Design and Implementation of a Video Game System: Rodent Revenge in Space

Naoshin Haque
Matthew Kwan
Lynne Salameh

6.111: Introductory Digital Systems Laboratory
December 9, 2004

Abstract: Rodent Revenge in Space is a video game, featuring Tim the beaver. Tim moves around the virtual world, where he encounters various enemies and obstacles. When facing an enemy, Tim throws logs at him. Tim will lose a life if an enemy injures him, but can gain lives by picking up powerups. The three modules in the game are the game controller unit, video controller unit, and the resizer. The game controller interfaces with the user and coordinates with the video controller unit to advance the game accordingly. The video controller unit outputs the images onto the screen and interfaces with the resizer in order to output the correct enemy size frame onto the screen. The resizer uses interpolation and decimation to shrink the enemy frame into different sizes. In conclusion, the project was a satisfying and extremely informative experience.

Table of Contents

List of Figures

List of Tables

1 Video Game System Overview

This project will consist of a video game named Rodent Revenge in Space. The main character, Tim the beaver, is lost in outer space. Tim is trying to find the wormhole, which will lead him back to MIT. Along the way, many aliens will try to stop Tim from completing his journey, because they want to use his intelligence to take over the universe. Tim can only counter the aliens by throwing deadly wooden stakes at them. Initially, Tim has three lives, but if he is injured by an alien, he loses a life. However, if Tim has less than three lives and he collects enough energy drinks, he can regain the lives he lost.

The game will be configured so that the user's perspective coincides with Tim's perspective. The user will have access to six controls to manipulate the game:

1. Start button: initiates the game. If the user has pressed quit once, then pressing start would cancel the quit.
2. Up button: moves Tim forward through space.
3. Left Button: moves Tim sideways left across the screen
4. Right Button: moves Tim sideways right across the screen
5. Shoot Button: Triggers Tim to shoot a stake from where he is located on the screen

The screen display will be divided into several frames, which will each have x, y, and z coordinates. The x and y coordinates denote the horizontal and vertical position of the frame on the screen, respectively. The z coordinate determines which frame will take precedence on the screen. The frames will be of the following description:

1. Tim's frame: A frame containing a back view of Tim's head. This frame will be positioned at the bottom of the screen, and will have the highest z value, meaning it will take the most precedence on screen.
2. Background frame: This frame is static, has the lowest z value, and consists of a space backdrop.
3. Road frames: There will only be one road frame on screen at a time, depending upon Tim's forward motion. Each road frame will lie centrally in the screen and will have a higher z value than the background frame.
4. Enemy frames: These frames will move along the road frame and their size depends on the enemy's distance from Tim. Their z value will be higher than the previous frames.
5. Log/bullet frames: The log/bullet frames will be moving along the road frame. These frames will have the second highest z value.
6. Energy frame: This frame will be moving with the road frame, but it will have a higher z value than the road frame.
7. Score/Lives frames: These frames will be at the top of the screen and will have a z value higher than that of the background frame.

The game will be subdivided into three sections of roughly the same complexity: a game controller (Lynne), a decimator (Naoshin) and a video display module (Matt). A general block diagram for the system is shown below:

Game Controller Unit:

The game controller module handles the game's logistics. The game controller is in fact an FSM which has various states depending on the inputs. The game controller's

main job is to decide which frames are going to be used in the video display at each time-slice. The time-slice's value will be chosen to simulate the game smoothly. The game controller will have an algorithm which decides when the alien will come on screen, and its trajectory towards Tim. The information associated with the alien will take form of output *size* which will be passed on to the Decimator. *Size* denotes how big the alien frame should be on screen.

Depending on the input of the user, the game controller will move Tim accordingly. The controller will keep track of the score and of Tim's lives. In addition, the game controller handles the timing and trajectory of the stakes and alien's photon bullets. Therefore, the game controller will output each frame's x, y, and z coordinates to the frame handler. The game controller will also output to the frame handler the frames which will be used in the next scene. In addition, the game controller will receive inputs from the frame handler indicating whether a bullet frame overlaps with the beaver frame, or if the log frame overlaps with the enemy frame. The game controller will then either, respectively, decrement the beaver's life or kill the enemy and increase the score of the beaver. After the beaver loses all lives, the game controller would signal the game is over.

Resizer:

The resizer changes the size of the enemy frames. The resizer receives input from the game controller denoting the size of the frame to be passed on to the frame handler. The resizer has access to a fixed-sized image which will then filter and downsamples at a rate specified by the game controller. The sampled image will be stored in the resizer's

ROM. It outputs an address range denoting where it will store the picture to the frame handler.

Video Controller Unit:

The Video Controller is divided into several submodules: Frame Handler, Sync generator, video memory, and the DAC.

Frame Handler: The frame handler receives an output from the Sync generator denoting the coordinates of the next pixel to be drawn. The frame handler will then check each frame to see if it contains the indicated pixel. If the pixel is contained in more than one frame, the handler will check the z values of those frames to see which frame has precedence. It will then output the color value of the pixel in that frame to the screen.

Sync generator: The sync generator controls the how the fast the electron beam sweeps across the screen. There are two sync signals: vertical and horizontal sweep. It outputs to the frame handler the coordinates of the next pixel to be drawn. The video memory stores the frames and outputs them to the frame handler, and the DAC converts the pixels from digital to analog.

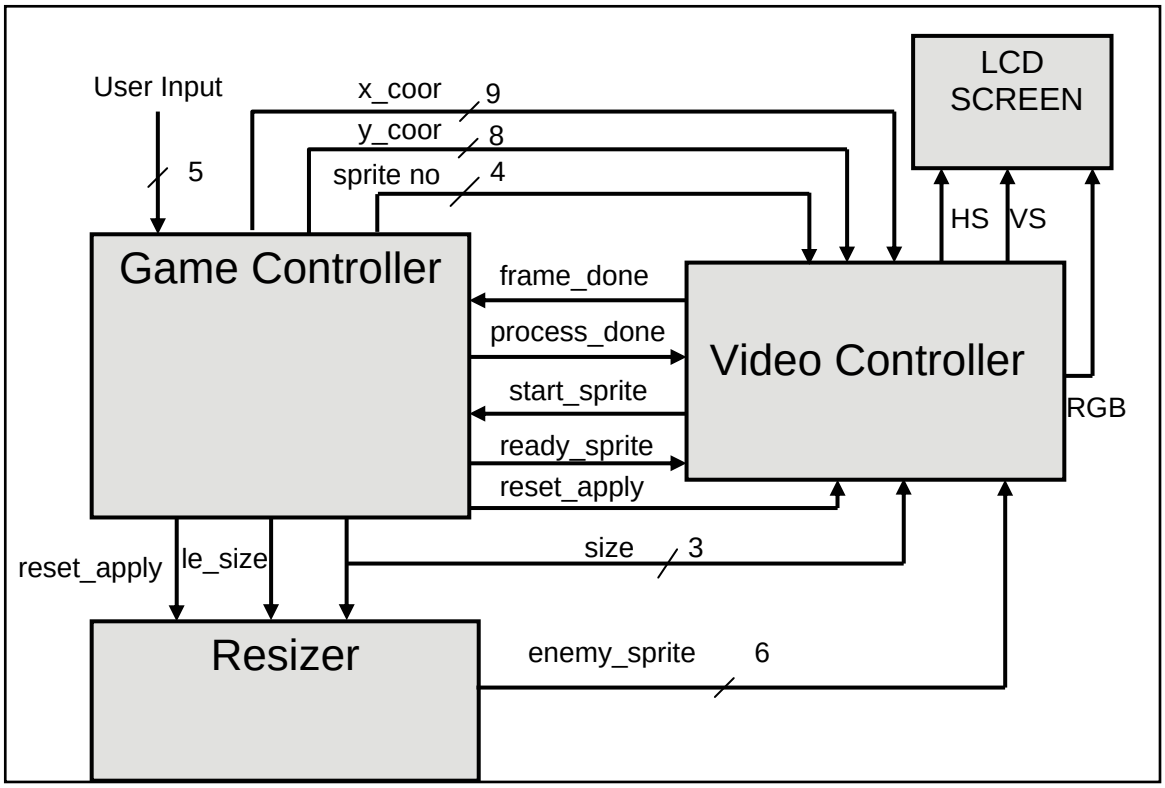


Figure 1: Overall Block Diagram for Video Game System

2 Subsystem 1: Game Controller Unit (by Lynne Salameh)

The game controller handles the user input and the progress of the game. It is in a sense a simple microprocessor that is programmed with instructions that simulate the game by choosing which sprites are going to be present on screen next. This functionality is captured by the sprite's *sprite_no*, an index needed to differentiate between the different sprites, and its location on screen given by *xcoor* and *ycoor*. The information about the sprites is transmitted to the Video Controller as soon as it is ready. In addition, the controller determines the size of the alien sprite which needs to be passed on to both the Resizer and the Video Controller. The game controller is subdivided into three main submodules: PC module, Instruction Decoder module and the Register FSM module. A 512 x 20 built in MegaWizard ROM was used to store the instructions, and three 32 x 8 MegaWizard RAM's were used as the three register files needed to store sprite_no, x_coor and y_coor. The game controller receives several control signals from the Video Controller which allow it to switch between processing mode and outputting the sprites, and these signals are *frame_done*, triggering the start of the processing cycle, and a *start_sprite* signal, triggering the sprite output cycle. A block diagram of the over all Game Controller Module can be seen in Figure 2.1. on the following page.

Java was used to write a compiler, which would take in an instruction and transform it into a 20 bit binary line, which was written to a .mif file.

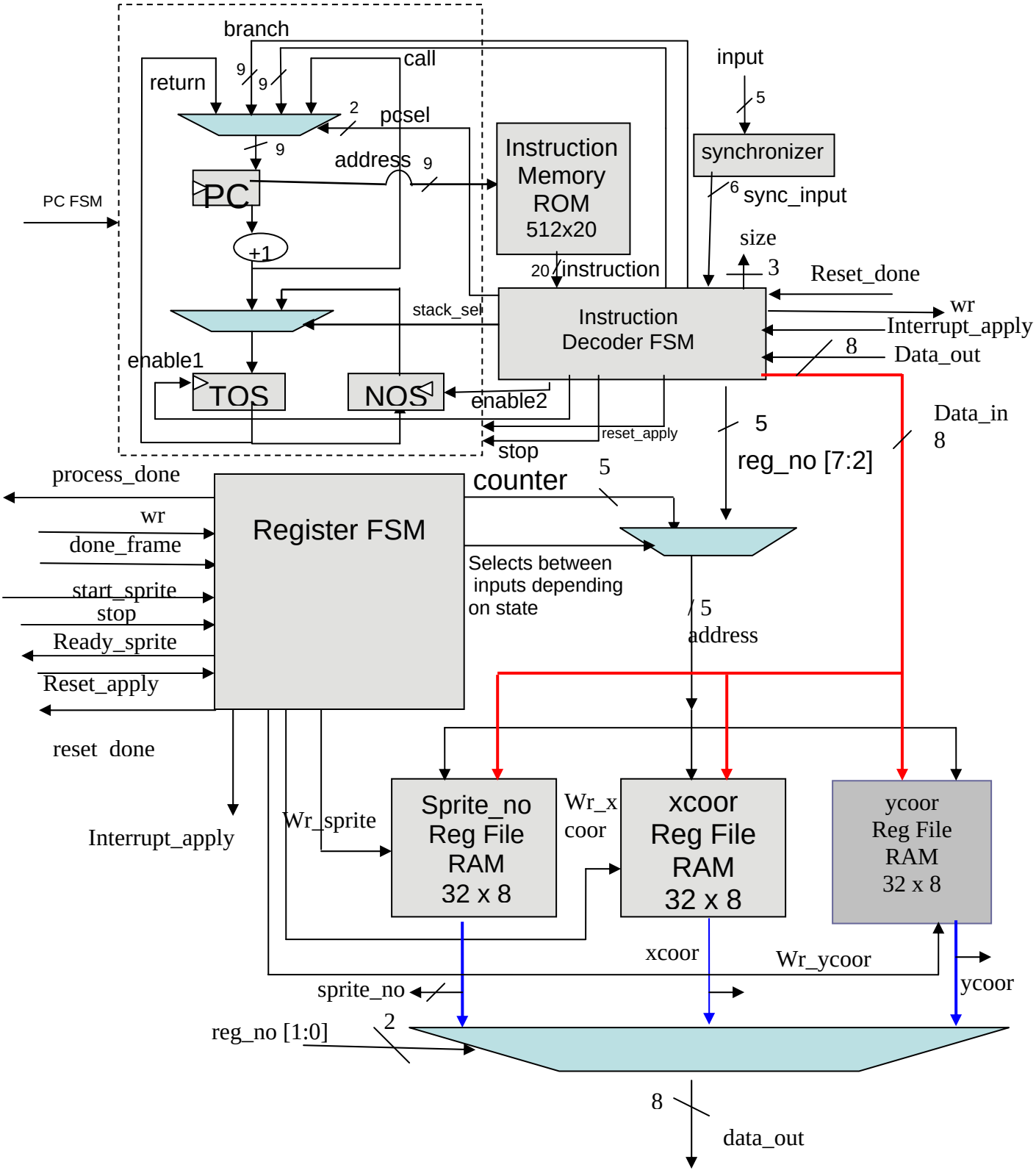


Figure 2.1: Game Controller Block Diagram

2.1 PC Module

The PC module is in fact a simple FSM that controls the addressing of the instruction memory. The address is 9 bits, and is stored in a register called PC, or program counter. The PC determines which instruction will be read next. The PC module receives several control signals from the Instruction Decoder, in addition to the *frame_done* signal from the Video Controller. The *frame_done* signal indicates the begging of the processing cycle, and moves the FSM from state_idle to state_busy, as can be seen in the State Transition Diagram in figure 2.2. In state_busy, the PC FSM takes into consideration several control signals : *pcsel*, *enable* and *stack_sel* from the Instruction Decoder. The *pcsel* signal determines what value the PC register will assume next and therefore which line to be read from the Instruction Memory. The PC module also implements a two element stack, in the form of the registers TOS and NOS. These are loaded when they receive the *enable* signal from the Instruction Decoder.

There are five different values for *pcsel*, which are:

1. *pcsel = 0* : The program is advancing normally, and therefore the next instruction to be read is the current instruction + 1, i.e. PC + 1.
2. *pcsel = 1*: This control signal causes the PC to branch to a new value, and therefore loads the value of the *branch* from the Instruction Decoder into the PC register. This in effect causes the address of the instruction memory to jump from on value to the other.
3. *pcsel = 2* : Facilitates calls to instruction lines. A call causes the PC to jump to a new value, while at the same time storing the current value + 1 into a stack. The

signal *call* from the Instruction Decoder determines the new address, and is loaded into the PC register.

4. *pcsel = 3*: Triggers a return, that is, the value of TOS is loaded into the PC register, and therefore the instruction that is read returns to the instruction after when the most recent call was made.
5. *pcsel = 4*: Maintains the current PC and does not change it. This is needed during the intermediate states of the Instruction Decoder which will be discussed later.

As for *enable*, it is three bit, and it can take 3 meaningful values:

1. *enable[0]* = 1: Loads PC +1 into the top of the stack TOS.
2. *enable[1]* = 1: Loads the value of TOS into NOS the rest of the stack.
3. *enable[3]* = 1 : Loads the current PC into TOS, which is needed when an interrupt occurs.

Similarly *stack_sel* can take two values:

1. *stack_sel = 0*: Allows PC +1 to be loaded into TOS.
2. *stack_sel = 1*: Allows the value of NOS to be loaded into TOS.

The control signals described enable PC + 1 to be pushed onto the top of the stack, TOS, on a call. In other words *stack_sel = 0* and *enable* = 3 causes the TOS to be loaded with PC + 1 and NOS would take the value of the old TOS. On the other hand, *stack_sel* is set of 1 on a return, which, combined with setting *pcsel* to 3 and *enable* to 1 pushes the return address of the TOS and loads it into the PC. When *enable[2]* = 1, indicating that an interrupt has occurred, the TOS will be loaded with PC rather than PC + 1, NOS is loaded with the old value of TOS and therefore an interrupt functions in a similar way to call, except for that PC is loaded into TOS.

During the busy state, a *stop* signal from the Instruction Decoder would cause the PC to revert back into its idle state, and await from a new *frame_done* signal from the Video Controller. A *reset_apply* from the Instructon Decoder causes the PC FSM to revert to the reset state, from which it would go to the busy state on receiving a *frame_done* signal from the Register FSM.

