data does not travel well across the 50 pin ribbon cable. We quickly ran out of pins for the ram addresses and ram data, which resulted from the large amount of video data being stored into ram, as well as the computation being carried out on the data.

```
// Stage I - Edward Semper

module addr_counter(clk, reset, hinc, hreset, vinc, vreset, addr);
// This module implements an address counter to lessen the number of pins that the 16 bit
// address takes up on the kit. Since counting is done incrementally. Only 4 pins are needed
// to get the appropiate address.

input clk, reset, hinc, hreset, vinc, vreset;
output [15:0] addr;

reg [15:0] addr;

always @ (posedge clk)
begin
addr[7:0] <= ~reset*~hreset*(hinc ? {addr[7:0]} + 1 : addr[7:0]);
addr[15:8] <= ~reset*~vreset*(vinc ? {addr[15:8]} + 1 : addr[15:8]);
end

endmodule
```

```
// Stage I - Edward Semper
module arbiter (clk, reset,
                    bgrbar, fwbar, midwbar,
                    s1w2, s2w3, s1w3, s3r
                );
// This module handles the access to the two RAMs connected the the FPGA. It is based on a 25 MHz
clock.
// It signals the read time and the write time to the other modules. It gives 80 ns for each
cycle.
// fg represents the signals connected with the foreground RAM
// bg represents the signals connected with the background RAM

// removes the s3r_state FSM
// removes addressing
// resets 3 inputs... four outputs

// corrects the spelling... and adjusts for bus delay in input signals

input clk, reset;
input bgrbar, fwbar, midwbar;
output s1w2, s2w3, s1w3, s3r;


// The States
// state[0] : 1 when storing bg first line
// state[1] : flag for first line being completed
// state[2] : state counter;
// state[3] : 0 = write; 1 = read; for output to screen
// state[4] : 0 = s1 time on the bus. s2 time on the bus
reg [4:0] state;

always @ (posedge clk)
begin
if (reset) state <= 0;
else if (state==0) state <= bgrbar ? 1 : 0;
else if (state==1) state <= bgrbar ? 1 : 3;
else state <= (state[1:0]==3) ? state + 4 : 0;
end

wire rwbar;
assign rwbar = state [3];

reg fgwbar;
reg s1w2, s2w3, s1w3, s3r;

always @ (posedge s3r)
fgwbar <= fwbar;

always @ (posedge clk)
begin
s1w2 <= ~(~rwbar*state[4]*~midwbar);
s1w3 <= rwbar||state[4];
s2w3 <= ~(~rwbar*state[4]*~fgwbar*~midwbar);
s3r  <= rwbar;
end
endmodule
```

```
// Stage I - Edward Semper

module storebg(clk, reset, hsync, sample, odd, wbar, addr, we, on1st,
                bgdata_in, bgdata_out, done, nextstate);
// This module is the FSM for storing the background image into RAM.
// It is idle until the beginning of the odd frame which is indicated by the odd signal.
// It then waits for the picture information to begin (after 11 lines has passed)


input clk, reset, hsync, sample, odd, wbar;
input [3:0] bgdata_in;
output [7:0] bgdata_out;
output on1st, we, done;
output [7:0] addr;
output [2:0] nextstate;

reg [2:0] state, nextstate;
//states
parameter wait_odd = 0;     // waits for the start of the odd frame
parameter wait_image = 1;   // waits for the off-screen lines to pass
parameter wait_sample = 2;  // waits for pixel
parameter write_data = 3;
parameter wait_wbar = 4;    // waiting for the write signal
parameter update_haddr = 5; // update horizontal address counter
parameter update_vaddr = 6; // update vertical address counter
parameter wait_hsync = 7;   // waits for end of line

//vertical and horizontal counters
reg [8:0] hcount;
reg [7:0] vcount;

reg image, we, done;
reg [7:0] addr;
wire on1st;
wire pixreg_sel;

always @ (posedge clk)
begin
if (reset) state <= 0;
else state <= nextstate;
hcount <= (reset || image || (nextstate==update_vaddr)) ? 0 : ((nextstate==wait_image) ? (hsync ?
hcount + 1 : hcount) : ((nextstate==update_haddr) ? hcount + 1 : hcount));
vcount <= reset ? 0 : ((nextstate==update_vaddr) ? vcount + 1 : vcount);

image <= (state==wait_image)&&(hcount==11); // image starts after the first 11 lines of the video
frame
addr <= {vcount[0],hcount[7:1]};            // storing pixels 2 at a time in RAM.
we <= (nextstate==write_data)*pixreg_sel;
end

assign on1st = ~vcount[0];                  // indicates when storing the first line of image

reg [3:0] pix0, pix1;
assign bgdata_out = {pix0,pix1};
assign pixreg_sel = hcount[0];

always @ (posedge clk)
begin
if (pixreg_sel) pix1 <= bgdata_in;
else pix0 <= bgdata_in;
done <= (nextstate == wait_odd);
end

always @ (state or odd or image or sample or hcount or vcount or wbar)
case (state)
wait_odd: nextstate <= odd ? wait_image : state;
wait_image : nextstate <= image ? wait_sample : state;
wait_sample : nextstate <= sample ? (wbar ? wait_wbar : write_data) : state;
```

```
write_data : nextstate <= update_haddr;
wait_wbar : nextstate <= ~wbar ? write_data : state;
update_haddr : nextstate <= (hcount==285) ? update_vaddr : wait_sample;
update_vaddr : nextstate <= wait_hsync;
wait_hsync : nextstate <= hsync ? ((vcount==240) ? wait_odd : wait_sample) : state;
endcase

endmodule
```

```
// Stage I - Edward Semper

module storefg(clk, reset, hsync, sample, odd, wbar, haddr_inc, hreset, vaddr_inc, vreset, webar,
                data_in, data_out, done);
// This module is the FSM for storing the foreground image into RAM.