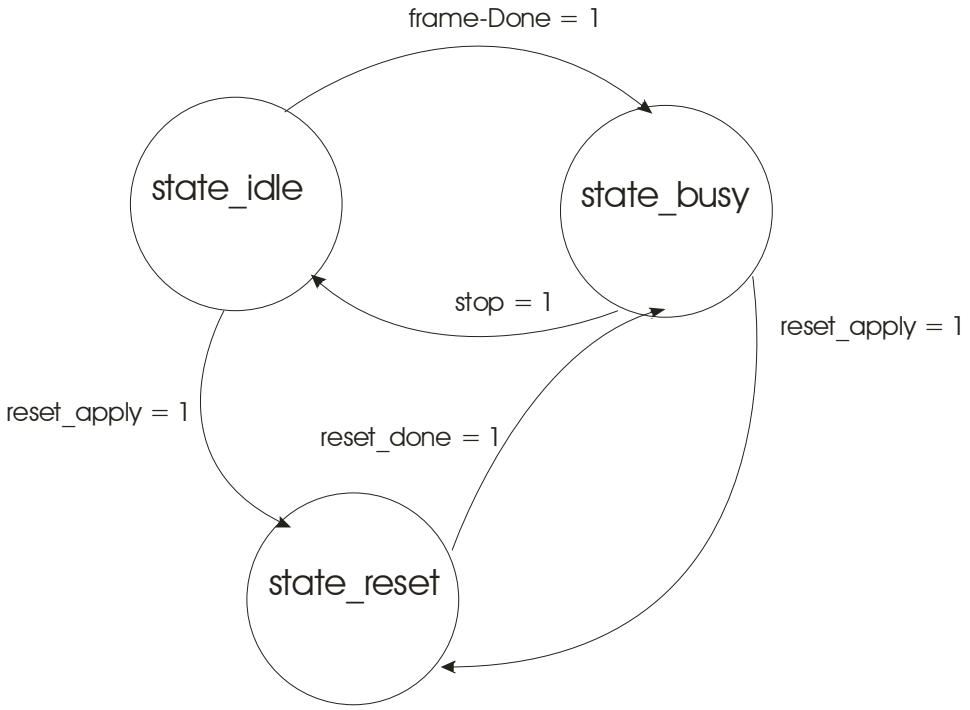


Figure 2.2: State Transition Diagram for PC FSM

2.2 Instruction Decoder

The Instruction Decoder processes the 5 user inputs that signify forward, right, left, shoot and reset. This 5 bit input in the form of *user_in* is first synchronized to match the 12 MHz clock the Game Controller runs on. The Game Controller also takes the 20 bit instruction outputted from the Instruction Memory, and determines which action to take next depending on the 4 bit operation code at the beginning of the instruction. The

action is in the form of several control signals to be sent to the other modules. The rest of the 20 bit instruction contains relative information that will be used by the decoder to complete the operations described by the opcode, such as the address to be read or written to in the Register RAMS (*reg_no [7:0]*), the register which is to be read or written to (i.e. whether it is sprite_no, xcoor or ycoor, handled by the signal *reg_no[1:0])*, call and branch addresses, which input to look at, and the internal register number for which results of comparisons are stored. The Instruction Decoder reserves four internal registers that will store the values of comparisons and these are set by operations such as COMPEQ, COMPLE, COMPEQC, and COMPLEC. The signals *branch* and *call* are always set to be equal to *instruction[8:0]*. Table 1 summarizes up all the opcodes and the information captured by the remaining 16 bits of the instruction.

Table 1: Opcodes and their Description

Opcode	Instruction	Description
0000	ASSIGN	Assigns a value equal to *instruction[7:0]* to the register memory line with address *reg_no[15:9]*
0001	ADDC	Reads in the value of the register memory line with address *instruction[15:9]* and writes back onto that address the value of the memory line plus *instruction[7:0]*
0010	BRANCHCOND	Compares the synchronized input *control* to *instruction[15:11]* and causes a branch if they are equal
0011	COMPEQ	Compares the value of the register file at address *reg_no = instruction[15:9]* with the value of the register file at address *reg_no= instruction[8:2]* and sets the internal registers specified by *instruction[1:0]* to be 1 if the values are equal, 0 otherwise
0100	BEQ	Causes a branch to *instruction[8:0]* if the internal register specified by *instruction[15:14]* equals 0
0101	CALL	Causes a call to *instruction[8:0]*
0110	RETURN	Causes a return
0111	STOP	Makes the stop signal that is an input to the PC and Register File modules go high for 1 clock cycle
1000	SIZE	Sets *size* to be equal to *instruction[2:0]*
1001	COMPEQC	Compares the value of the register file at address *reg_no = instruction[15:9]* with a literal of value *instruction[8:2]* and sets the internal registers specified by *instruction[1:0]* to be 1 if the values are equal, 0 otherwise
1010	COMPLE	Compares the value of the register file at address *reg_no = instruction[15:9]* with the value of the register file at address *reg_no= instruction[8:2]* and sets the internal registers specified by *instruction[1:0]* to 1 if the former value is less than or equal to the later.
1011	COMPLEC	Compares the value of the register file at address *reg_no = instruction[15:9]* with the literal of value *instruction[8:2]* and sets the internal registers specified by *instruction[1:0]* to 1 if the former value is less than or equal to the later.
1100	AND	Computes the value of the internal registers specified by *instruction[15:14]* AND the value of the internal register specified by *instruction[13:12]*, placing the result in the internal register specified by *instruction[11:10]*
1101	OR	Computes the value of the internal registers specified by *instruction[15:14]* AND the value of the internal register specified by *instruction[13:12]*, placing the result in the internal register specified by *instruction[11:10]*
1110	EQIN	Sets the internal register specified by *instruction[12:11]* to 1 if *control* specified by *instruction[15:13]* is equal to *instruction[10]*
1111	BNE	Causes a branch to *instruction[8:0]* if the internal register specified by *instruction[15:14]* is not equal to 0

Since the register file memories can only be read and written to at separate times, several states were constructed in order to allow for this functionality, as shown in Figure 2.3, which is a state transition diagram of the decoder. The instruction decoder remains in the idle state till it receives a *frame_done* signal from the Video Controller, indicating that it needs to transition to state_op, where the begging of the processing occurs. In state op, the opcodes are used to determine which state will be chosen next. In cases where there will be no accesses to memory, i.e. for operations such as COMPEQ and CALL, the control signals are set to the correct values, and the FSM transitions to state_incpc, in which control signals are set to increment the PC in the PC module according to the opcode. Other operations such as ASSIGN and ADDC, require writing onto the memory, in which case, whereas operations such as COMPEQ require reading from two memory locations. In order to facilitate these operations, several read states for the two reads required were added to the FSM, a write state where *wr* would be high for one clock cycle, and a process state acts as an intermediate and chooses to continue onto the write state or the state_incpc. Assigning the *xcoor* and *ycoor* memory locations to a non-zero value when the corresponding *sprite_no* location is 0 has no effect on displaying the sprites, since the Video Controller ignores the sprites with sprite number 0. Therefore assigning these registers to certain values was used when writing the game code in order to use several local variables.

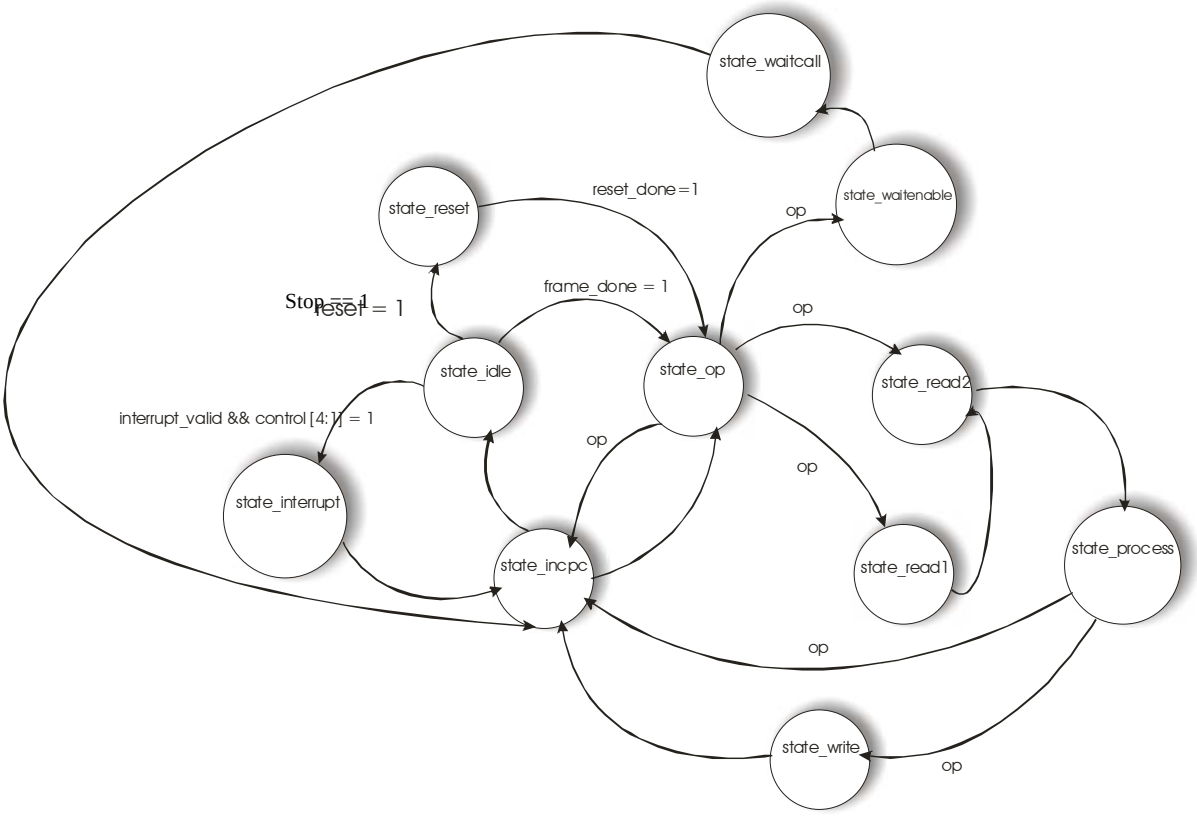


Figure 2.3: state transition diagram for the Instruction Decoder. All states lead into the reset state when *reset_apply* is high. Similarly, all states lead into state_interrupt when *interrupt_valid* and *control[4:1]* are high

The call operation requires the enable signal for NOS to go high before the enable signal for the TOS, at the same time keeping the PC constant for an extra clock cycle to avoid timing issues. Adding two extra states wait_enable and wair_call allows for this behavior to occur. These states are only accessed during a call procedure.

On reaching state_inpc, a stop operation from the Instruction Memory would cause the FSM to revert back to state_idle, and wait for the next *frame_done* signal. Otherwise, the state_incpc would lead back into state_op on the next clock cycle, freezing the pc by setting *pcsel* to 4. Two deviations may occur from this standard flow:

1. A reset is pressed, and therefore *reset_apply* is high, which triggers all the states to revert to state_reset. In state_reset, the PC is frozen by selecting *pcsel* = 4 and data_in is set to 0, until a *reset_done* signal is received from the Register File FSM, indicating that the memories have been blanked and that it is possible to proceed to state_op.
2. An interrupt occurs, i.e. the *interrupt_valid* signal from the Register FSM, occurring a clock cycle after the *frame_done* signal, coincides with a non-zero value for *control[4:1]*, in other words, the user has pressed one or more of the buttons. An interrupt can occur in any state, and this causes the FSM to jump to state_interrupt, where control signals are chosen so as to force the PC to jump to location 461 of the memory, the location of the interrupt handler code. The control signals also ensure that the PC is stored in the TOS.

2.3 Register FSM

The Register FSM controls the reading and writing onto the three 32 x 8 RAM's that contain information about the sprite_no, xcoor and ycoor of the sprites to be used in the current frame of the video memory. The Register FSM has two main states, as shown in Figure 2.4, and these are state_process and state_loop. An intermediate stage between these two processes occurs as state_waitforsprite, in which the FSM waits for either a *start_sprite* signal from the Video Controller, causing it to transition to state_loop, or a *frame_done* signal, causing it to return to state_process.

1. **state_process:** in this state the Instruction Decoder drives the address line using *reg_no*, whose lowest 2 bits are used to determine which of the three memories to be written or read from. In this state, the signal *wr* is translated into the separate

write enables for each RAM, depending on the value of *reg_no*[1:0]. Similarly, *data_in* is the value from the Instruction Decoder that is written to one of the memories, and *data_out* is the value read from one of the memories, which is chosen by *reg_no[1:0]*. During state_process, the control signal *process_done* which is outputted to the Video Controller goes low as long as the FSM is in this state, indicating that the Game Controller is processing instructions from the Instruction Memory. As soon as a *stop* signal is received, the FSM would revert to state_waitforsprite.

2. **state_loop:** In this state, and internal counter which counts from 0 to 31 drives the address lines of the three RAM's at the same time. The control signal *ready_sprite* goes high as long as the FSM is in this state. The outputs of the three RAM's, in the form of xcoor, ycoor and sprite_no, are transmitted to the Video Controller. The information is transmitted in 32 cycles, after which the FSM returns to state_waitforsprite.

On receiving a *reset_apply* signal from the Instruction Decoder, the FSM would revert to state_reset, in which the counter drives the RAM's once again, but this time the write enable signals of all three RAM's is set to high, and a zero valued *data_in* is used to blank the RAMs.

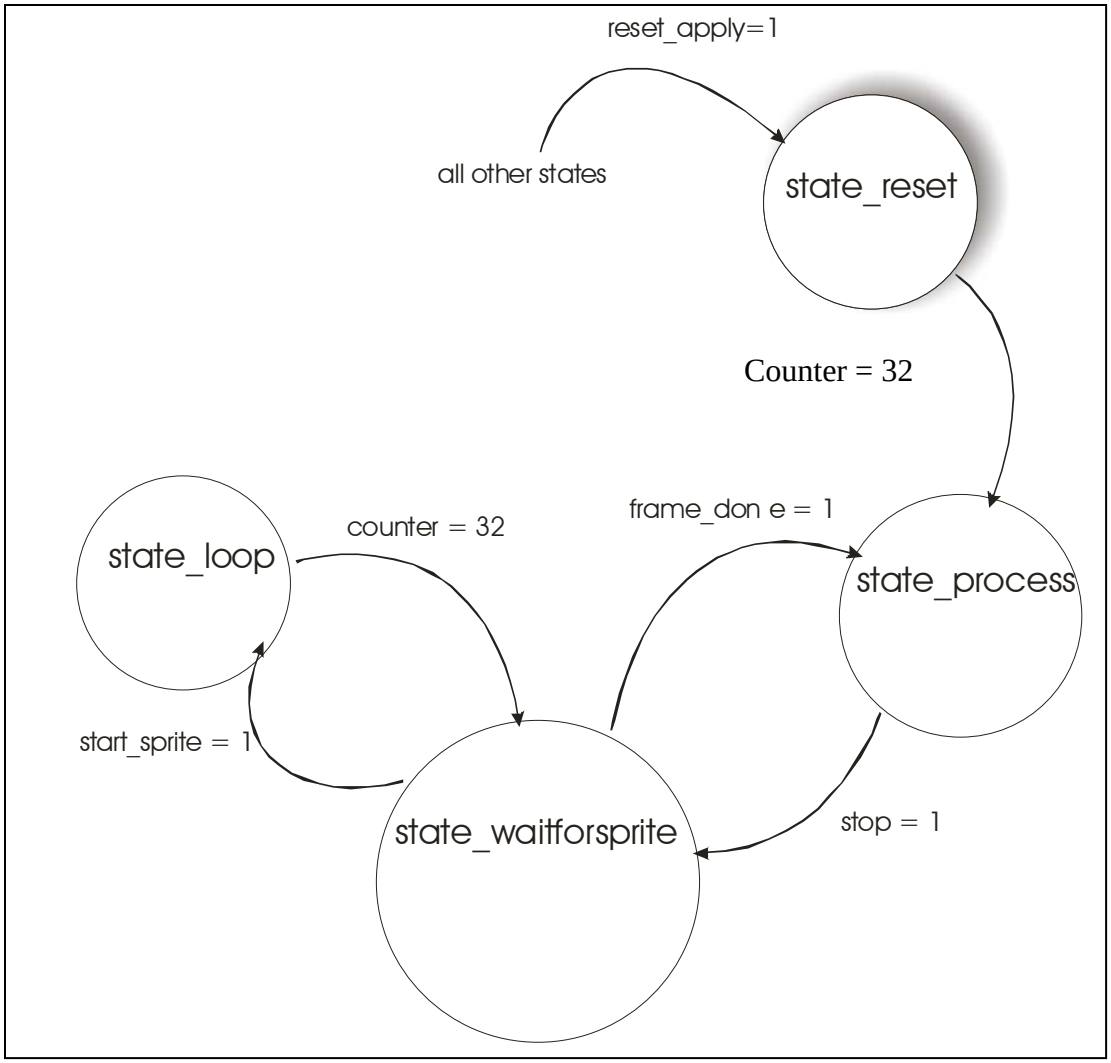


Figure 2.4: State Transition Diagram for Register FSM

2.4 Testing and Debugging

The early stages of debugging incorporated unit testing of each separate module, and used Max II plus' waveform editor to simulate the results of the module. For the Instruction Decoder module, each opcode was tested separately in the waveform editor in or to determine whether it indeed provided the correct functionality. After completing all the smaller modules, integration testing was performed on the top level module,

controller. A simple program was written using the Java compiler, which only assigned several frames and provided an interrupt handler for the forward input. The Controller module was simulated using this program in the Instruction Memory, and the 32 cycle output to the Video Controller was checked to see whether it contains the correct information about the sprites. Wire signals were temporarily converted into outputs in order for them to appear in the waveform editor.

A had added a few more opcodes after I had done testing Instruction Decoder, and regression testing was performed through more simulation of the Instruction Decoder module to double check whether it still provided the correct functionality.

After the top level Display module was constructed, combining the Game Controller module and the Video Controller into one module on a single FPGA, the *vsync* and *hsync* values were changed to smaller ones for the purpose of simulation, and the module was simulated to ensure the correct functionality. The Display module was then loaded onto the FPGA and was tested directly by examining the output to the screen. By examining the screen, several bugs were detected, and to help with the debugging process, the 9 bit *pc* was converted to an output and connected to the logic analyzer, which was triggered on the *frame_done* signal. An illegal loop which occurred when applying a STOP instruction followed by a RETURN was discovered and the higher order bits of the *enable* signal were also connected to the logic analyzer to study their behavior.

3 Subsystem 2: The Resizer (by Naoshin Haque)

3.1 Resizer System Overview

The purpose of the resizer is to take an original 64x64 pixel image of the alien and either replicate it or shrink it down to five different sizes. The six different sized images that can be produced are 64x64, 43x43, 32x32, 22x22, 16x16, and 11x11 pixels. In order to achieve this, interpolation and decimation methods are used on the original image. The image can be interpolated by a factor of 1 or 2 and decimated by a factor of either 1, 2, 3, 4, or 6. See Figure 3.1 on page 21 for an overall block diagram of the resizer subsystem.

The original colored image is stored in an external 32Kx8 Flash Memory, where only 4097 lines of memory are taken up. The first 4096 lines hold the image, and the 4097^{th} line is all zeros. The resizer receives a constant 3-bit size from the video controller unit. This size is used to look up the interpolation and decimation factors in a lookup table. After the size has been valid for at least one clock cycle, the resizer receives a load enable signal, called le_size, from the video game controller unit, signaling the resizer to begin its processing.

Once the resizer starts its processing, it goes through three or four stages, depending on the decimation factor. The FPGA sends control signals and addresses to all of the ROMs and the 32kx8 external RAM, accordingly. In addition, a tristate bus is shared between the FPGA, the RAM, and the ROM for the 8-bit data. The first stage is copying the original image from the ROM to the RAM or storing an upsampled version of the image in the RAM if the interpolation factor is 1 or 2, accordingly. Once the image is stored in the RAM, the ROM is no longer used for any calculations. The second stage is where the image stored in the RAM is interpolated, while the third stage is decimation

on this interpolated image. If the decimation factor is 4 or 6, the resizer goes into its fourth stage, so that it can perform decimation on the already decimated image.

The color being used for the image is 64-bits, meaning there are only four shades of intensity per R, G, and B. When reading from the RAM to do the calculations, the 8-bit data is separated into 2-bits for R, 2-bits for G, 2-bits for B, and the last 2-bits are discarded. The arithmetic unit is triplicated so that two-dimensional convolution can be performed on the R, G, and B in parallel. The two-dimensional convolution consists of looking at a 3x3 matrix around the appropriate R, G, or B component of the pixel and multiplying these 9 values with the appropriate filter coefficients. The filter coefficients are stored in a 64x8 ROM. The 9 products are accumulated and sent through a range to select the 2-bits for R, G, and B. These bits are then concatenated with 2 zeros and sent to the tristate bus to be written back to a different portion of the RAM.

Once all of the calculations for the two-dimensional convolution are completed and the final data has been written to the RAM, the video controller unit reads from the RAM. The addresses to the RAM, which are sent from both the resizer module and the video controller module, are put through four 8-to-1 multiplexers, while the data bus for the RAM is put through two tristate drivers. The output enable signals to the multiplexers and the tristate drivers are controlled by the video controller unit. Once the video controller unit is done processing, it once again sets the output enables to their correct values to enable the resizer to start its processing once again.

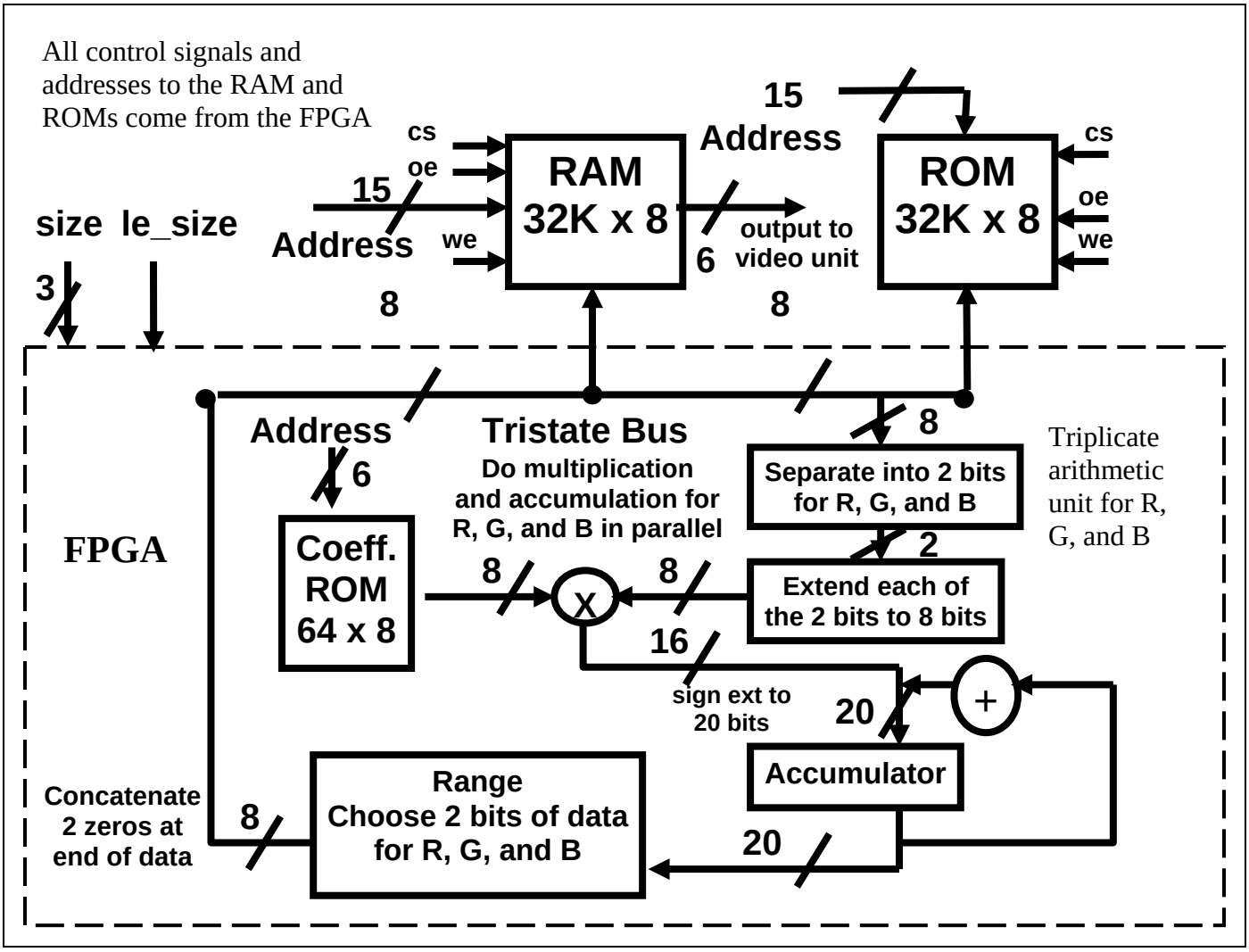


Figure 3.1: Overall Resizer Block Diagram

3.2 Resizer System Module Descriptions and Implementation

The resizer system consists of two main modules, which are the major finite state machine and the minor finite state machine (FSM). The minor FSM can be in four different stages, which are copying, interpolating, decimating, and decimating twice. The major FSM sends the appropriate control signals to tell the minor FSM which stage to be in. The minor FSM sends busy signals back to the major FSM, to denote when it is still processing. Using these communication signals, the major and minor FSM are able to coordinate such that the two-dimensional convolution is performed correctly.

3.2a Major FSM

The main purpose of the major FSM is to control which of the four stages the minor FSM is in. These four stages include copying, which includes both copying and upsampling, if necessary, interpolation, decimation, and decimation twice if needed. The major FSM starts off in the idle state, which it also returns to any time a reset button is pushed. The major FSM remains in the idle state until it receives a load enable signal from the video controller unit. It then sets start_copy to equal le_size, which is just high for one clock cycle. This tells the minor FSM to go to its copy stage. The major FSM then transitions to the copy state. See Figure 3.2 on page 24 for the major FSM state transition diagram.

Once in the copy state, the major FSM remains there as long as it receives a high busy_copy signal from the minor FSM. This busy_copy signal denotes that the minor FSM is still processing. Once the busy_copy signal goes low, the major FSM goes to the wait1 state, where it remains for one clock cycle. In this state, the major FSM sets the start_int signal to high, indicating that the minor FSM should now go into its interpolation stage. The major FSM then transitions to the interpolation stage, where it remains as long as it receives a high busy_int signal from the minor FSM. Once the minor FSM is finished performing the interpolation, it sends a low busy_int signal to the major FSM. Upon receiving this signal, the major FSM transitions to the wait2 state.

The wait2 state has a duration of one clock cycle and is also the state in which the major FSM sets start_dec1 to high. The major FSM then transitions to the dec1 stage, which is where the minor FSM outputs a high busy_dec1 signal as long as it is in this first decimation stage. After receiving a low busy_dec1 signal, the major FSM must look up

the decimation factor by using the size lookup table before progressing. If the decimation factor, M, is equal to 1, 2, or 3, the major FSM transitions to the idle state, where it remains until it receives another high load enable signal from the video controller unit. However, if M is equal to 4 or 6, the major FSM goes to stage wait3, where it sets start_dec2 to high. After staying in the wait3 state for one clock period, the major FSM transitions to the dec2 stage.

The major FSM continues to stay in the dec2 state as long as it receives a high busy_dec2 signal from the minor FSM. Once the minor FSM is finished doing the second decimation, it sends a low busy_dec2 signal to the major FSM. The major FSM then transitions back to its idle state, where it will remain until it receives another high load enable signal. When the major FSM has returned back to the idle state, it denotes that one full resizing has occurred, meaning that the final replicated or shrunken image has been written into the RAM, and is waiting to be read by the video controller unit.

Another signal in the major FSM module is the oe_driver signal, which is used as an output to the minor FSM. While the major FSM is in its idle state, the video controller unit must be able to read from the RAM, so the oe_driver signal is low only during this time to indicate this. Therefore, when the minor FSM is in its idle state, it will use the oe_driver signal as an input to set the control signals of the RAM accordingly.

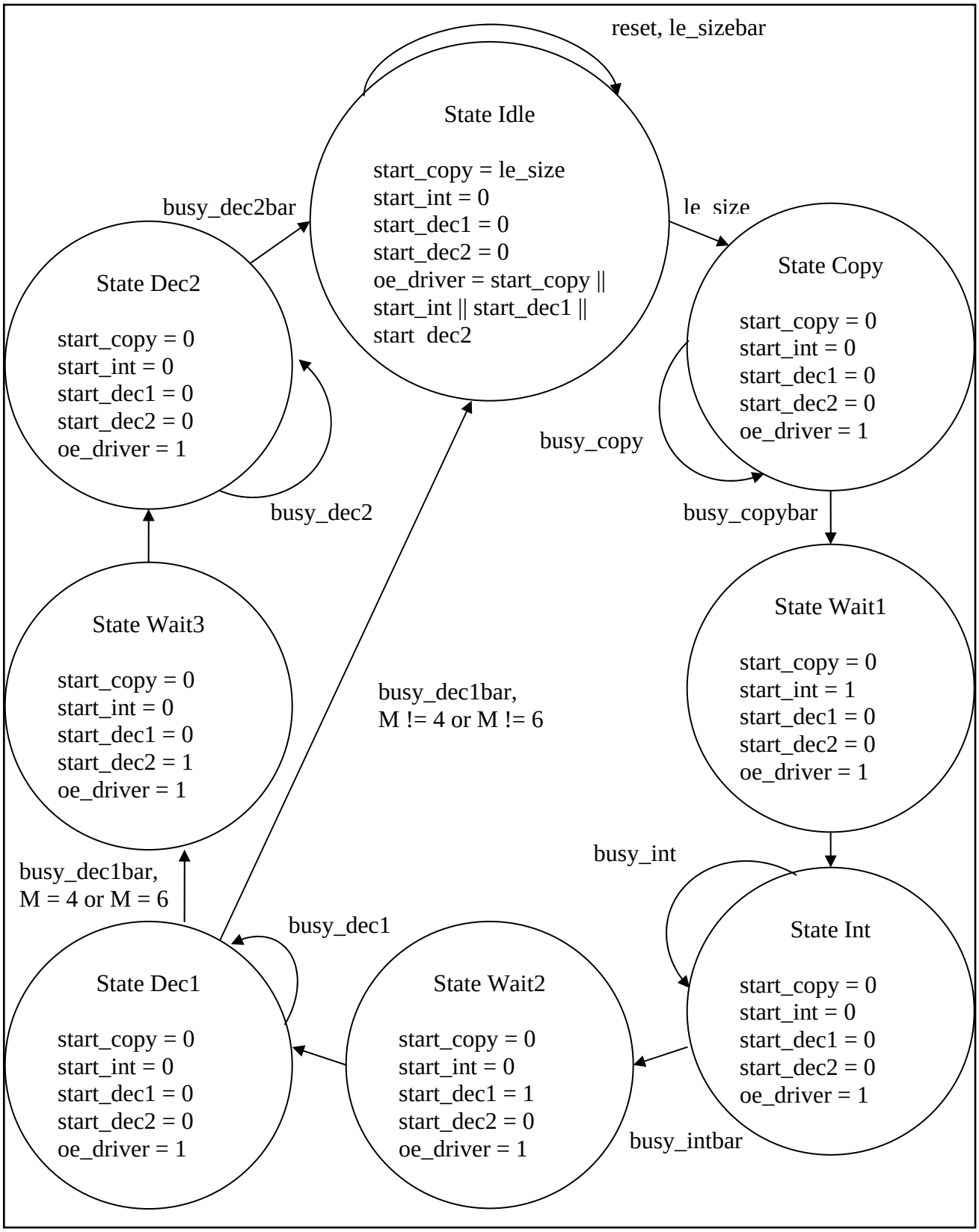


Figure 3.2: Major FSM State Transition Diagram

3.2b Minor FSM

The main purpose of the minor FSM is to perform all of the operations necessary to take the 64x64 image and resize it accordingly. The minor FSM can be in four different modes, which are the copy mode, the interpolation mode, the decimation mode, and the second decimation mode. The minor FSM has nine states, whose general state transition diagram can be seen in Figure 3.3 on page 28.

The minor FSM is able to receive four different start signals from the major FSM. Upon receiving one of these signals, the minor FSM will act accordingly by going to the correct mode. In addition, the minor FSM is the module that outputs the 15-bit addresses to the RAM and the ROM. The minor FSM also outputs the 6-bit address to the 64x8 filter coefficients ROM, which was generated using the Mega-Wizard, and is instantiated within the minor FSM module.