// It assumes that it is connected to a special counter that will correctly handle
// the addressing so that the interlaced images are handled correctly.
// It is idle until the beginning of the odd frame which is indicated by the odd signal.
// It then waits for the picture information to begin (after 11 lines has passed)

input clk, reset, hsync, sample, odd, wbar;
input [3:0] data_in;
output [3:0] data_out;
output webar, done;
output haddr_inc, hreset, vaddr_inc, vreset;

reg [2:0] state, nextstate;
//states
parameter wait_odd = 0;     // waits for the start of the odd frame
parameter wait_image = 1;   // waits for the off-screen lines to pass
parameter wait_sample = 2;  // waits for pixel
parameter write_data = 3;
parameter wait_wbar = 4;    // waiting for the access time on the Bus
parameter update_haddr = 5; // update horizontal address counter
parameter update_vaddr = 6; // update vertical address counter
parameter wait_hsync = 7;   // waits for end of line

//vertical and horizontal counters
reg [8:0] hcount;
reg [7:0] vcount;

reg image, webar, done;
reg haddr_inc, hreset, vaddr_inc, vreset;
reg [7:0] addr;
wire on1st;
wire pixreg_sel;


reg [3:0]data_out;
always @ (posedge clk)
begin
if (reset) state <= 0;
else state <= nextstate;
hcount <= (reset || image || (nextstate==update_vaddr)) ? 0 : ((nextstate==wait_image) ? (hsync ?
hcount + 1 : hcount) : ((nextstate==update_haddr) ? hcount + 1 : hcount));
vcount <= reset ? 0 : ((nextstate==update_vaddr) ? vcount + 1 : vcount);

image <= (state==wait_image)&&(hcount==11); // image starts after the first 11 lines of the video
frame
webar <= ~(nextstate==write_data);

haddr_inc <= (nextstate==update_haddr);
hreset <= (nextstate==update_vaddr)||(nextstate==wait_odd);
vaddr_inc <= (nextstate==update_vaddr);
vreset <= (nextstate==wait_odd);

done <= (nextstate == wait_odd);
data_out <= data_in;
end

always @ (state or odd or image or sample or hcount or vcount or wbar)
case (state)
wait_odd: nextstate <= odd ? wait_image : state;
wait_image : nextstate <= image*~wbar ? wait_sample : state;
wait_sample : nextstate <= sample ? (wbar ? wait_wbar : write_data) : state;
write_data : nextstate <= update_haddr;
wait_wbar : nextstate <= ~wbar ? write_data : state;
update_haddr : nextstate <= (hcount==256) ? update_vaddr : wait_sample;
```

```
update_vaddr : nextstate <= wait_hsync;
wait_hsync : nextstate <= hsync ? ((vcount==240) ? wait_odd : wait_sample) : state;
endcase

endmodule
```

```
// Stage II - Edward Semper

module storefg_out(clk, reset, start, wbar, source_addr, we_dest, oe_source,
                haddr_inc, hreset, vaddr_inc, vreset,
                data_in, data_out, done);
// This module is the FSM for storing the completed foreground image into RAM.
// It is idle until receiving a start signal.
// It saves two pixels per address.

input clk, reset, start, wbar;
input [3:0] data_in;
output [7:0] data_out;
output we_dest, oe_source, done;
output [15:0] source_addr;
output haddr_inc, hreset, vaddr_inc, vreset;

reg [2:0] state, nextstate;
//states
parameter idle = 0;     // waits for the start of the odd frame
parameter get_data = 1; // waits for the off-screen lines to pass
parameter write_data = 2;
parameter update_haddr = 3; // update horizontal address counter
parameter update_vaddr = 4; // update vertical address counter

//vertical and horizontal counters
reg [8:0] hcount;
reg [7:0] vcount;

reg we_dest, oe_source, done;
reg haddr_inc, hreset, vaddr_inc, vreset;
reg [15:0] source_addr;

wire pixreg_sel;

always @ (posedge clk)
begin
if (reset) state <= 0;
else state <= nextstate;
hcount <= (reset || (nextstate==update_vaddr)) ? 0 : ((nextstate==update_haddr) ? hcount + 1:
hcount);
vcount <= reset ? 0 : ((nextstate==update_vaddr) ? vcount + 1 : vcount);

haddr_inc <= (nextstate==update_haddr);
hreset <= (nextstate==update_vaddr)||(nextstate==idle);
vaddr_inc <= (nextstate==update_vaddr);
vreset <= (nextstate==idle);

source_addr <= {vcount[7:0],hcount[7:0]};
we_dest <= ~((nextstate==write_data)*pixreg_sel);
oe_source <= ~((nextstate==get_data)||(nextstate==write_data));
end

reg [3:0] pix0, pix1;
assign data_out = {pix0,pix1};
assign pixreg_sel = hcount[0];

always @ (posedge clk)
begin
if (pixreg_sel) pix1 <= data_in;
else pix0 <= data_in;
done <= (nextstate == idle);
end

always @ (state or start or hcount or vcount or wbar)
case (state)
idle: nextstate <= start ? get_data : state;
get_data : nextstate <= wbar ? write_data : state;
write_data : nextstate <= update_haddr;
```

```
update_haddr : nextstate <= (hcount==256) ? update_vaddr : get_data;
update_vaddr : nextstate <= (vcount==240) ? idle : get_data;
endcase

endmodule
```

```
// Stage I - Edward Semper

module sync(clk,in,out);

//This module is designed to synchronize a button press to the clock by feeding the input through
//two registers in order to prevent against metastability.