The minor FSM also receives a 3-bit size from the video controller unit, which it uses to look up the interpolation factor, L, and the decimation factor, M, in a lookup table. Upon reset, the minor FSM goes to its idle state, where all of the initial values are set. The minor FSM remains in the idle state until it receives a start signal from the major FSM. If this start signal is start_copy, the minor FSM sets busy_copy to high and transitions to the read_rom state. In this state, the minor FSM reads from the appropriate line of the ROM, according to the assigned address. If the interpolation factor is 1, the exact image is copied from the ROM to the RAM, so the 4097th of the ROM is never used. However, if the interpolation factor is 2, an upsampled image is stored into the RAM. This upsampled image contains zeros in every other row and zeros in every other column, so the ROM reads from the 4097^{th} line for quite some time.

The minor FSM stays in the read_rom state for one cycle and then transitions to state wait1. State wait1 lasts for 1 clock cycle and sets the control signals so that the data read from the ROM can be written to the RAM. The data pins of the external ROM are connected directly to the data pins of the external RAM, which are also connected to the FPGA. The minor FSM writes the data to the RAM in state write_ram1. After one clock period, the minor FSM transitions to the wait2 state. In this state, the minor FSM checks to see if all of the pixels of the image have been copied or upsampled depending on the interpolation factor. If this process is not done, the minor FSM will transition back to the read_rom state to read and copy another pixel from the ROM to the RAM. However, if the process of copying is complete, the minor FSM goes to the idle state, where it remains until it receives another start signal from the minor FSM. The minor FSM only sets busy_copy to low when it returns to the idle state.

The minor FSM goes through a very similar process when it receives either a high start_int, start_dec1, or start_dec2 signal. After receiving either of these signals, the minor FSM sets the appropriate busy signal to high. This busy signal remains high until the FSM returns to the idle state again. After any of these three start signals goes high, the minor FSM transitions to the read_ram state. The minor FSM remains in this state for one clock cycle and reads from the RAM according to the address assigned.

The minor FSM then transitions to the wait3 state, where it stays for one clock period. There are three multipliers, which are made by the Mega-Wizard, instantiated within the minor FSM module. Each of these multipliers take in two 8-bit inputs and outputs a 16-bit product. Since all of the numbers are positive, unsigned multiplication is being used. The two-dimensional convolution is performed by looking at a 3x3 matrix

around the appropriate R, G, or B component of the pixel. This means that each element of the matrix must be multiplied by the appropriate filter coefficient, which is read from the filter coefficients ROM. Since the R, G, and B are only 2-bits, 6 zeros are concatenated to the end of each component. This way higher precision arithmetic is being used, because the 6 zeros are considered to be after the decimal point. Then, these 9 products must be accumulated in order to output the new pixel. Since there are 9 products that are 16-bits each, three 20-bit accumulators are used to do the parallel processing of the R, G, and B. Therefore, all of the products must be sign extended from 16-bits to 20-bits. In the wait3 state, the accumulation of the sign extended multiplication products takes place.

There also many internal counters that are used in the wait3 state, as well as some other states in the minor FSM. The first counter, count0, is used to tell which mode the minor FSM is in. Since the copying uses different states than the minor FSM, it is not included in these modes. The minor FSM is in the interpolation mode when count0=1, the decimation 1 mode when count0=2, and the decimation 2 mode when count0=3. Another counter, count1 is used to count the entries in each row of the image being processed on at the time. Count2 and Count3 are 15-bit counters used to tell the RAM what address to read and write to, respectively. Another counter, count4, denotes when all of the accumulations are done, and which filter coefficient to use for this element of the matrix. Finally, count5 is used for the decimation by 6 stage to see whether an entry in the 22x22 matrix is divisible by 22 or not, since the modulo function in Altera was not working.

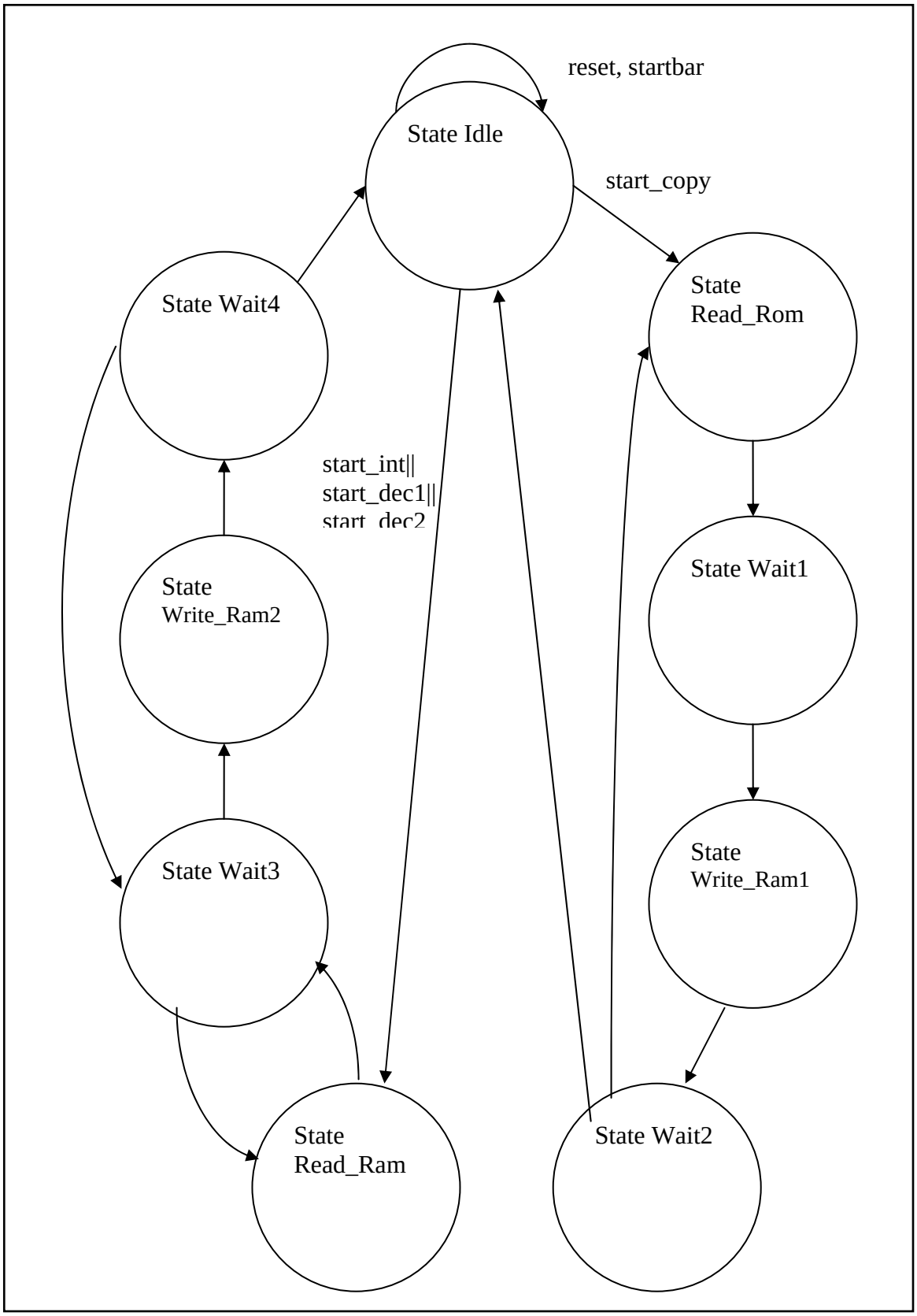


Figure 3.3: Minor FSM State Transition Diagram

In state wait3, count4 is always incremented. If count4 does not equal 9, many other steps must be taken. First, after staying in the wait3 state for one clock cycle, the minor FSM will then transition to the read_ram state. While in the wait3 state, count0 is checked to see whether the minor FSM is in the interpolation, decimation 1, or decimation 2 mode. Depending on the mode, the minor FSM will check the interpolation or decimation factor, so that it can act accordingly. When count4 is equal to 0 through 8, it assigns the address of the RAM so that data can be read from the RAM. It also assigns the address to the filter coefficients ROM, so that this data can be used in the multiplication along with the data from the RAM.

At each count4 value, the value of count2 is checked, to see which entry of the image matrix it is reading from. If count2 is assigned to a pixel that is placed in either the top row, bottom row, most left column, or most right column, then some of the entries of the 3x3 matrix surrounding the pixel will not exist in the image matrix. For instance, in Figure 3.4 shown on page 30, the entry 0 will only have four existing elements in its 3x3 matrix, which are the entries 0, 1, 6, and 7. In order to make sure the other products go to 0, count2 is checked to see whether it is in either the top, bottom, left, or right according to the what size image is being looked at the time. If the entry is in either of these locations, then the address of the filter coefficients RAM is set to 45, which is a line of 8 zeros. This way the product is equal to 0, and this value will not be taken into consideration during the accumulation. Usually the entries can be checked by seeing if count2 falls within a range of values or if count2 is a multiple of some power of 2 according to the size of the matrix. However, the only size where this was not the case was when doing decimation on the 22x22 image. Therefore, the counter count5 needed to

be used to check whether the entry was divisible by 22 or not, in order to see where the entry was located in the image matrix.

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

Figure 3.4: Example Image Matrix

However, if count4 equals 9, this means that all of the calculations in the arithmetic unit have taken place, so the data is ready to be written into the RAM. In this case, the address to be written into is set to count3 and the control signals of the RAM are set appropriately. Since the accumulation results are 20-bits, they must be sent through a range to pick 2-bits for R, 2-bits for G, and 2-bits for B. In the write_ram2 state, the data output to the tristate bus is then the concatenation for the bits for R, G, and B, with 2 zeros, in order to comply with the 8-bits that must be written into the RAM.

After this data has been written to the RAM, the minor FSM sets all three of the accumulators back to zero, in order to make them ready for the next set of calculations. The minor FSM then transitions to the wait4 state, where it increments the address of where the RAM should be written to, which only takes effect if the minor FSM ever transitions back to the write_ram2 state. Also in the wait4 state, the minor FSM first checks which mode it is in. Then, according to the mode, it uses either the interpolation

and decimation factor to check whether all of the data has been written to the ram by checking the value of either count2 or count3.

If all of the data has not been written to the RAM, the minor FSM transitions to the wait3 state, where it outputs the addresses and correct control signals to the RAM and coefficients ROM. This denotes the beginning of another cycle through the arithmetic unit, which will perform all of the necessary multiplication and accumulation before outputting another pixel of the new image. However, if all of the data for this new image has been written to the RAM, the minor FSM returns to the idle state. The minor FSM remains in this idle state, until it receives another high start signal from the major FSM.

3.2c Overall Resizer Module

The top level module is a very simple module, which included instantiations of both the major and minor FSMs. The inputs to the entire resizer system are the clock, the synchronized reset from the game controller unit, the 3-bit size and le_size signal from the video controller unit, and the 8-bit data from the RAM. The output of the resizer system are the 15-bit addresses to the RAM and the 32Kx8 ROM, and all of the control signals to both of these external memory devices. All other signals that were just used to communicate internally between the ROM and the RAM were wired within the top level module for the resizer system.

3.3 Testing and Debugging

Due to the fact that much time was spent planning the design and implementation of the process, only a short time was needed to debug the entire module. First, the major FSM was designed and tested. Since this module was quite simple, it worked perfectly on the first try. Next, the complex minor FSM was tested. Various things had to be checked

for this module. For instance, each of the modes had to do all of the calculations correctly and at the correct time, as well as transitioning to the proper state. In addition, the counters had to increment in the proper pattern, so that the RAM and ROMs were being read from and in the case of the RAM, also written to correctly. After changing minor issues, such as the timing of the busy signals and timing of the count4 values, the minor FSM worked properly for a general set of filter coefficients. In addition, the top level module had to be tested to make sure that both of the modules worked together properly. After fixing a few timing issues between state transitions, such as the timing for the multiplication between the correct filter coefficient and the correct component of the image matrix, the top level module seemed to work correctly.

After the resizer seemed to be working for any general filter coefficients, the correct filter coefficients needed to be found. A MATLAB program had to written in order to find the most accurate coefficients. After this, the resizer needed to be tested to see the real-time image that it produced. Since the resizer was not interfaced with the video controller unit, there needed to be a way to output the image from the RAM onto a screen.

At first serial communication was going to be used to read the data from the RAM and output it to the computer screen. However, the timing for this was very complicated, so instead, the HP Logic Analyzer was used to store data. Using the Logic Analyzer in the state mode, a set of triggers can be set such that the Logic Analyzer only stores data on certain states. The Logic Analyzer was used to check the state transitions and data that was being written to the RAM. While the range was pretty much chosen by normalizing

the filter coefficients, it was still helpful to check them and at times even improve them by looking at the data from the Logic Analyzer.

The stored data in the Logic Analyzer was then saved using a floppy disk and converted into the appropriate format to be used in MATLAB. A program was written to change the 8-bit data into a 64-bit color image. Using this program, it was demonstrated that the resizer module was working as it should be and was able to output 6 different sized images of the alien.

4 Subsytem 3: Video Controller Unit (by Matthew Kwan)

4.1 Video Controller Overview

In the final project, I was in charge of the video controller. The general purpose of the video was being able to display each screen of the video game or any function run by the game controller. It did this by taking input from the game logic subsystem and used it to generate the images displayed to the user. The video component was partially designed using 3 external ROMs which stored every sprite. In addition to being an important feedback tool for the game controller, the video component was also used as a useful debugging tool for both the game controller and the video controller. The video system displayed a 64-bit color VGA video at 60 frames per second. The resolution was 320 by 480. There were 4 major modules. These were the sync_generator module, the line_register module, the sprite_table module, and the overview module which pieced together the entire video component. The 4 modules were internal, meaning that they were written and programmed into the FPGA. See Figure 4.1 for block diagram.

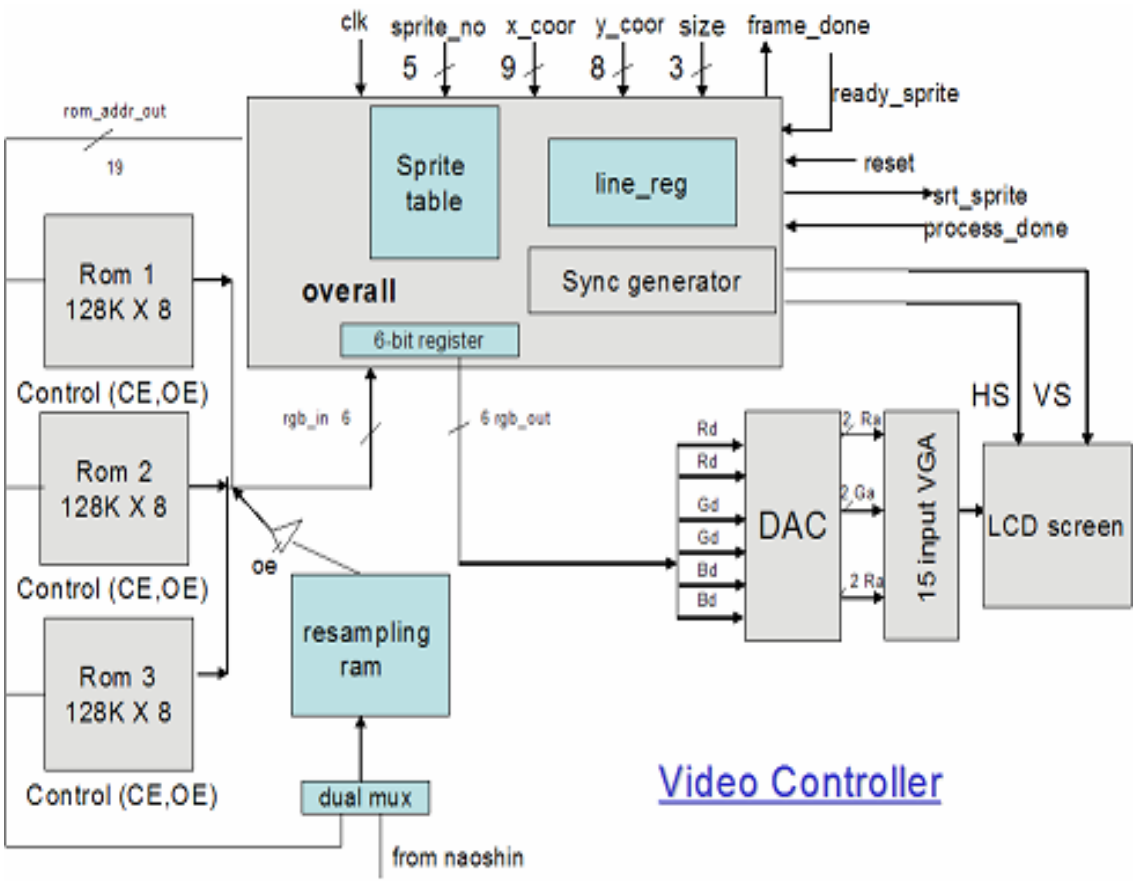


Figure 4.1: Block Diagram of Video Controller

4.2 Sprite_table Module

The sprite_table module was essentially a lookup table. Its inputs included the sprite numbers to be displayed onto the screen, and if the sprite number referred to the enemy sprite, a certain size would also be inputted. Altogether there were 25 sprites. Each sprite has as outputs the memory address where the 1st pixel of a particular sprite is located at in the memory, and the sprite's corresponding width and height. The first two bits of the memory address output was dedicated to determining which of the 3 ROMs or RAM were being used. Two of the ROM's contained the 4 road sprites which were each 240 by 240 resolution. The other ROM contained the background image and all the other

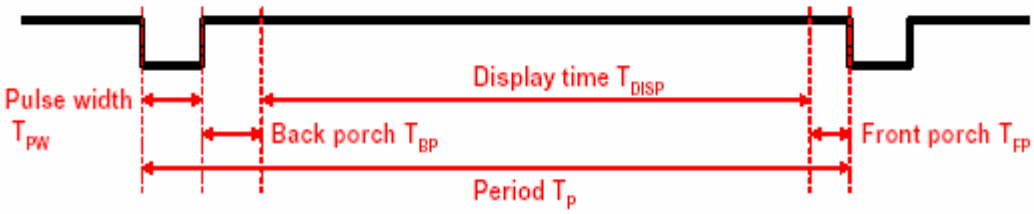
sprites, including the powerup and score. Most of the sprites were drawn on Paint, the rest were drawn on Coral Draw. The sprites were drawn using RGB values, with 2 bits assigned to each color, therefore each color having 4 different shades. The overall number of colors was 64. The sprites were then saved as a bitmap file, and formatted appropriately using Matlab. Lastly, the sprites were programmed on the chip. The RAM, on Naoshin's board, contained the enemy image whose size could change depending on the current stage of the game, and what size the game controller inputted. Because both the resampler and the video controller had to use the same RAM, a dual input multiplexor was used to figure out when either the resampler or the video controller could access it.

4.3 Sync_generator Module

The sync_generator module generates the control signals necessary to drive the VGA monitor. These signals include the vertical sync, the horizontal sync, the vertical blanking, and the horizontal blanking signals. The sync_generator module reads the color value of the current pixel from the external ROMs and outputs it to the DAC, which consists of a 6-bit register and 6 resistors. The DAC performs the digital to analog conversion necessary to display the data on the VGA monitor. Both the game controller and the video controller operated on the same 12mhz clock. This was used so that the timing controls between the two parts would be in sync.

The 12 mhz clock was mainly used as a "pixel clock" to display on the VGA screen. Every time the clock pulsed high, the current pixel data presented to the 6-bit register in the DAC would be latched in and converted to analog through the resistors. Each horizontal sync pulse is preceded by a complete line of pixels. Each vertical sync pulse is similarly preceded by a complete frame of lines of pixels. Each sync is

surrounded by a blanking interval, which consists of the back and front porches, and a pulse width. The duration and timing of these pulses depends on the resolution and refresh rate of the VGA monitor being used. See figure 4.2 below.



Format		CLK	P	PW	BP	DISP	FP
VGA	HS (pixels)	25Mhz	794	95	47	640	13
	VS (lines)	--	528	2	33	480	13

Figure 4.2: Timing durations based on 25 mhz clock, for timing based on 12 mhz clock multiple each number by .48

4.4 Timing Controls

Every time after the last pixel of a frame is displayed on the screen, the game controller gives a frame_done signal. In the overall module, this is when vblankon is pulsed high. The game controller then deasserts the process_done signal to tell the video controller that it is done processing all the information needed to setup for the next picture frame to be displayed onto the screen. When the game controller is done processing, which should take much less than 2x10^4 clock cycles (app. the length of time vblanking is on), it's process_done signal then goes high. This signals the opportunity for the video controller to give the game controller a start_sprite signal whenever h_blankon is high and v_blankon is not high. Whenever the game controller is ready, the ready_sprite signal will be asserted high. The game controller will then "spit" out 32 sprite numbers and their x and y coordinates that will be in the next frame, with each sprite number and corresponding information given during each clock cycle. The

video controller uses this information to detect whether the certain pixels of any of the sprites the game controller “spits” out are on a given line. Since there are only 25 sprites in total and most of them won’t be displayed on a frame at the same time, the game controller will input to the video controller a sprite number of zero to signal that there is no sprite. After the 32 clock cycles, the game controller’s ready_sprite signal is deasserted, and the video controller can start displaying on the screen whenever its h_reset signal is pulsed high. The cycle then starts over again when the video controller finishes displaying a line of pixels on the screen, and its h_blankon signal is high (meaning the video controller gives another start_sprite signal). See Figure 4.3 below for timing diagram.

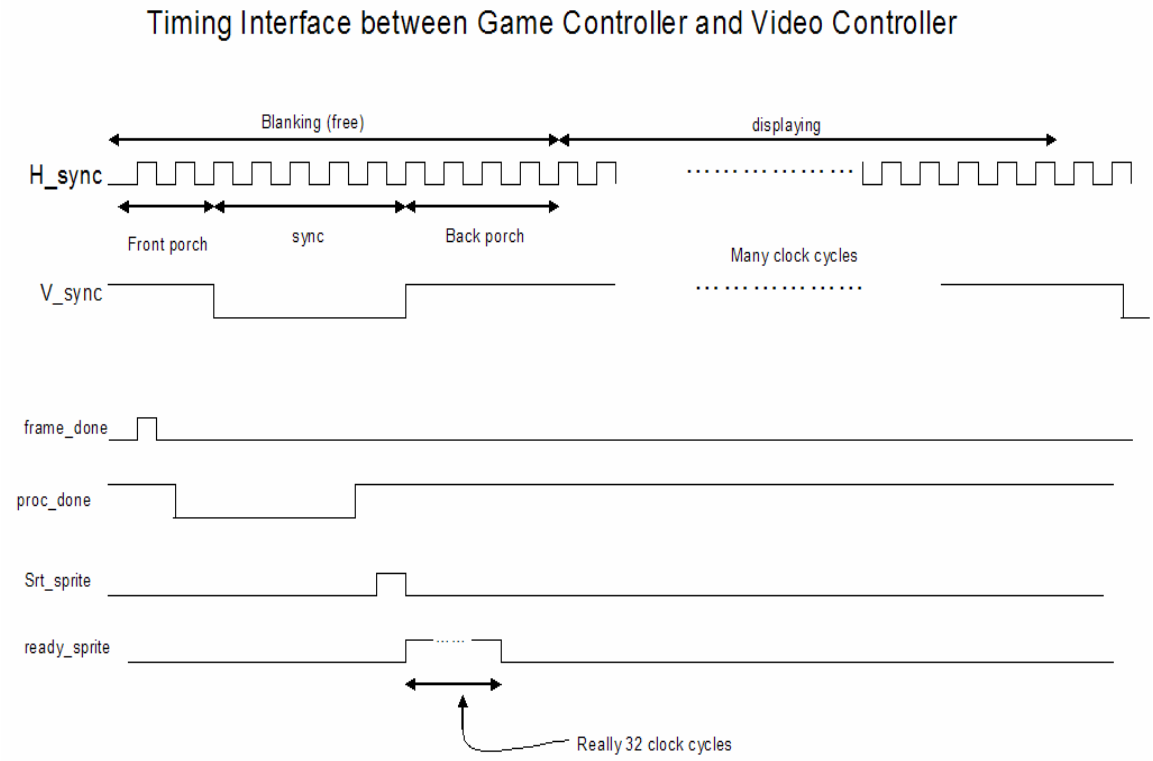


Figure 4.3: timing interface. Not in scale

4.5 Line Register Module

The line register module is primarily used to store all the requisite information of each sprite that is supposed to be on a given line on the screen. At each line, the overview module (which will be discussed shortly) will check to see whether or not the sprites given by the game controller have pixels that are contained in that line. If it is, the load_enable signal goes high, and that particular sprite's width, height, and rom_addr (also discussed later) gets latched into a register. Those three pieces of information of each sprite is stored into registers to hold and remember the values later on. All this is done during the horizontal blanking intervals. When it comes time to display on to the screen (at h_reset), the x_hit signal then checks which of the stored sprite's pixels on that line get displayed on the screen. This is done by checking if the current x_coor of the pixel at display is greater or equal to the x_coor of the sprite (given by the game controller) with larger precedence and less than the x_coor plus width of that sprite. If it is, x_hit is one, otherwise x_hit is zero. The overview module then proceeds to display that most precedence sprite's pixels onto the screen on the particular line. Whenever reset is high (which is when srt_sprite is high, see the instantiation in overview) or frame_done is high, the video controller makes sure to clear all the previous line's information from the registers so that it can use those registers to store information for the next line of pixels. In addition, the video controller sets the x_coor register to a large value, so that it won't accidentally get an x_hit.

4.6 Overview Module

The overview component is the module that puts all the other modules together, and directly interfaces with the game controller's top module. Whenever the game controller's ready_sprite signal is high (this means when game controller tells me the sprites that are in the current frame), the video controller checks whether yt (where the current y_coor of the pixel at display is on screen) is greater than or equal to the y_coor of the sprite or less than/equal to the y_coor plus height of the sprite. It also checks to see if the sprite_no is not zero. If these conditions are true, y_hit is high which means the sprite is on the next line, and will get stored into the registers in the line_reg module. Otherwise, y_hit is low and nothing happens. The video controller also keeps a counter in this module. Initially, when srt_sprite is high, its counter is set to zero. Whenever ready_sprite and y_hit are asserted high, it increments the counter. The video controller then enables the load signal (which would load the sprite information into a register as described in the line_reg module) at each count and when y_hit is high. Because it is not expected for there to be more than 10 sprites displayed onto a single line in the screen, there are only 10 load enable signals and, therefore, 10 instantiations of the line_reg module in the overview module. As mentioned before, the registers in the line_reg module latch in 3 different values, one which is the rom_addr. The rom_addr is not merely the same as the mem_addr, which is the address of one of the 3 ROMs where the 1st pixel of a particular sprite is at. Rather, the rom_addr is the address in the ROM whose data (pixel) contents are fetched, and whose address changes depending on where in the screen the current pixel is being displayed. Besides knowing the mem_addr to calculate

the rom_addr, the y_coor, x_coor, width of the sprite, and yt are used to calculate the rom_addr. Although yt is 10 bits, the least significant bit is taken out because each line needs to be displayed twice on the screen to get the 640 by 480 frame. This is because all the sprites were drawn relative to a 320 by 240 frame. The calculation of rom_addr abnormally took a vast majority of the 83 nanoseconds of the clock, because of the somewhat complicated computation and huge propagation delays. This complicated things because the video controller had to store all the sprite information in one clock cycle, since the game controller "spitted" out each sprite in one clock cycle. Therefore, all inputs to the rom_addr made as registers and, in consequence, delayed by one clock cycle. This meant that though the game controller gave the sprite number and its corresponding x_coor and y_coor at a particular cycle, the information and calculations wouldn't be latched in until the following clock cycle. This gave ample time for the rom_addr calculation as well as other things to be done in one clock cycle. The addition of xt was not added into the rom_addr until much later when actually displaying on the screen to further alleviate the situation of having a potential problem due to "heavy" computation. If there were no x-hits, then the background pixels would show up by default.

4.7 Debugging and Testing

During the course of the final project, my video controller went through many design stages. The biggest one turned out to be one week before it was due. In my previous design, I used many external chips, including two SRAMs used for memory. Two video memory chips were used because so that I could interleave the two memories together such that one would be read from at the same time the other would be written to.

I also would have used a number of tri-state buffers and a 8-bit register for my DAC. Though I did a lot of extensive simulations on my modules pertaining to that previous design, it was unfortunate that I only realized the futility of my design this late into the process when I was about to wire it and assign pins. In total, I had to use approximately 70 pins, while the FPGA only had 50+ pins. This did not include the additional 10 pins that the game controller needed since we would be using the same FPGA. Although it was still somewhat feasible to decrease the number of pins using various methods, including faking more pins by time multiplexing several signals onto one set of pins, I would have needed to run at a higher clock frequency in order to get the same amount of information as the non-multiplexed scheme out of the chip within a given time interval. This meant running I would have had to run at a 25mhz pixel clock, which was not guaranteed to work on the FPGA kits. In order to be within the allotted set of pins, I still would have had to reduce approximately 15 pins, and though this was plausible theoretically (I won't go into detail) it was both not guaranteed to work and even more timing issues would be involved. I also decided to do away with my original design because the amount of wiring would have been enormous. Although I was extremely hesistant to implement the new design back then, in the end, it turned out to be a success. Though there was still a lot of wiring, it was much less than what it could have been. The number of FPGA pins that I used were also within the scope of what the FPGA could provide.

One of the biggest issues I had when debugging my overall module was the amount of propagation delay I had because of somewhat "heavy" computation which used depended on a lot of inputs. I couldn't afford this since I had to do everything in one

clock cycle, since there was only a limited amount of time the hblanking could provide. This led me to delay all the inputs by one clock cycle which was good, since I would be using up only an extra clock cycle as mentioned previously. Another issue I had was the number of if and else if statements I had in my code, with more than half of those statements in the sprite table module. One improvement I made to my code was the use of case statements. The optimization of my code in the sprite table module managed to decrease the computation and propagation delays by nearly 30ns, which was a lot considering each clock cycle was 83ns. Another problem I had was that I didn't clear my registers in the line_reg after each line drawn on the screen. This presented problems since there would be times when I was not supposed to have an x-hit, but had one anyway.

5 Conclusion

Design which facilitated easy interfacing between three separate components was the lesson to be learnt from this experience. The design process itself consumed a large part of our project time. It was particularly challenging to work out the timing between the three separate components, especially since the Video Controller needed to output to the screen continuously. We learnt how to bypass such timing constraints with clever design techniques, which allow the processing time of the Game Controller to occur during the blanking of the vertical signal from the Video Controller. We also learnt about designing modules while taking into consideration the total size of the FPGA and EAB's. At the very beginning, each of us was busy designing his/her own module that we completely overlooked the size and pin limitations of the FPGA.

The progress of the project should have been planning more carefully, allocating more time towards implementation and interfacing, since we learnt that interfacing consumes a very large amount of time. In fact, we couldn’t manage to fit the Resizer Module onto one FPGA, and in the end we did not have time to interface with the Resizer. Yet, the project, although challenging, was a greatly informative experience, and although our results did not match our original expectations, we were quite satisfied with the progress we made.