input clk, in;
output out;

reg r, out;
always @ (posedge clk)
begin
r <= in;
out <= r;
end

endmodule
```

```
// Stage 2 : Vincent Wu

module top_final(clk, reset, sample_oe, int_we, sample_we, int_oe,x_add,y_add,ram_data,
vsync,hsync,rgb, state);

input clk, reset;
inout [7:0] ram_data;

output [7:0] x_add;
output [8:0] y_add;
output sample_oe, int_we, sample_we, int_oe, vsync, hsync;
output [3:0] rgb;
output [3:0] state;

parameter zero = 9'b000000000;
wire [7:0] bus_data;
wire we;

wire start_v, start_h, start_s, busy_v, busy_h, busy_s, busy_w, start_w;
wire [7:0] hout_bus, vout_bus, hin_bus, vin_bus, sin_bus;
wire vs_we, vs_oe, vi_oe, vi_we, hs_we, hs_oe, hi_we, hi_oe, ss_oe, ws_we;
wire [7:0] v_xadd, h_xadd, w_xadd;
wire [8:0] v_yadd, h_yadd, w_yadd;
wire [16:0] s_add;


vertical_fsm top_fsm_vertical (
  .clk(clk),
  .image_done(start_v),
  .reset(reset),
  .data_in(ram_data),
  .lum_data(vout_bus),
  .x_add(v_xadd),
  .y_add(v_yadd),
  .sample_oe(vs_oe),
  .int_we(vi_we),
  .busyv(busy_v),
  .sample_we(vs_we),
  .int_oe(vi_oe));

horizontal_fsm top_fsm_hor (
  .clk(clk),
  .image_done(start_h),
  .reset(reset),
  .data_in(ram_data),
  .lum_data(hout_bus),
  .x_add(h_xadd),
  .y_add(h_yadd),
  .sample_oe(hs_oe),
  .int_we(hi_we),
  .busyh(busy_h),
  .sample_we(hs_we),
  .int_oe(hi_oe));

major_fsm top_major (
  .clk(clk),
  .reset(reset),
  .start_v(start_v),
  .start_h(start_h),
  .start_s(start_s),
  .busy_v(busy_v),
  .busy_h(busy_h),
  .busy_s(busy_s),
  .busy_w(busy_w),
  .start_w(start_w),
  .state(state));

ramwrite topramwrite(
```

```
  .clk(clk),
  .start(start_w),
  .busy(busy_w),
  .reset(reset),
  .x_add(w_xadd),
  .y_add(w_yadd),
  .lum_data(wout_bus),
  .sample_we(ws_we));

//demov output code provided by Sue Zheng
demov9 testdemo(
  .clk(clk),
  .reset(reset),
  .ramdata(ram_data),
  .addresstoram(s_add),
  .rgb(rgb),
  .hsync(hsync),
  .vsync(vsync),
  .done(start_s),
  .busy(busy_s));

assign sample_oe = (state == 2) ? vs_oe :
                   (state == 4) ? 1 :
                   (state == 6) ? 0 : 1;

assign int_oe = (state == 2) ? vi_oe :
                (state == 4) ? hi_oe : 1;


assign sample_we = (state == 2) ? vs_we :
                   (state == 4) ? hs_we :
                   (state == 7) ? ws_we : 1;

assign int_we = (state == 2) ? vi_we :
                (state == 4) ? hi_we : 1;

assign x_add = (state == 7) ? w_xadd:
               (state == 2) ? v_xadd :
               (state == 4) ? h_xadd :
               (state == 6) ? s_add[7:0] : 8'b00000000;

assign y_add = (state == 7) ? w_yadd:
               (state == 2) ? v_yadd :
               (state == 4) ? h_yadd :
               (state == 6) ? s_add[16:8] : zero;

assign bus_data = ((state == 7) && ~sample_we) ? wout_bus:
               ((state == 2) && ~int_we) ? vout_bus :
               ((state == 4) && ~sample_we) ? hout_bus : 8'b00000000;

assign we = ((state == 2) && ~int_we) || ((state == 4) && ~sample_we) || ((state == 7) &&
~sample_we);

assign ram_data = we ? bus_data : 8'hzz;

endmodule
```

```
// Stage 2 : Vincent Wu

module synchronizer (clk, reset, scale, rotate, scale_sync, rotate_sync,
translate_sync,reset_sync);

  input clk, reset;
  input scale, rotate;

  output scale_sync, rotate_sync,  reset_sync;

  reg reset1, reset2;
  reg scale1, scale2, scale3;
  reg rotate1, rotate2, rotate3;


  always @ (posedge clk) begin
    reset1 <= reset;
    scale1 <= scale;
    rotate1 <= rotate;

    reset2 <= reset1;
    scale2 <= scale1;
    rotate2 <= rotate1;

    scale3 <= scale2;
    rotate3 <= rotate2;
  end

  assign reset_sync = reset2;
  assign scale_sync = ~scale3 & scale2;
  assign rotate_sync = ~rotate3 & rotate2;

endmodule
```

```
// Stage 2 : Vincent Wu

module horizontal_fsm (clk,image_done,reset,x_add,y_add,sample_oe, int_we, busyh, data_in,
sample_we, int_oe,lum_data,s_coeff,offset,y_res);

  input clk, image_done, reset;
  input [6:0] s_coeff;
  input [4:0] offset;
  input [8:0] y_res;

  input [7:0] data_in;
  output [7:0] lum_data;

  output [7:0] x_add;
  output [8:0] y_add;
  output sample_oe, int_we, busyh, sample_we, int_oe;

  reg [12:0] offset_acc;
  reg [7:0] x_add;
  reg [8:0] y_add;

  reg [7:0] x_ptr;
  reg [8:0] y_ptr;
  reg [3:0] state;
  reg [19:0] x_read;
  reg [7:0] lum_data;
  reg [11:0] lum_dataA;
  reg [11:0] lum_dataB;
  reg [8:0] add_data1;
  reg [4:0] filter0;
  reg [3:0] filter1;
  reg sample_oe, int_we, busyh, sample_we, int_oe;
  wire busy;

  parameter IDLE = 0;
  parameter COMPUTE = 1;
  parameter CHECK = 2;
  parameter INTER_DATA0 = 3;
  parameter INTER_DATA1 = 4;
  parameter INTER_DATA1_READ = 5;
  parameter INTER_DATA_MULT = 6;
  parameter READ_DATA = 7;