6 Appendix: Verilog Code

6.1 Game Controller Unit

6.1a Controller

```
module controller(user_in, clk, xcoor, ycoor, size, sprite_no, le_size, frame_done, ready_sprite, start_sprite,
reset_apply,
 data_out, data_in, process_done, wr, reset_done, interrupt_valid, enable, instruction);
input [4:0] user_in;
input clk, frame_done, start_sprite;
output [7:0] data_out, data_in;
output [19:0] instruction;
output[8:0] xcoor;
output [7:0] ycoor;
output [2:0] size, enable;
output [4:0] sprite_no;
output wr;
output le_size, process_done, reset_apply, ready_sprite, reset_done, interrupt_valid;
wire  stop, wr, busy,  temp_reg3, temp_reg4, reset_done, reset_apply, interrupt_valid;
wire [2:0] pcsel, size_dat;
wire [1:0] stack_sel;
wire [2:0] enable;
wire [6:0] reg_no;
wire [8:0] branch, call, pc;
wire [7:0] data_in, data_out;
wire [19:0] instruction;

insrom myrom(pc, instruction);
decoder insdecoder(clk, user_in, instruction, pcsel, stack_sel, enable, data_out, data_in, le_size, size,
stop,branch,
        call, reg_no, wr, temp_reg3, temp_reg4, frame_done, reset_apply, reset_done, reset_reg,
interrupt_valid);

pc mypc(clk, reset_apply, pcsel,call, branch,frame_done, enable, pc, stack_sel, stop, reset_done, tos, nos);

reg_file regfile(clk, reset_apply, start_sprite, frame_done, wr, data_in, data_out,
 reg_no, sprite_no, xcoor, ycoor,  ready_sprite, stop, wr_sprite, wr_xcoor, wr_ycoor, reset_done,
process_done, interrupt_valid);


endmodule
```

6.1b Decoder

```
module decoder(clk, user_in, instruction, pcsel, stack_sel, enable, data_out, data_in, le_size, size_dat,
stop,branch,
        call, reg_no, wr, temp_reg3, temp_reg4, frame_done, apply_reset, reset_done, reset_reg,
interrupt_valid);
input clk, frame_done, reset_done, interrupt_valid;
input [4:0] user_in;
input[19:0] instruction;
input [7:0] data_out;
output [2:0] enable;
output stack_sel, le_size, stop, wr, temp_reg3, temp_reg4, apply_reset, reset_reg;
output [7:0] data_in;
output [2:0] size_dat, pcsel;
```

```
output [8:0] branch, call;
output [6:0] reg_no;

reg [2:0] enable;
reg stack_sel, size_en, sizen, le_size, stop, reset, wr, intreg1, intreg2, intreg3, intreg4, apply_reset,
reset_reg;
reg temp_reg3, temp_reg4;
reg [7:0] data_in, temp_reg1, temp_reg2;
reg [2:0] size_dat, pcsel;
reg [3:0] state;
reg [8:0] branch, call;
reg [6:0] reg_no;
reg [4:0] user_in_old, control;

parameter op_assign=0;
parameter op_addc = 1;
parameter op_branchcond = 2;
parameter op_compeq =3;
parameter op_beq  = 4;
parameter op_call = 5;
parameter op_return = 6;
parameter op_stop = 7;
parameter op_size = 8;
parameter op_compeqc = 9;
parameter op_comple = 10;
parameter op_complec = 11;
parameter op_and = 12;
parameter op_or = 13;
parameter op_eqin = 14;
parameter op_bne = 15;


parameter state_write = 0;
parameter state_op = 1;
parameter state_process = 2;
parameter state_read1 = 3;
parameter state_read2= 4;
parameter state_incpc = 5;
parameter state_idle = 6;
parameter state_reset = 7;
parameter state_interrupt = 8;



        always @ (posedge clk)
                begin
                        //Synchronizer for the inputs
                        user_in_old <= user_in;
                        control <= user_in_old;
                        reset <= control[0];
                        sizen <= size_en;
                        le_size <= sizen;

                        wr <=0;
                        branch <= instruction[8:0];
                        call <= instruction [8:0];
```

```
stop <= 0;
size_en <= 0;
stack_sel <=0;
apply_reset<=0;
enable <=0;
reset_reg <= reset? 1: reset_reg;

if(reset_reg )
        begin
                pcsel <=4;
                enable <= 0;
                stack_sel <= 0;
                stop<= 0;
                wr<=0;
                reset_reg <=0;
                apply_reset <=1;
                state <= state_reset;
        end
        else if((control [4:1] !=0) && interrupt_valid)
                begin
                        call<= 461;
                        pcsel <= 2;
                        enable <= 7;
                        stack_sel <=0;
                        state <= state_interrupt;
                end
else case(state)
state_reset:
        begin
                if(reset_done)
                        begin
                        state<= state_op;
                reset_reg <=0;
                data_in <=0;
                apply_reset <=0;
                        end
        end
state_interrupt:
        begin
        enable <=0;
        pcsel<=4;
        state<= state_incpc;
        end
state_write:
                begin
                        wr<= 1;
                        state <= state_incpc;
                        pcsel <=0;
                end
        state_incpc:
                begin
                        if(stop)
                                begin
                                        pcsel <=4;
                                        state <= state_idle;
                                end
```

```
                else
                        begin
                        pcsel<= 4;
                        state <= state_op;
                        end
                end
        state_idle:
                begin
                        if(frame_done)
                                begin
                                        state<= state_op;
                                        pcsel <= 4;
                                end

                end
        state_process:
                begin
                        case(instruction[19:16])
                                op_addc:
                                        begin
                                                data_in<= temp_reg1 +
instruction[7:0];
                                                state<= state_write;
                                        end
                                op_compeq:
                                        begin
                                                case (instruction[1:0])
                                                        0: intreg1 <=
(temp_reg1 == temp_reg2)? 1: 0;
                                                        1: intreg2 <=
(temp_reg1 == temp_reg2)? 1: 0;
                                                        2: intreg3 <=
(temp_reg1 == temp_reg2)? 1: 0;
                                                        3: intreg4 <=
(temp_reg1 == temp_reg2)? 1: 0;
                                                endcase
                                                state<= state_incpc;
                                                pcsel <= 0;
                                        end
                                op_comple:
                                        begin
                                                case (instruction[1:0])
                                                        0: intreg1 <=
(temp_reg1 <= temp_reg2)? 1: 0;
                                                        1: intreg2 <=
(temp_reg1 <= temp_reg2)? 1: 0;
                                                        2: intreg3 <=
(temp_reg1 <= temp_reg2)? 1: 0;
                                                        3: intreg4 <=
(temp_reg1 <= temp_reg2)? 1: 0;
                                                endcase
                                                state<= state_incpc;
                                                pcsel <= 0;
                                        end
                                op_complec:
                                        begin
```

```
                                                                        case (instruction[1:0])
                                                                                0: intreg1 <=
(temp_reg1 <= instruction[8:2])? 1: 0;
                                                                                1: intreg2 <=
(temp_reg1 <= instruction[8:2])? 1: 0;
                                                                                2: intreg3 <=
(temp_reg1 <= instruction [8:2])? 1: 0;
                                                                                3: intreg4 <=
(temp_reg1 <= instruction[8:2])? 1: 0;
                                                                        endcase
                                                                        state<= state_incpc;
                                                                        pcsel <= 0;
                                                                end
                                                        op_compeqc:
                                                                begin
                                                                        case (instruction[1:0])
                                                                                0: intreg1 <=
(temp_reg1 == instruction[8:2])? 1: 0;
                                                                                1: intreg2 <=
(temp_reg1 == instruction[8:2])? 1: 0;
                                                                                2: intreg3 <=
(temp_reg1 == instruction [8:2])? 1: 0;
                                                                                3: intreg4 <=
(temp_reg1 == instruction[8:2])? 1: 0;
                                                                        endcase
                                                                        state<= state_incpc;
                                                                        pcsel <= 0;
                                                                end
                                                        default: begin
                                                                        state <= state_incpc;
                                                                        pcsel <=0;
                                                                end
                                                endcase
                                        end
                                state_read1:
                                        begin
                                                temp_reg2 <= data_out;
                                                reg_no <= instruction [15:9];
                                                state <= state_read2;
                                        end
                                state_read2:
                                        begin
                                                temp_reg1 <= data_out;
                                                state <= state_process;
                                        end

                                state_op:
                                 case(instruction[19:16])
                                        op_assign:
                                                begin
                                                        reg_no <= instruction [15:9];
                                                        data_in <= instruction [7:0];
                                                        enable <=0;
                                                        state<= state_write;
                                                end
                                        op_addc:
```

```
		begin
			reg_no <= instruction [15:9];
			enable <=0;
			state<= state_read1;
		end
op_branchcond:
		begin
			if(instruction[15:11] == control)
				pcsel <= 1;
			else
				begin
					pcsel <=0;

				end
				enable <= 0;
				state <= state_incpc;
		end
op_compeq:
		begin
			reg_no <= instruction[8:2];
			state<= state_read1;
			enable <= 0;
		end
op_beq:
		begin
			case(instruction[15:14])
				0: pcsel <= intreg1? 0:1;
				1: pcsel <= intreg2? 0:1;
				2: pcsel <= intreg3? 0:1;
				3: pcsel <= intreg4? 0:1;
			endcase
			enable <= 0;
			state <= state_incpc;

		end
op_bne:
begin
			case(instruction[15:14])
				0: pcsel <= intreg1? 1:0;
				1: pcsel <= intreg2? 1:0;
				2: pcsel <= intreg3? 1:0;
				3: pcsel <= intreg4? 1:0;
			endcase
			enable <= 0;
			state <= state_incpc;

		end

op_call:
		begin
			pcsel <= 2;
			enable <= 3;
			stack_sel <= 0;
			state <= state_incpc;

		end
```

```
op_return:
        begin
                pcsel <=3;
                stack_sel <= 1;
                enable <= 1;
                state <= state_incpc;
        end
op_stop:
        begin
                stop <= 1;
                enable<= 0;
                pcsel <= 0;
                state <= state_incpc;
        end
op_size:
        begin
                size_dat<= instruction [2:0];
                size_en <= 1;
                pcsel <= 0;
                enable <= 0;
                state <= state_incpc;
        end
op_comple:
        begin
                reg_no <= instruction[8:2];
                state<= state_read1;
                enable <= 0;
        end
op_compeqc:
        begin
                reg_no <= instruction[15:9];
                state<= state_read2;
                enable <= 0;
                //pcsel <= 4;
        end
op_complec:
        begin
                reg_no <= instruction[15:9];
                state<= state_read2;
                enable <= 0;
        end
op_and:
        begin
                case(instruction [11:10])
                        0: intreg1 <= (temp_reg3 &&
temp_reg4)? 1:0;
                        1: intreg2 <= (temp_reg3 &&
temp_reg4)? 1:0;
                        2: intreg3 <= (temp_reg3 &&
temp_reg4)? 1:0;
                        3: intreg4 <= (temp_reg3 &&
temp_reg4)? 1:0;
                endcase
                enable <=0;
                pcsel <=0;
                state <= state_incpc;
```

```
                                        end
                                op_or:
                                        begin
                                                case(instruction [11:10])
                                                        0: intreg1 <= (temp_reg3 ||
temp_reg4)? 1:0;
                                                        1: intreg2 <= (temp_reg3 ||
temp_reg4)? 1:0;
                                                        2: intreg3 <= (temp_reg3 ||
temp_reg4)? 1:0;
                                                        3: intreg4 <= (temp_reg3 ||
temp_reg4)? 1:0;
                                                endcase
                                                enable <=0;
                                                pcsel <=0;
                                                state <= state_incpc;
                                        end
                                op_eqin:
                                        begin
                                                case(instruction [12:11])
                                                        0: intreg1 <= (temp_reg3 ==
instruction[10]);
                                                        1: intreg2 <=(temp_reg3 ==
instruction[10]);
                                                        2: intreg3 <= (temp_reg3 ==
instruction[10]);
                                                        3: intreg4 <=(temp_reg3 ==
instruction[10]);
                                                endcase
                                                pcsel <= 0;
                                                enable <=0;
                                                state <= state_incpc;

                                        end

                        endcase


                endcase




        end

always @(state)
        begin
                if(state == state_op)
                        begin
                                case(instruction[19:16])
                                        op_and:
                                                begin
                                                case(instruction [15:14])
                                                        0: temp_reg3 <= intreg1;
                                                        1: temp_reg3 <= intreg2;
                                                        2: temp_reg3 <= intreg3;
```

```
                                        3: temp_reg3 <= intreg4;
                                    endcase
                                    case(instruction [13:12])
                                        0: temp_reg4 <= intreg1;
                                        1: temp_reg4 <= intreg2;
                                        2: temp_reg4 <= intreg3;
                                        3: temp_reg4 <= intreg4;
                                    endcase
                                end
                            op_or:
                                begin
                                    case(instruction [15:14])
                                        0: temp_reg3 <= intreg1;
                                        1: temp_reg3 <= intreg2;
                                        2: temp_reg3 <= intreg3;
                                        3: temp_reg3 <= intreg4;
                                    endcase
                                    case(instruction [13:12])
                                        0: temp_reg4 <= intreg1;
                                        1: temp_reg4 <= intreg2;
                                        2: temp_reg4 <= intreg3;
                                        3: temp_reg4 <= intreg4;
                                    endcase
                                end
                            op_eqin:
                                begin
                                    case(instruction[15:13])
                                        0: temp_reg3 <=
control[0];
                                        1: temp_reg3 <= control
[1];
                                        2: temp_reg3 <= control
[2];
                                        3: temp_reg3 <=
control[3];
                                        4: temp_reg3 <= control
[4];
                                        default: temp_reg3 <=0;
                                    endcase
                                end
                            default:
                                begin
                                    temp_reg3<=0;
                                    temp_reg4 <=0;
                                end

                        endcase
                    end
                    else begin temp_reg3 <=0; temp_reg4 <=0; end
                end


endmodule
```

6.1c PC

```
module pc(clk, reset_apply, pcsel,call, branch,frame_done, enable, pc, stack_sel, stop, reset_done, tos,
nos);
        input clk, frame_done, stack_sel, stop, reset_apply, reset_done;
        output[8:0] tos, nos;
        input [2:0]  pcsel;
        input [2:0] enable;
        input [8:0] call, branch;
        output[8:0] pc;
        wire [8:0] new_pc, return;
        reg [8:0] pc;
        reg[8:0] tos, nos;
        reg [1:0] state;

        parameter state_idle = 0;
        parameter state_busy = 1;
        parameter state_reset = 3;

        assign new_pc = reset_apply? 0 : pc +1;
        assign return = tos;

        always @(posedge clk)
                begin
                        if(reset_apply)
                                begin
                                        pc <= 0;
                                        tos<=0;
                                        nos<=0;
                                        state<= state_reset;
                                end
                        else case(state)
                                        state_idle:
                                                begin
                                                        if(frame_done)
                                                                begin
                                                                        state<= state_busy;
                                                                end
                                                end
                                        state_reset:
                                                begin
                                                        if(reset_done)
                                                                begin
                                                                        state <= state_busy;
                                                                end
                                                end
                                        state_busy:
                                                begin
                                                        if(stop)
                                                                begin
                                                                        state <= state_idle;

                                                                end

                                                        case(pcsel)
                                                                0: pc <= new_pc;
                                                                1: pc <= branch;
```

```
                                                    2: pc <= call;
                                                    3: pc <= return;
                                                    4: pc <= pc;
                                                    default: pc <= 3'bxxx;
                                                endcase
                                                    case(stack_sel)
                                                        0:
                                                            begin
                                                                if(enable
== 7) tos <= pc;

        else if (enable == 3) tos<= new_pc;
                                                            end
                                                        1: tos<= enable[0]? nos:
tos;
                                                    endcase
                                                nos<= enable[1]? tos: nos;
                                            end
                                    endcase
                end

endmodule
```

6.1d Register File

```
module reg_file(clk, reset_apply, start_sprite, frame_done, wr, data_in, data_out,
 reg_no, sprite_no, xcoor, ycoor, ready_sprite, stop, wr_sprite, wr_xcoor, wr_ycoor,
reset_done, process_done, interrupt_valid);
input clk, start_sprite, frame_done, wr,  reset_apply, stop;
input [7:0] data_in;
input [6:0] reg_no;
output ready_sprite, wr_xcoor, wr_ycoor, wr_sprite, reset_done, process_done, interrupt_valid;
reg ready_sprite,wr_sprite, wr_xcoor, wr_ycoor, process_done, interrupt_valid;

reg reset_done; //to indicate that blanking the regfile memory is done

output [7:0] data_out;
wire [7:0] sprite_ram_out, xcoor_ram_out, ycoor_ram_out;
reg [7:0] data_out;

reg [4:0] address;
reg [1:0] state;
reg [5:0] counter;

output [4:0] sprite_no;
output [8:0] xcoor;
output [7:0] ycoor;

regram1 sprite_ram(address, wr_sprite, data_in, sprite_ram_out);
regram1 xcoor_ram(address, wr_xcoor, data_in, xcoor_ram_out);
regram1 ycoor_ram(address, wr_ycoor, data_in, ycoor_ram_out);

parameter state_process = 1;
parameter state_waitforsprite = 0;
parameter state_loop = 2;
parameter state_reset = 3;
```

```
always @ (posedge clk)
	begin
		reset_done <=0;
		interrupt_valid <=0;

		if(reset_apply)
			begin
				counter <= 0;
				state<= state_reset;

				ready_sprite <= 0;
				process_done <=0;
				address <=0;
			end
		else case(state)
			state_reset:
				begin
					address<=counter[4:0];
					counter<= counter+1;
					if(counter == 32)
						begin
							reset_done<=1;
							state <= state_process;
							interrupt_valid <=1;
							process_done<=0;
							counter<=0;
						end


				end
			state_process:
				begin
					address <= reg_no[6:2];
					if(stop)
						begin
							state <= state_waitforsprite;
							process_done <=1;
							counter <= 0;
						end
				end

			state_waitforsprite:
				begin
					if(start_sprite)
						begin
							state<= state_loop;

						end
					else if(frame_done)
						begin
							state <= state_process;
							interrupt_valid <=1;
							process_done <=0;
						end
				end
```

```
                state_loop:
                        begin
                                address <=  counter[4:0] ;
                                ready_sprite <=1;
                                counter<= counter+1;
                                if(counter == 32)
                                        begin
                                                state <= state_waitforsprite;
                                                ready_sprite <=0;
                                                counter<=0;
                                        end
                        end

                endcase

        end

                assign sprite_no = sprite_ram_out [4:0];
                assign xcoor[8:1] = xcoor_ram_out;
                assign xcoor[0] = 0;
                assign ycoor = ycoor_ram_out[7:0];


always @(state)
        begin
                 if (state == state_process)
                        begin
                                case(reg_no[1:0])
                                        0:
                                                begin
                                                        wr_sprite <=wr;
                                                        data_out <= sprite_ram_out;
                                                end
                                        1:
                                                begin
                                                        wr_xcoor <= wr;
                                                        data_out <= xcoor_ram_out;
                                                end
                                        2:
                                                begin
                                                        wr_ycoor <= wr;
                                                        data_out <= ycoor_ram_out;
                                                end
                                        default:;
                                endcase

                        end
                        else if (state== state_reset)
                                begin
                                        wr_sprite <=1;
                                        wr_xcoor <=1;
                                        wr_ycoor <=1;
                                end
                        else
                                begin
```

```
                                        data_out <= 0;
                                        wr_sprite <=0;
                                        wr_xcoor <=0;
                                        wr_ycoor <=0;
                                end

        end
endmodule
```

6.2 The Resizer

6.2a Major FSM

```
module majorfsm(clock, reset, start_copy, start_int, start_dec1, start_dec2, busy_copy, busy_int,
busy_dec1, busy_dec2, size, le_size, oe_driver);
input clock, reset, busy_copy, busy_int, busy_dec1, busy_dec2, le_size;
input [2:0] size;
output start_copy, start_int, start_dec1, start_dec2, oe_driver;

reg start_copy, start_int, start_dec1, start_dec2, oe_driver;
reg [2:0] state, M;
reg [1:0] L;

parameter idle = 0;
parameter copy = 1;
parameter wait1 = 2;
parameter int = 3;
parameter wait2 = 4;
parameter dec1 = 5;
parameter wait3 = 6;
parameter dec2 = 7;

always @ (state)
begin
case (state)
        idle: begin     start_copy = le_size;
                        start_int = 0;
                        start_dec1 = 0;
                        start_dec2 = 0;
                        oe_driver = start_copy || start_int || start_dec1 || start_dec2;    end
        copy: begin     start_copy = 0;
                        start_int = 0;
                        start_dec1 = 0;
                        start_dec2 = 0;
                        oe_driver = 1;    end
        wait1: begin    start_copy = 0;
                                start_int = 1;
                                start_dec1 = 0;
                                start_dec2 = 0;
                                oe_driver = 1;    end
        int: begin      start_copy = 0;
                        start_int = 0;
                        start_dec1 = 0;
                        start_dec2 = 0;
                        oe_driver = 1;    end
        wait2: begin    start_copy = 0;
```

```
                                        start_int = 0;
                                        start_dec1 = 1;
                                        start_dec2 = 0;
                                        oe_driver = 1;      end
        dec1: begin         start_copy = 0;
                                start_int = 0;
                                start_dec1 = 0;
                                start_dec2 = 0;
                                oe_driver = 1;      end
        wait3: begin        start_copy = 0;
                                        start_int = 0;
                                        start_dec1 = 0;
                                        start_dec2 = 1;
                                        oe_driver = 1;      end
        dec2: begin         start_copy = 0;
                                start_int = 0;
                                start_dec1 = 0;
                                start_dec2 = 0;
                                oe_driver = 1;      end
        default: begin      start_copy = 1'hx;
                                        start_int = 1'hx;
                                        start_dec1 = 1'hx;
                                        start_dec2 = 1'hx;
                                        oe_driver = 1'hx; end
endcase
end

always @ (posedge clock)
begin
    begin
        case (size)
            3'b000: begin           L <= 2'b01;
                        M <= 3'b001;        end
            3'b001: begin           L <= 2'b10;
                        M <= 3'b011;        end
            3'b010: begin           L <= 2'b01;
                        M <= 3'b010;        end
            3'b011: begin           L <= 2'b01;
                        M <= 3'b011;        end
            3'b100: begin           L <= 2'b01;
                        M <= 3'b100;        end
            3'b101: begin           L <= 2'b01;
                        M <= 3'b110;        end
            default:    begin       L <= 2'bxx;
                         M <= 3'bxxx;       end
        endcase
    end

    if (reset) state <= idle;
    else case (state)
        idle:   begin
                                    if (le_size) state <= copy;
                else state <= state;
            end
        copy:   state <= busy_copy ? state : wait1;
        wait1:  state <= int;
```

```
            int:       state <= busy_int ? state : wait2;
            wait2:   state <= dec1;
            dec1: begin
                                                    if (busy_dec1) state <= state;
                                                    else
                                                            begin
                        if ((M == 3'b100) || (M == 3'b110)) state <= wait3;
                        else state <= idle;
                    end
                                                end
            wait3: state <= dec2;
            dec2: state <= busy_dec2 ? state : idle;
        endcase
end
endmodule
```