  parameter STORE_DATA = 8;
  parameter INC_X= 9;
  parameter INC_Y= 10;

  parameter PUREWHITE = 8'b11110000;
  parameter BLANKING = 8'b11111111;
  parameter COMPRESS = 8'b11100000;


  always @ (posedge clk) begin
    x_add <= x_ptr;
    y_add <= y_ptr;
    sample_oe <= 1;
    sample_we <=1;
    int_we <= 1;
    int_oe <= 1;
    busyh<= busy;
    if (reset) state <= IDLE;
    else case (state)
      IDLE : begin
        state <= image_done ? COMPUTE : IDLE;
      end

      COMPUTE : begin
        int_oe <= 0;
        x_read <= ({x_ptr,4'b0000} * s_coeff) - {3'b000,offset_acc,4'b0000};
```

```
  state <= CHECK;
end

CHECK : begin
  int_oe <= 0;
  x_add <= x_read[15:8];
  if (x_read[19] || (x_read[19:8] > 255)) begin
    int_oe <=1;
    sample_we <=0;
    x_add <= x_ptr;
    lum_data <= BLANKING;
    state <= STORE_DATA;
  end
  else if (x_read[7:4] == 0) begin
    state <= READ_DATA;
  end
  else begin
    state <= INTER_DATA0;
  end
end

INTER_DATA0 : begin
  int_oe <= 0;
  x_add <= x_read[15:8];
  add_data1 <= x_read[15:8] + 1;
  filter0 <= 5'b10000 - {1'b0,x_read[7:4]};
  filter1 <= x_read[7:4];
  lum_dataA <= data_in * filter0[3:0];
  state <= (add_data1 > 255) ? READ_DATA : INTER_DATA1;
end

INTER_DATA1 : begin
  int_oe <= 0;
  x_add <= add_data1;
  state <= INTER_DATA1_READ;
end

INTER_DATA1_READ : begin
  int_oe <= 0;
  x_add <= add_data1;
  lum_dataB <= (data_in * filter1);
  state <= INTER_DATA_MULT;
end

INTER_DATA_MULT : begin
  sample_we <= 0;
  lum_data <= (lum_dataA[11:4] + lum_dataB[11:4]);
  state <= STORE_DATA;
end

READ_DATA : begin
  int_oe <=0;
  int_we <=1;
  x_add <= x_read[15:8];
  lum_data <= data_in;
  state <= STORE_DATA;
end

STORE_DATA : begin
  int_oe<= 1;
  sample_we <= 0;
  state <= INC_X;
end

INC_X : begin
  sample_we <=0;
  x_ptr <= x_ptr + 1;
  state <= (x_ptr == 255) ? INC_Y : COMPUTE;
```

```
      end

      INC_Y : begin
        y_ptr <= y_ptr + 1;
        x_ptr <= 0;
        offset_acc <= offset_acc + {8'b00000000,offset};
        state <= (y_ptr == y_res) ? IDLE : COMPUTE;
      end
   endcase //end case
  end //end always
assign busy = ~(state == IDLE);
endmodule
```

```
// Stage 2 : Vincent Wu

module vertical_fsm (clk,image_done,reset,x_add,y_add,sample_oe, int_we, busyv, data_in, lum_data,
sample_we, int_oe, s_coeff, offset, y_res);

  input clk, image_done, reset;
  input [6:0] s_coeff;
  input [4:0] offset;
  input [8:0] y_res;

  input [7:0] data_in; //data from RAM
  output [7:0] lum_data; // data being stored to into ram

  output [7:0] x_add; //column number
  output [8:0] y_add; // row number
  output sample_oe, int_we, sample_we, int_oe, busyv; //ram ctrl & major fsm signals

  reg [12:0] offset_acc; //adds up offset from column to column
  reg [7:0] x_add;
  reg [8:0] y_add;

  reg [7:0] x_ptr;
  reg [8:0] y_ptr;
  reg [4:0] state;
  reg [19:0] y_read; // {[19:17] carryover bits[16:8] integer bits ,[7:4] decimal bits,[3:0] extra
decimal bits}
  reg [7:0] lum_data;
  reg [11:0] lum_dataA;
  reg [11:0] lum_dataB;
  reg [8:0] add_data1;
  reg [4:0] filter0;
  reg [3:0] filter1;
  reg sample_oe, int_we, int_oe, sample_we,busyv;
  wire busy;

  parameter IDLE = 0;
  parameter COMPUTE = 1;
  parameter CHECK = 2;
  parameter INTER_DATA0 = 3;
  parameter INTER_DATA1 = 4;
  parameter INTER_DATA1_READ = 5;
  parameter INTER_DATA_MULT = 6;
  parameter READ_DATA = 7;
  parameter STORE_DATA = 8;
  parameter INC_Y= 9;
  parameter INC_X= 10;

  parameter PUREWHITE = 8'b11110000;
  parameter BLANKING = 8'b11111111;
  parameter COMPRESS = 8'b11100000;

  always @ (posedge clk) begin
    x_add <= x_ptr;
    y_add <= y_ptr;
    sample_oe <= 1;
    int_we <= 1;
    sample_we <=1;
    int_oe <= 1;
    busyv <= busy;
    if (reset) state <= IDLE;
    else case (state)
      //waiting for signal to indicate image storage is finished
      IDLE : begin
        state <= image_done ? COMPUTE : IDLE;
      end

      //compute Y read address using formula
      //store address * scaling coefficent - col# * offset = yread address
```

```
COMPUTE : begin
  sample_oe <= 0;
  y_read <= ({y_ptr,4'b0000} * s_coeff) - {3'b000,offset_acc,4'b0000};
  state <= CHECK;
end

//delegate next state according to form of yaddress
//if read address is out of bounds, store a blanking signal
// if decimal part, linearly interpolate
//if whole number, proceed to reading and storing
CHECK : begin
  sample_oe <= 0;
  sample_we <= 1;
  y_add <= y_read[16:8];
  if (y_read[19] || (y_read[19:8] > 239)) begin
    sample_oe <= 1;
    int_we <= 0;
    y_add <= y_ptr;
    lum_data <= BLANKING;
    state <= STORE_DATA;
  end
  else if (y_read[7:4] == 0) begin
    state <= READ_DATA;
  end
  else begin
    state <= INTER_DATA0;
  end
end

//read the first first of data to interpolate, which is the integer part
//assign weighted average values according to decimal part of yread address
INTER_DATA0 : begin
  sample_oe <= 0;
  sample_we <= 1;
  y_add <= y_read[16:8];
  add_data1 <= y_read[16:8] + 1;
  filter0 <= 5'b10000 - {1'b0,y_read[7:4]};
  filter1 <= y_read[7:4];
  lum_dataA <= (data_in * filter0[3:0]);
  state <= (add_data1 > 239) ? READ_DATA : INTER_DATA1;
end

// assert signals for reading second data value for interpolation
INTER_DATA1 : begin
  sample_oe <= 0;
  sample_we <= 1;
  y_add <= add_data1;
  state <= INTER_DATA1_READ;
end

// asserts for an extra clock cycle to ensure hold times for valid data
INTER_DATA1_READ : begin
  sample_oe <= 0;
  sample_we <= 1;
  y_add <= add_data1;
  lum_dataB <= (data_in * filter1);
  state <= INTER_DATA_MULT;
end

// computes the weighted average of the two pixels retrieved
INTER_DATA_MULT : begin
  int_we <= 0;
  int_oe <= 1;
  lum_data <= (lum_dataA[11:4] + lum_dataB[11:4]);
  state <= STORE_DATA;
end

// assert control signals for write hold times, compress down 'white white' values
```

```
     // since that is our blanking signal
     READ_DATA : begin
       sample_oe <= 0;
       sample_we <= 1;
       y_add <= y_read[16:8];
       lum_data <= (data_in == PUREWHITE) ? COMPRESS : data_in;
       state <= STORE_DATA;
     end

    // assert control signals for write hold times
     STORE_DATA : begin
       int_we <= 0;
       int_oe <= 1;
       state <= INC_Y;
     end

       // increment y, unless y_ptr is y_resolution of the image
     INC_Y : begin
       int_we <= 0;
       int_oe <= 1;
       y_ptr <= y_ptr + 1;
       state <= (y_ptr == y_res) ? INC_X : COMPUTE;
     end

       // increment X
     INC_X : begin
       x_ptr <= x_ptr + 1;
       y_ptr <= 0;
       offset_acc <= offset_acc + {8'b00000000,offset};
       state <= (x_ptr == 255) ? IDLE : COMPUTE;
     end
   endcase //end case
  end //end always

assign busy = ~(state == IDLE);
endmodule
```

```
// Stage 2 : Vincent Wu

module coeff_chooser (clk, reset, scale, rotate, v_scale_add, h_scale_add, v_res_add, h_res_add,
v_offset, h_offset);

/*This module takes in synchronized, pulsated scale and translate button presses,
counts them, and interprets them as ROM addresses for scaling coefficients and
rotation offsets*/

  input clk, reset, scale, rotate;

  output [5:0] v_scale_add, h_scale_add, v_res_add, h_res_add;
  output [4:0] v_offset, h_offset;

  reg[3:0] scale_cnt;
  reg[1:0] rotate_cnt;

  always @ (posedge clk) begin
    if (reset) begin
      scale_cnt <= 0;
      rotate_cnt <= 0;
    end
    else if (scale) begin
      if (scale_cnt == 8) scale_cnt <=0;
      else scale_cnt <= scale_cnt + 1;
    end
    else if (rotate) begin
      rotate_cnt <= rotate_cnt + 1;
    end
    else begin
      scale_cnt <= scale_cnt;
      rotate_cnt <= rotate_cnt;
    end
  end

  assign v_scale_add[5:2] = scale_cnt;
  assign v_scale_add[1:0] = rotate_cnt;
  assign h_scale_add[5:2] = scale_cnt;
  assign h_scale_add[1:0] = rotate_cnt;

  assign v_res_add[5:2] = scale_cnt;
  assign v_res_add[1:0] = rotate_cnt;
  assign h_res_add[5:2] = scale_cnt;
  assign h_res_add[1:0] = rotate_cnt;

  assign v_offset = (rotate_cnt == 0) ? 5'b00000 :
                    (rotate_cnt == 1) ? 5'b00100 :
                    (rotate_cnt == 2) ? 5'b01000 :
                    (rotate_cnt == 3) ? 5'b01011 : 5'b00000;

  assign h_offset = (rotate_cnt == 0) ? 5'b00000 :
                    (rotate_cnt == 1) ? 5'b01001 :
                    (rotate_cnt == 2) ? 5'b01000 :
                    (rotate_cnt == 3) ? 5'b10000 : 5'b00000;

endmodule
```

```
// Stage 2 : Vincent Wu
module major_fsm (clk, reset, start_w, start_v, start_h, start_s, busy_v, busy_h, busy_s, busy_w,
state, scale, rotate);

  input clk, reset, scale, rotate;
  input busy_v, busy_h, busy_s, busy_w;

  output start_v, start_h, start_s, start_w;
  output [3:0] state;

  reg start_v, start_h, start_s, start_w;
  reg [3:0] state, next;

  always @ (posedge clk) begin
    state <= (reset | scale | rotate) ? 0 : next;
  end

  always @ (state or busy_v or busy_h or busy_s or busy_w) begin
    start_v = 0;
    start_h = 0;
    start_s = 0;
    start_w = 0;
    next = 0;
    case (state)
      0: begin
        start_w = 1;
        next = busy_w ? 7 : 0;
      end

      7: begin
        next = busy_w ? 7 : 1;
      end

      1: begin // assert start until v starts
        start_v = 1;
        next = busy_v ? 2 : 1;
      end

      2: begin //wait until v finishes
        next = busy_v ? 2 : 3;
      end

      3: begin //assert start until h starts
        start_h = 1;
        next = busy_h ? 4 : 3;
      end

      4: begin //wait until h finished
        next = busy_h ? 4 : 5;
      end

      5: begin
        start_s = 1;
        next = busy_s ? 6 : 5;
      end

      6: begin
        next = 6;
      end

    endcase
  end
endmodule
```

```
// Stage 2 : Vincent Wu
module ramwrite(clk,start,busy,reset,x_add,y_add,lum_data, sample_we);

  input clk, start, reset;
  output [7:0] lum_data; // data being stored to into ram

  output [7:0] x_add; //column number
  output [8:0] y_add; // row number
  output sample_we,busy; //ram ctrl & major fsm signals

  reg [7:0] x_add;
  reg [8:0] y_add;

  reg [2:0] state;
  reg [7:0] lum_data;
  reg sample_we,busy;
  wire busy_w;

  always @ (posedge clk) begin
    lum_data <= 8'b10100000;
    sample_we <= 1;
    busy <= busy_w;
    if (reset) state <= 0;

    else case (state)
      0: begin
        state <= start ? 1: 0;
      end

      1: begin
        lum_data <= 8'b10100000;
        sample_we <=0;
        state <= 2;
      end

      2: begin
        sample_we <=0;
        state <= 3;
        end

      3 : begin
        sample_we <= 1;
        y_add <= y_add + 1;
        state <= (y_add == 511) ? 4: 1;
      end

      4: begin
        sample_we <= 1;
        x_add <= x_add + 1;
        y_add <= 0;
        state <= (x_add == 255) ? 0 : 1;
      end
   endcase //end case
  end //end always

assign busy_w = ~(state==0);
endmodule
```

```
// Stage 3 - Sue Zheng
// uses internal rams for foreground and background to test the output-
// positioning of video2 within video 1

module topfile (clk, reset, right, down, go_right, go_down, hsync, vsync, rgb, read);

input clk, reset, right, down, go_right, go_down;
output hsync, vsync, read;
output [7:4] rgb;

wire [15:0] user_center, ramaddress2;
wire [7:0] address1, data1, data2, ramdata2, ramdata1, ramaddress1, address;
wire inch, incv, reseth, resetv, done, we;
wire incv2, inch2, resetv2, reseth2, done2, we2;
wire hinc, hreset, vinc, vreset, usingthebus;
wire[8:0] ramaddress3;

// user input module which determines where the user wants the center
centerv2 c0(clk, reset, right, down, go_right, go_down, user_center);

// backendv3 contains the fsm which decides the rgb output based on position calculations
backendv3 back0(clk, reset, read, user_center, address1, ramdata1, ramdata2,
hsync, vsync, rgb, on_first_line, inch, incv, reseth, resetv, usingthebus);

// addr_counter is a module created by edward which converts the address control signals
// that i send into 16 bit address output for video2
addr_counter a0(clk, reset, hinc, hreset, vinc, vreset, ramaddress2); // 16 bit output

// 8 bit addr. 1 vert, 7 horizontal.  background video. only two lines
ram v1(ramaddress1, we, data1, ramdata1);
// testramv7 is a module which writes data to the background ram (ram v1)
testramv7 t0 (clk, reset, address, data1, done, we);

// 9 bit addr.  4 vertical, 5 horizontal. foreground video
biggerram v2 (ramaddress3, we2, data2, ramdata2);
// testbiggerram is a module which writes data to the foreground ram (biggerram v2)
testbiggerram t1 (clk, reset, incv2, inch2, resetv2, reseth2, data2, done2, we2);

// adjust ram address into bigger ram (foreground image) since the internal
// ram has 9 address bits instead of 16
// if outside range of data, sent to addr 9'b111111111 because it has blanking on
assign ramaddress3 = ((ramaddress2[15:12] > 4'b0000) | (ramaddress2[7:5] > 3'b000)) ? 9'b111111111
: {ramaddress2[11:8], ramaddress2[4:0]};
// hand over control of addr counter to appropriate module
assign {hinc, hreset, vinc, vreset} = (done2) ? {inch && usingthebus, reseth && usingthebus, incv
&& usingthebus, resetv && usingthebus} : {inch2, reseth2, incv2, resetv2};
// hand over control of ram address for background video ram to appropriate module
assign ramaddress1 = (done) ? address1 : address;
// control signal for backendv3 fsm
assign read = done & done2;

endmodule
```

```
// Stage 3 - Sue Zheng

//used to write to background ram
module testramv7 (clk, reset, address, data, done, we);

input clk, reset;
output [8:0] address;
output [7:0] data;
output done, we;


reg vcount;
reg [7:0] hcount;
reg done;
reg we;
reg[1:0] state;
always @ (posedge clk)
begin

    if (reset)
        begin
        {done, vcount, hcount} <= 0;
        end

        else case (state)
        0:  begin
// check if at end of the line
// that's when hcount = 127 because only 128 addresses (256 pixel = 128 addr with 2 pixels/addr)
                if (hcount == 127)
                    begin
// if done with line, is it also done with the second line?
                        if (vcount == 1)
// if so, then done.
                            begin
                            we <= 0;
                            done <= 1;
                            end
// otherwise, write to second line
                        else begin
                                    hcount <= 0;
                                    vcount <= vcount + 1;
                                    state <= state + 1;
                                end
                    end
// or if not at end of line, just increment to next addr
                else begin
                hcount <= hcount + 1;
                state <= state + 1;
                    end

            end
// addr has been set up. bring we high
        1: begin state <= state + 1;
            we <= 1;
            end
// bring it back down
        2: begin
            state <= 0;
            we <= 0;
            end

        endcase
end
// cycle through the addresses. vcount -> 2 lines.  hcount[6:0] -> 128 pixels
assign address = {vcount, hcount[6:0]};
// data assigned will be color bars (dark gradient because hcount7 always = 1)
assign data = {hcount[7:4], hcount[7:4]};
//assign data = {vcount, vcount, vcount, vcount, vcount, vcount, vcount, vcount};
```

```
endmodule
```

```
// Stage 3 - Sue Zheng

// writes data to the foreground ram to confirm outputting onto screen
module testbiggerram (clk, reset, incv2, inch2, resetv2, reseth2, data2, done2, we2);

input clk, reset;
output incv2, inch2, resetv2, reseth2;
output [7:0] data2;
output done2, we2;

reg [3:0] vcount;
reg [4:0] hcount;
reg done2;
reg we2, incv2, inch2, resetv2, reseth2;
reg[2:0] state;
always @ (posedge clk)
begin

    if (reset)
        begin
        {done2, vcount, hcount} <= 0;
        end
        else case (state)
        0: begin // initialize state - go straight to write!
            state <= 3;
            we2 <= 1;
            end
        1:  begin // check if done with a row or all rows
                if (hcount == 31)
                    begin
                        if (vcount == 15) // done all lines
                            begin
                            we2 <= 0;
                            done2 <= 1;
                            end
                        else begin // done with line. not done with ram. go to next line.
                                    hcount <= 0;
                                    vcount <= vcount + 1;
                                    state <= state + 1;
                                    reseth2 <= 1;
                                    incv2 <= 1;
                                    inch2 <= 0;
                                    resetv2 <= 0;
                                    we2 <=0;
                                end
                    end
                else begin // if not done with line.  go to next pixel
                hcount <= hcount + 1;
                reseth2 <= 0;
                incv2 <= 0;
                inch2 <= 1;
                resetv2 <= 0;
                state <= state + 1;
                we2 <= 0;
                    end

            end
        2: begin state <= state + 1; // wait state
            we2 <= 0;
            reseth2 <= 0;
            incv2 <= 0;
            inch2 <= 0;
            resetv2 <= 0;

            end
        3: begin  // write to ram
            state <= 1;
            we2 <= 1;
```

```
            end

        endcase
end

// total size of video 2 is 32 x 16  (horizontal x vertical)
// make a rectangle of size 16 x 15 with color 1000
// outside of square is color 1111 (blanking signal)
assign data2 = (hcount > 15 | vcount > 14) ? 8'b11111111 : 8'b10001000;

endmodule
```

```
// Stage 3 - Sue Zheng

// Used to keep track of where the user wants the center
module centerv2(clk, reset, right, down, go_right, go_down, user_center);

input clk, reset, right, down, go_right, go_down;
output[15:0] user_center;
wire go_right_sync, go_down_sync;
button2 b0(clk, go_right, go_right_sync);
button2 b1 (clk, go_down, go_down_sync);

reg [19:0] countdown;
reg[19:0] countright;
reg [15:0] user_center;

always @ (posedge clk)

    if (reset)
        begin
            countdown <= 524287;
            countright <= 524287;
// default user center at lower right corner of screen
// with the assumption that picture is 1/16th of total screen size
            user_center <= 16'b1101000111011111;
        end
    else
            begin
                case (countdown)
                        // if counted from 524287 to 0, then decrement the vcount in user_center
                        // but if already at top of screen - can't move anymore up
                        // and reset the count
                0:      begin
                            user_center[15:8] <= (user_center[15:8] == 8'b00000000) ? 0 :
user_center[15:8] - 1;
                            countdown <= 524287;
                        end
                        // if counted from 524287 to 1048575, then increment the vcount in
user_center
                1048575:    begin
                            user_center[15:8] <= (user_center[15:8] == 239) ? 239 :
user_center[15:8] + 1;
                            countdown <= 524287;
                        end
                        // otherwise, just increment/decrement counter
                default: countdown <= (go_down_sync) ? ((down) ? countdown + 1 : countdown - 1) :
countdown;
                endcase

                case (countright)
                        // if counted from 524287 to 0, then decrement the hcount in user_center
                        // but if already at left side of screen - can't move anymore left
                        // and reset the count
                0:      begin
                            user_center[7:0] <= (user_center[7:0] == 8'b00000000) ? 0 :
user_center[7:0] - 1;
                            countright <= 524287;
                        end
                        // if counted from 524287 to 1048575, then increment the hcount in
user_center
                1048575:    begin
                            user_center[7:0] <= (user_center[7:0] == 255) ? 255 : user_center[7:0]
+ 1;
                            countright <= 524287;
                        end
                        // otherwise, just increment/decrement counter
                default: countright <= (go_right_sync) ? ((right) ? countright + 1 : countright -
1) : countright;
                endcase
```

```
            end
endmodule
```

```
// Stage 3 - Sue Zheng

// fsm which controls the addressing to foreground and background video rams
// based on positioning calculation (for foreground)

module backendv3(clk, reset, read, user_center, address1, data1, data2,
hsync, vsync, rgb, on_first_line, inch, incv, reseth, resetv, usingthebus);

input clk, reset, read;
input [15:0] user_center; // center comes from vincent = size of image
input [7:0] data1, data2; // data from v1, data from v2


output hsync, vsync, on_first_line;
output [7:0] address1; // saving only a couple of lines to memory - only 8 bit addr
output [7:4] rgb;
output inch, incv, reseth, resetv, usingthebus;

reg [7:0] address1;
reg [2:0] state;

reg hsync, vsync, hblank, vblank,
    startv2, abouttostartv2,
    outside_screen_left, outside_screen_right,
    outside_screen_top, outside_screen_bottom,
    usingthebus;

// 4 bit output to DAC then to screen
reg [7:4] rgb;
// so we don't need to access again. receive two pixels at once
reg [3:0] stored_data;
// because choice of center ranges from [0, 255] x [0, 239]
reg [7:0] h2start, v2start;
// ** be careful! user_center only ranges from [0, 128] x [0, 239] ** //

// used to address v2, b/c otherwise too many bits to transfer between us
// and because i read (mostly) sequentially
reg inch, incv, reseth, resetv;

wire [15:0] center;
// size of foreground image
assign center = 16'b0011101100111111;


//////////////////////////
// ** sync generator ** //
//////////////////////////

// pixel number on current line (out of 634 because 20mhz clk, although we repeat)
reg [9:0] hcount;
// line number (out of 480, although we repeat)
reg [9:0] vcount;

  // horizontal = 31.76us
  // display 512 pixels per line (256 unique)
  wire hsyncon,hsyncoff,hreset,hblankon;
  assign hblankon = (hcount == 511);
  assign hsyncon = (hcount == 518);
  assign hsyncoff = (hcount == 594);
  assign hreset = (hcount == 635);

  // vertical = 16.77us
  // display 480 lines (240 unique)
  wire vsyncon, vsyncoff, vreset, vblankon;
  assign vblankon = hreset & (vcount == 479);
  assign vsyncon = hreset & (vcount == 492);
  assign vsyncoff = hreset & (vcount == 494);
  assign vreset = hreset & (vcount == 527);
```

```
always @ (posedge clk)

begin

    if (reset)
        begin
            state <= 0;
            hcount <= 0;
            vcount <= 0;
            stored_data <= 0;
            reseth <= 1;
            resetv <= 1;
            inch <= 0;
            incv <= 0;
            usingthebus <= 0;
        end
    else
        begin

// ** calculation of whether foreground image will be outside the screen ** //

            outside_screen_right <= ((center[7:0] - {center[7:1]}) > (255 - user_center[7:0]));
            outside_screen_left <= (center[7:1] > user_center[7:0]);
            outside_screen_top <= (center[15:9] > user_center[15:8]);
            outside_screen_bottom <= ((center[15:8] - center[15:9]) > (239 - user_center[15:8]));

// ** calculation of where foreground image starts ** //

// on every new field, update the row to start reading v2
            v2start <= (vreset|reset) ? (outside_screen_top ? 0 :
                        (outside_screen_bottom ? 239 - center[15:8] :
                                user_center[15:8] - center[15:9])) : v2start;

// on every new field, update the h-pixel to start reading v2
            h2start <= (vreset|reset) ? (outside_screen_left ? 0 :
                        (outside_screen_right ? 255 - center[7:0] :
                                user_center[7:0] - center[7:1])) :
                                        h2start;

// high when foreground image at next pixel
    abouttostartv2 <= ((hcount == 634) && (h2start == 0) && (vcount == 527) && (v2start == 0)) |
                     ((hcount == 634) && (h2start == 0) && ((vcount[8:1] +1) >= v2start[7:0])) |
                     (((hcount[8:1] + 1) >= h2start[7:0]) && (vcount[8:1] >= v2start[7:0]));

// high when foreground image at current pixel
        startv2 <= abouttostartv2;


////////////////////////////////////////
// ** sync generator / pixel counter ** //
////////////////////////////////////////

            hcount <= (state == 0 | state == 1 | hreset) ? 0 : hcount + 1;
            hblank <= hreset ? 0 : hblankon ? 1 : hblank;
            hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;

            vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
            vblank <= vreset ? 0 : vblankon ? 1 : vblank;
            vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;

///////////////
// ** FSM ** //
///////////////

        case (state)
                    // initialize state. wait for read to sync up
```

```
            3'b000: if (read)  // initialize/synchronize state
                        state <= state + 1; // can read!  enable the bus~!
                    else
                        begin
                        address1 <= 0;
                        inch <= 0;
                        incv <= 0;
                        resetv <= 1;
                        reseth <= 1;
                        usingthebus <= 1;
                        end
            3'b001:     // another state which is part of synchronization
                        state <= state + 1; // wait for data

            3'b010: begin // latches data from ram. not reading
                        state <= state + 1;
                        stored_data <= (~abouttostartv2 || (data2[3:0] == 4'b1111)) ?
                                        data1[3:0] :
                                        data2[3:0];
                        rgb <= (vblank | (hblank & ~hreset)) ?
                                0 :
                                (~abouttostartv2 || (data2[7:4] == 4'b1111)) ?
                                    data1[7:4] :
                                    data2[7:4];
                        inch <= (startv2 && (hcount < 512));
                        incv <= (startv2 && (hcount == 512) && (vcount[0] == 1));
                        reseth <= (startv2 && (hcount == 512));
                        resetv <= (vblank);
                        address1 <= (vblank | (hblank & ~hreset)) ?
                                    {vcount[1], 7'b0000000} :
                                    address1 + 1;
                        usingthebus <= 0;
                    end
            3'b011: state <= state + 1; // not reading
            3'b100: // held current pixel for 80 ns. time to switch pixels. reading at this state
                    begin
                        state <= state + 1;
                        rgb <= (vblank | (hblank & ~hreset)) ? 0 : stored_data[3:0];
                        usingthebus <= 1;
                    end
            3'b101: begin // stop incrementing the address! second cycle of reading
                        state <= 3'b010;
                        inch <= 0;
                        incv <= 0;
                        reseth <= 0;
                        resetv <=0;

                    end

            endcase

        end
end

// used so that the middle stage doesn't write foreground
// while third stage is reading the first line
assign on_first_line = (vcount[8:1] == v2start);

endmodule
```