6.2b Minor FSM

```
module minorfsm(clock, reset, start_copy, start_int, start_dec1, start_dec2, busy_copy, busy_int,
busy_dec1,
busy_dec2, addr_rom, addr_ram, size, cs_rom, oe_rom, we_rom, cs_ram, oe_ram, rw_ram, data_ram,
oe_driver);
input clock, reset, start_copy, start_int, start_dec1, start_dec2, oe_driver;
input [2:0] size;
output busy_copy, busy_int, busy_dec1, busy_dec2, cs_rom, oe_rom, we_rom, cs_ram, oe_ram, rw_ram;
output [14:0] addr_rom, addr_ram;
inout [7:0] data_ram;

reg busy_copy, busy_int, busy_dec1, busy_dec2, cs_rom, oe_rom, we_rom, cs_ram, oe_ram, rw_ram;
reg [14:0] addr_rom, addr_ram;
reg [1:0] L;
reg [2:0] M;
reg [3:0] state;
reg [1:0] count0; //Used to tell what mode the minorfsm is in, 1 for int, 2 for dec1, 3 for dec2
reg [5:0] count1; //Used to count the entries in each row
reg [14:0] count2; //Generally used to tell what address ram should read from
reg [14:0] count3; //Generally used to tell what address ram should write to
reg [3:0] count4; //Used to tell when all accumulations are done/which coeff. to use
reg [5:0] count5; //Used to see whether a number is divisible by 22 or not
reg [5:0] addr_rom_coeff;
reg [19:0] accumr, accumg, accumb;

wire [7:0] data_rom_coeff, data_ramr, data_ramg, data_ramb;
wire [15:0] multr_result, multg_result, multb_result;
wire [19:0] sxt_multr, sxt_multg, sxt_multb;

parameter idle = 0;
parameter read_rom = 1;
parameter wait1 = 2;
parameter write_ram1 = 3;
parameter wait2 = 4;
parameter read_ram = 5;
parameter wait3 = 6;
parameter write_ram2 = 7;
parameter wait4 = 8;
```

```
rom_coeff rom_coeff1(addr_rom_coeff, data_rom_coeff);
mult multr(data_ramr, data_rom_coeff, multr_result);
mult multg(data_ramg, data_rom_coeff, multg_result);
mult multb(data_ramb, data_rom_coeff, multb_result);

assign data_ram = (state == write_ram2) ? ( (count0==1) ? ((L==2'b01) ? {accumr[13:12], accumg[13:12],
accumb[13:12], 2'b00} :
   {accumr[14:13], accumg[13:12], accumb[13:12], 2'b00}) : ( (count0==2) ? ((M==3'b001) ?
{accumr[13:12], accumg[13:12],
          accumb[13:12], 2'b00} : ( ((M==3'b010)||(M==3'b100)) ? {accumr[11:10], accumg[10:9],
accumb[10:9], 2'b00} :
          {accumr[13:12], accumg[12:11], accumb[12:11], 2'b00} )) : ((M==3'b100) ? {accumr[11:10],
accumg[10:9], accumb[10:9], 2'b00} :
          {accumr[11:10], accumg[10:9], accumb[10:9], 2'b00}))) : 8'bz;

assign data_ramr = {data_ram[7:6], 6'b000000};
assign data_ramg = {data_ram[5:4], 6'b000000};
assign data_ramb = {data_ram[3:2], 6'b000000};
assign sxt_multr = multr_result[15] ? {4'b1111, multr_result} : {4'b0000, multr_result};
assign sxt_multg = multg_result[15] ? {4'b1111, multg_result} : {4'b0000, multg_result};
assign sxt_multb = multb_result[15] ? {4'b1111, multb_result} : {4'b0000, multb_result};

always @ (posedge clock)
begin
      begin
           case (size)
                3'b000: begin          L <= 2'b01;
                              M <= 3'b001;         end
                3'b001: begin          L <= 2'b10;
                              M <= 3'b011;         end
                3'b010: begin          L <= 2'b01;
                              M <= 3'b010;         end
                3'b011: begin          L <= 2'b01;
                              M <= 3'b011;         end
                3'b100: begin          L <= 2'b01;
                              M <= 3'b100;         end
                3'b101: begin          L <= 2'b01;
                              M <= 3'b110;         end
                default:      begin    L <= 2'bxx;
                               M <= 3'bxxx;      end
           endcase
      end

          if (reset) state <= idle;
          else case (state)
                    idle:      begin
                                                  cs_rom <= 1;
                                                  oe_rom <= 1;
                                                  we_rom <= 1;
                                                  cs_ram <= oe_driver;
                                                  oe_ram <= oe_driver;
                                                  rw_ram <= 1;
                                                  count0 <= 0;
                                                  count1 <= 0;
                                                  count2 <= 0;
                                                  count3 <= 0;
```

```
                                    count4 <= 1;
                                    count5 <= 0;
                                    busy_copy <= 0;
                                    busy_int <= 0;
                                    busy_dec1 <= 0;
                                    busy_dec2 <= 0;
                                    addr_rom <= 0;
                                    addr_ram <= 0;
                                    addr_rom_coeff <= 45;
                                    accumr <= 0;
                                    accumg <= 0;
                                    accumb <= 0;
                                    if (start_copy)
     begin
       state <= read_rom;
                                                busy_copy <= start_copy;
                                                addr_ram <= 0;
   cs_rom <= 0;
   oe_rom <= 0;
                                                cs_ram <= 1;
                                                oe_ram <= 1;
   if (L==2'b10)
      begin   addr_rom <= 4096;
              addr_ram <= 1;
      end
end
                                    else if (start_int)
                                            begin
                                                    count0 <= 1;
                                                    count3 <= 16384;
                                                    state <= read_ram;
                                                    busy_int <= start_int;
                                                    addr_ram <= 0;
                                                    cs_ram <= 0;
                                                    oe_ram <= 0;
                                            end
                                    else if (start_dec1)
                                            begin
                                                    count0 <= 2;
                                                    count2 <= 16384;
                                                    state <= read_ram;
                                                    busy_dec1 <= start_dec1;
                                                    addr_ram <= 0;
                                                    cs_ram <= 0;
                                                    oe_ram <= 0;
                                            end
                                    else if (start_dec2)
                                            begin
                                                    count0 <= 3;
                                                    count3 <= 16384;
                                                    state <= read_ram;
                                                    busy_dec2 <= start_dec2;
                                                    addr_ram <= 0;
                                                    cs_ram <= 0;
                                                    oe_ram <= 0;
                                            end
```

```
                                    else state <= state;
                                end
read_rom:       begin       busy_copy <= 1;
                        state <= wait1;
                        cs_rom <= 1;
                        oe_rom <= 1;
                    end
wait1:      begin           busy_copy <= 1;
                    cs_ram <= 0;
                    rw_ram <= 0;
                    state <= write_ram1;
                end
write_ram1:     begin               busy_copy <= 1;
                        state <= wait2;
                        cs_ram <= 1;
                        rw_ram <= 1;
                    end
wait2: begin                        if (L==2'b01)
                        begin
                            if (addr_ram == 4095)
                             begin  state <= idle;
                                            busy_copy <= 0;
                                            count2 <= 0;
                                            count3 <= 0;       end
                            else
                                begin           busy_copy <= 1;
                                    state <= read_rom;
                                    cs_rom <= 0;
                                    oe_rom <= 0;

                                            addr_rom <= addr_rom + 1;

                                            addr_ram <= addr_ram + 1;
                                end
                        end
                    if (L==2'b10)
                        begin
                            if (count2 == 16383)
                             begin  state <= idle;
                                            busy_copy <= 0;
                                            count2 <= 0;
                                            count3 <= 0;       end
                            else
                                begin           busy_copy <= 1;
                                    state <= read_rom;
                                    cs_rom <= 0;
                                    oe_rom <= 0;
                                    count2 <= count2 + 1;
                                    if (count2 < 12287)
                                        begin
                                            addr_rom <= 4096;
                                            if (count3 < 63)
                                                begin
                                                    addr_ram <= addr_ram + 2;
                                                    count3 <= count3 + 1;
                                                end
```

```
                                else if ((count3 >= 63) && (count3 < 191))
                                    begin
                                        addr_ram <= addr_ram + 1;
                                        count3 <= count3 + 1;
                                    end
                                else if (count3 == 191)
                                    begin
                                        addr_ram <= addr_ram + 2;
                                        count3 <= 0;
                                    end
                            end
                        else if (count2 == 12287)
                            begin
                                addr_rom <= 0;
                                addr_ram <= 0;

                                                count3 <= 0;
                            end
                        else if ((count2 > 12287) && (count2 < 16383))
                            begin
                                addr_rom <= addr_rom + 1;
                                if (count3 < 63)
                                    begin
                                        addr_ram <= addr_ram + 2;
                                        count3 <= count3 + 1;
                                    end
                                else if (count3 == 63)
                                    begin
                                        addr_ram <= addr_ram + 130;
                                        count3 <= 0;
                                    end
                            end
                        end
                end
    end
read_ram:       begin
                        accumr <= accumr + sxt_multr;
                        accumg <= accumg + sxt_multg;
                        accumb <= accumb + sxt_multb;
                        cs_ram <= 1;
                        oe_ram <= 1;
                        state <= wait3;
                        busy_int <= (count0 == 1) ? 1 : 0;
                        busy_dec1 <= (count0 == 2) ? 1 : 0;
                        busy_dec2 <= (count0 == 3) ? 1 : 0;
                    end
wait3:  begin
                    cs_ram <= 0;
                    oe_ram <= 0;
                    count4 <= count4 + 1;
                    if (count4 == 9)
                            begin   state <= write_ram2;
                                            rw_ram <= 0;
                                            addr_ram <= count3;
                                            count4 <= 0;    end
                    else    begin   state <= read_ram;
```

```
                                        if (count0 == 1)
                                                begin
                                                        busy_int <= 1;
                                                        if (L == 2'b01)
                                                                begin
                                                                        case (count4)
                                                                                0:
begin   addr_ram <= (count2 < 65) ? 0 : count2 - 65;

        addr_rom_coeff <= ( (count2 < 64) || (count2[5:0] == 6'b000000) ) ? 45: 0;       end
                                                                                1:
begin   addr_ram <= (count2 < 64) ? 0 : count2 - 64;

        addr_rom_coeff <= (count2 < 64) ? 45 : 1;    end
                                                                                2:
begin   addr_ram <= (count2 < 63) ? 0 : count2 - 63;

        addr_rom_coeff <= ( (count2 < 64) || (count2[5:0] == 6'b111111) ) ? 45 : 2;      end
                                                                                3:
begin   addr_ram <= (count2 < 1) ? 0 : count2 - 1;

        addr_rom_coeff <= (count2[5:0] == 6'b000000) ? 45 : 3;        end
                                                                                4:
begin   addr_ram <= count2;

        addr_rom_coeff <= 4;        end
                                                                                5:
begin   addr_ram <= count2 + 1;

        addr_rom_coeff <= (count2[5:0] == 6'b111111) ? 45 : 5;        end
                                                                                6:
begin   addr_ram <= count2 + 63;

        addr_rom_coeff <= ( (count2[5:0] == 6'b000000) ||

                ((count2 >= 4032) && (count2 < 4096)) ) ? 45 : 6;     end
                                                                                7:
begin   addr_ram <= count2 + 64;

        addr_rom_coeff <= ((count2 >= 4032) && (count2 < 4096)) ? 45 : 7;    end
                                                                                8:
begin   addr_ram <= count2 + 65;

        addr_rom_coeff <= ( ((count2 >= 4032) && (count2 < 4096)) ||

                (count2[5:0] == 6'b111111) ) ? 45 : 8;          end
                                                                                default:
begin   addr_ram <= 14'hxxxx;

                        addr_rom_coeff <= 6'hxx; end
                                                                        endcase
                                                                end
                                                        else if (L == 2'b10)
                                                                begin
                                                                        case (count4)
```

```
                                                                    0:
begin   addr_ram <= (count2 < 129) ? 0 : count2 - 129;

        addr_rom_coeff <= ( (count2 < 128) || (count2[6:0] == 7'b0000000) ) ? 45 : 9;   end
                                                                    1:
begin   addr_ram <= (count2 < 128) ? 0 : count2 - 128;

        addr_rom_coeff <= (count2 < 128) ? 45 : 10;                 end
                                                                    2:
begin   addr_ram <= (count2 < 127) ? 0 : count2 - 127;

        addr_rom_coeff <= ( (count2 < 128) || (count2[6:0] == 7'b1111111) ) ? 45 : 11; end
                                                                    3:
begin   addr_ram <= (count2 < 1) ? 0 : count2 - 1;

        addr_rom_coeff <= (count2[6:0] == 7'b0000000) ? 45 : 12;    end
                                                                    4:
begin   addr_ram <= count2;

        addr_rom_coeff <= 13;     end
                                                                    5:
begin   addr_ram <= count2 + 1;

        addr_rom_coeff <= (count2[6:0] == 7'b1111111) ? 45 : 14;    end
                                                                    6:
begin   addr_ram <= count2 + 127;

        addr_rom_coeff <= ( (count2[6:0] == 7'b0000000) ||

                ((count2 >= 16256) && (count2 < 16384)) ) ? 45 : 15;        end
                                                                    7:
begin   addr_ram <= count2 + 128;

        addr_rom_coeff <= ((count2 >= 16256) && (count2 < 16384)) ? 45 : 16;        end
                                                                    8:
begin   addr_ram <= count2 + 129;

        addr_rom_coeff <= ( ((count2 >= 16256) && (count2 < 16384)) ||

                (count2[6:0] == 7'b1111111) ) ? 45 : 17;    end
                                                                    default:
begin   addr_ram <= 14'hxxxx;

                        addr_rom_coeff <= 6'hxx; end

                                                                endcase

                                                            end
                                                    end
                                            else if (count0 == 2)
                                                    begin
                                                            busy_dec1 <= 1;
                                                            if (M == 1)
                                                                    begin
                                                                            case (count4)
```

```
                                                                    0:
begin   addr_ram <= count2 - 65;

        addr_rom_coeff <= ( (count2 < 16448) || (count2[5:0] == 6'b000000) ) ? 45 : 18; end
                                                                    1:
begin   addr_ram <= count2 - 64;

        addr_rom_coeff <= (count2 < 16448) ? 45 : 19;          end
                                                                    2:
begin   addr_ram <= count2 - 63;

        addr_rom_coeff <= ( (count2 < 16448) || (count2[5:0] == 6'b111111) ) ? 45 : 20; end
                                                                    3:
begin   addr_ram <= count2 - 1;

        addr_rom_coeff <= (count2[5:0] == 6'b000000) ? 45 : 21;       end
                                                                    4:
begin   addr_ram <= count2;

        addr_rom_coeff <= 22;      end
                                                                    5:
begin   addr_ram <= count2 + 1;

        addr_rom_coeff <= (count2[5:0] == 6'b111111) ? 45 : 23;       end
                                                                    6:
begin   addr_ram <= count2 + 63;

        addr_rom_coeff <= ( (count2[5:0] == 6'b000000) ||

                ((count2 >= 20416) && (count2 < 20480)) )? 45 : 24;end
                                                                    7:
begin   addr_ram <= count2 + 64;

        addr_rom_coeff <= ((count2 >= 20416) && (count2 < 20480)) ? 45 : 25;          end
                                                                    8:
begin   addr_ram <= count2 + 65;

        addr_rom_coeff <= ( ((count2 >= 20416) && (count2 < 20480)) ||

                (count2[5:0] == 6'b111111) ) ? 45 : 26;         end
                                                                    default:
begin   addr_ram <= 14'hxxxx;

        addr_rom_coeff <= 6'hxx; end
                                                                        endcase
                                                                    end
                                                                else if ((M == 2) || (M == 4))
                                                                    begin
                                                                        case (count4)
                                                                    0:
begin   addr_ram <= count2 - 65;

        addr_rom_coeff <= ( (count2 < 16448) || (count2[5:0] == 6'b000000) ) ? 45 : 27; end
                                                                    1:
begin   addr_ram <= count2 - 64;
```

```
        addr_rom_coeff <= (count2 < 16448) ? 45 : 28;         end
                                                                                2:
begin   addr_ram <= count2 - 63;

        addr_rom_coeff <= ( (count2 < 16448) || (count2[5:0] == 6'b111111) ) ? 45 : 29; end
                                                                                3:
begin   addr_ram <= count2 - 1;

        addr_rom_coeff <= (count2[5:0] == 6'b000000) ? 45 : 30;       end
                                                                                4:
begin   addr_ram <= count2;

        addr_rom_coeff <= 31;     end
                                                                                5:
begin   addr_ram <= count2 + 1;

        addr_rom_coeff <= (count2[5:0] == 6'b111111) ? 45 : 32;       end
                                                                                6:
begin   addr_ram <= count2 + 63;

        addr_rom_coeff <= ( (count2[5:0] == 6'b000000) ||

                ((count2 >= 20416) && (count2 < 20480)) )? 45 : 33;end
                                                                                7:
begin   addr_ram <= count2 + 64;

        addr_rom_coeff <= ((count2 >= 20416) && (count2 < 20480)) ? 45 : 34;          end
                                                                                8:
begin   addr_ram <= count2 + 65;

        addr_rom_coeff <= ( ((count2 >= 20416) && (count2 < 20480)) ||

                (count2[5:0] == 6'b111111) ) ? 45 : 35;       end
                                                                                default:
begin   addr_ram <= 14'hxxxx;

        addr_rom_coeff <= 6'hxx; end
                                                                        endcase
                                                                end
                                                        else if ((M == 3) || (M == 6))
                                                                begin   if (L == 2'b01)
                                                                        begin
                                                                        case (count4)
                                                                                0:
begin   addr_ram <= count2 - 65;

        addr_rom_coeff <= ( (count2 < 16448) || (count2[5:0] == 6'b000000) ) ? 45 : 36; end
                                                                                1:
begin   addr_ram <= count2 - 64;

        addr_rom_coeff <= (count2 < 16448) ? 45 : 37;         end
                                                                                2:
begin   addr_ram <= count2 - 63;

        addr_rom_coeff <= ( (count2 < 16448) || (count2[5:0] == 6'b111111) ) ? 45 : 38; end
```

```
                                                                    3:
begin   addr_ram <= count2 - 1;

        addr_rom_coeff <= (count2[5:0] == 6'b000000) ? 45 : 39;     end
                                                                    4:
begin   addr_ram <= count2;

        addr_rom_coeff <= 40;     end
                                                                    5:
begin   addr_ram <= count2 + 1;

        addr_rom_coeff <= (count2[5:0] == 6'b111111) ? 45 : 41;     end
                                                                    6:
begin   addr_ram <= count2 + 63;

        addr_rom_coeff <= ( (count2[5:0] == 6'b000000) ||

                ((count2 >= 20416) && (count2 < 20480)) ) ? 45 : 42;          end
                                                                    7:
begin   addr_ram <= count2 + 64;

        addr_rom_coeff <= ((count2 >= 20416) && (count2 < 20480)) ? 45 : 43;          end
                                                                    8:
begin   addr_ram <= count2 + 65;

        addr_rom_coeff <= ( ((count2 >= 20416) && (count2 < 20480)) ||

                (count2[5:0] == 6'b111111) ) ? 45 : 44;       end
                                                                    default:
begin   addr_ram <= 14'hxxxx;

        addr_rom_coeff <= 6'hxx; end
                                                            endcase
                                                            end
                                                    else if (L == 2'b10)
                                                            begin
                                                            case (count4)
                                                                    0:
begin   addr_ram <= count2 - 129;

        addr_rom_coeff <= ( (count2 < 16512) || (count2[6:0] == 7'b0000000) ) ? 45 : 36;
end
                                                                    1:
begin   addr_ram <= count2 - 128;

        addr_rom_coeff <= (count2 < 16512) ? 45 : 37;       end
                                                                    2:
begin   addr_ram <= count2 - 127;

        addr_rom_coeff <= ( (count2 < 16512) || (count2[6:0] == 7'b1111111) ) ? 45 : 38;
end
                                                                    3:
begin   addr_ram <= count2 - 1;

        addr_rom_coeff <= (count2[6:0] == 7'b0000000) ? 45 : 39;     end
```

```
                                                                    4:
begin   addr_ram <= count2;
        addr_rom_coeff <= 40;     end
                                                                    5:
begin   addr_ram <= (count2 == 32767) ? 0 : count2 + 1;
        addr_rom_coeff <= (count2[6:0] == 7'b1111111) ? 45 : 41;     end
                                                                    6:
begin   addr_ram <= (count2 > 32640) ? 0 : count2 + 127;
        addr_rom_coeff <= ( (count2[6:0] == 7'b0000000) ||
                ((count2 >= 32640) && (count2 < 32768)) ) ? 45 : 42;          end
                                                                    7:
begin   addr_ram <= (count2 > 32639) ? 0 : count2 + 128;
        addr_rom_coeff <= ((count2 >= 32640) && (count2 < 32768)) ? 45 : 43;          end
                                                                    8:
begin   addr_ram <= (count2 > 32638) ? 0 : count2 + 129;
        addr_rom_coeff <= ( ((count2 >= 32640) && (count2 < 32768)) ||
                (count2[6:0] == 7'b1111111) ) ? 45 : 44;      end
                                                                    default:
begin   addr_ram <= 14'hxxxx;
        addr_rom_coeff <= 6'hxx; end
                                                                endcase
                                                                end
                                                        end
                                                end
                                                else if (count0 == 3)
                                                        begin   busy_dec2 <= 1;
                                                                if (M == 4)
                                                                begin
                                                                case (count4)
                                                                0:
begin   addr_ram <= (count2 < 33) ? 0 : count2 - 33;
        addr_rom_coeff <= ( (count2 < 32) || (count2[4:0] == 5'b00000) ) ? 45 : 27;      end
                                                                1:
begin   addr_ram <= (count2 < 32) ? 0 : count2 - 32;
        addr_rom_coeff <= (count2 < 32) ? 45 : 28;  end
                                                                2:
begin   addr_ram <= (count2 < 31) ? 0 : count2 - 31;
        addr_rom_coeff <= ( (count2 < 32) || (count2[4:0] == 5'b11111) ) ? 45 : 29;      end
                                                                3:
begin   addr_ram <= (count2 < 1) ? 0 : count2 - 1;
        addr_rom_coeff <= (count2[4:0] == 5'b00000) ? 45 : 30;        end
```

```
                                                                    4:
begin   addr_ram <= count2;
        addr_rom_coeff <= 31;     end
                                                                    5:
begin   addr_ram <= count2 + 1;
        addr_rom_coeff <= (count2[4:0] == 5'b11111) ? 45 : 32;        end
                                                                    6:
begin   addr_ram <= count2 + 31;
        addr_rom_coeff <= ( (count2[4:0] == 5'b00000) ||
                ((count2 >= 992) && (count2 < 1024)) ) ? 45 : 33;     end
                                                                    7:
begin   addr_ram <= count2 + 32;
        addr_rom_coeff <= ((count2 >= 992) && (count2 < 1024)) ? 45 : 34;   end
                                                                    8:
begin   addr_ram <= count2 + 33;
        addr_rom_coeff <= ( ((count2 >= 992) && (count2 < 1024)) ||
                (count2[4:0] == 5'b11111) ) ? 45 : 35;          end
                                                                    default:
begin   addr_ram <= 14'hxxxx;
        addr_rom_coeff <= 6'hxx; end
                                                            endcase
                                                            end
                                                    else if (M == 6)
                                                            begin
                                                            case (count4)
                                                                    0:
begin   addr_ram <= (count2 < 23) ? 0 : count2 - 23;
        addr_rom_coeff <= ( (count2 < 22) || (count5 == 0) ) ? 45 : 36;end
                                                                    1:
begin   addr_ram <= (count2 < 22) ? 0 : count2 - 22;
        addr_rom_coeff <= (count2 < 22) ? 45 : 37;  end
                                                                    2:
begin   addr_ram <= (count2 < 21) ? 0 : count2 - 21;
        addr_rom_coeff <= ( (count2 < 22) || (count5 == 20) ) ? 45 : 38;      end
                                                                    3:
begin   addr_ram <= (count2 < 1) ? 0 : count2 - 1;
        addr_rom_coeff <= (count5 == 0) ? 45 : 39;  end
                                                                    4:
begin   addr_ram <= count2;
        addr_rom_coeff <= 40;     end
                                                                    5:
begin   addr_ram <= count2 + 1;
```

```
                addr_rom_coeff <= (count5 == 20) ? 45 : 41;             end
                                                                                6:
        begin   addr_ram <= count2 + 21;
                addr_rom_coeff <= ( (count5 == 0) ||
                        ((count2 >= 462) && (count2 < 482)) ) ? 45 : 42; end
                                                                                7:
        begin   addr_ram <= count2 + 22;
                addr_rom_coeff <= ((count2 >= 462) && (count2 < 482)) ? 45 : 43;     end
                                                                                8:
        begin   addr_ram <= count2 + 23;
                addr_rom_coeff <= ( ((count2 >= 462) && (count2 < 482)) ||
                        (count5 == 20) )  ? 45 : 44;            end
                                                                                default:
        begin   addr_ram <= 14'hxxxx;
                addr_rom_coeff <= 6'hxx; end
                                                                        endcase
                                                                        end
                                                                end
                                                        end
                                                end
                write_ram2:     begin   state <= wait4;
                                                        accumr <= 0;
                                                        accumg <= 0;
                                                        accumb <= 0;
                                                        cs_ram <= 1;
                                                        oe_ram <= 1;
                                                        rw_ram <= 1;
                                                        busy_int <= (count0 == 1) ? 1 : 0;
                                                        busy_dec1 <= (count0 == 2) ? 1 : 0;
                                                        busy_dec2 <= (count0 == 3) ? 1 : 0;
                                                end
                wait4:  begin
                                                count3 <= count3 + 1;
                                                if (count0 == 1)
                                                        begin   if (L == 2'b01)
                                                                                begin   if (count2 ==
4095)
                                                                                        begin   state <=
idle;

        busy_int <= 0;


        count0 <= 0;

        count2 <= 0;

        count3 <= 0;
```

```
            count4 <= 1;        end
                                                                                    else
begin       count2 <= count2 + 1;
                    state <= wait3;
                    busy_int <= 1;      end
                                                                        end
                                                                else if (L == 2'b10)
                                                                        begin       if (count2 ==
16383)
                                                                                    begin       state <=
idle;
            busy_int <= 0;
            count0 <= 0;
            count2 <= 0;
            count3 <= 0;
            count4 <= 1;        end
                                                                                    else
begin       count2 <= count2 + 1;

                    state <= wait3;
                    busy_int <= 1;      end
                                                                        end
                                                            end
                                                    else if (count0 == 2)
                                                            begin       if (M == 1)
                                                                        begin       if (count3 ==
4095)
                                                                                    begin       state <=
idle;
            busy_dec1 <= 0;
            count0 <= 0;
            count2 <= 0;
            count3 <= 0;
            count4 <= 1;        end
                                                                                    else
begin       count2 <= count2 + 1;
                    state <= wait3;
                    busy_dec1 <= 1;  end
                                                                        end
```

```
                                                    else if ((M == 2) || (M == 4))
                                                        begin     if (count3 ==
1023)
                                                                  begin     state <=
idle;
            busy_dec1 <= 0;
            count0 <= 0;
            count1 <= 0;
            count2 <= 0;
            count3 <= 0;
            count4 <= 1;        end
                                                                                    else
            begin     state <= wait3;
                      busy_dec1 <= 1;
                      if (count1 < 31)
                                begin     count1 <= count1 + 1;
                                          count2 <= count2 + 2;       end
                      else if (count1 == 31)
                                begin     count1 <= 0;
                                          count2 <= count2 + 66;      end
            end
                                                        end
                                                    else if ((M == 3) || (M == 6))
                                                                  begin     if (L ==
2'b01)
            begin     if (count3 == 483)
            begin     state <= idle;
                                busy_dec1 <= 0;
                                count0 <= 0;
                                count1 <= 0;
                                count2 <= 0;
                                count3 <= 0;
                                count4 <= 1;        end
```

```
        else    begin   state <= wait3;
                        busy_dec1 <= 1;
                        if (count1 < 21)
                                begin   count1 <= count1 + 1;
                                        count2 <= count2 + 3;   end
                        else if (count1 == 21)
                                begin   count1 <= 0;
                                        count2 <= count2 + 129; end
                end
        end
                                                                        else if
(L == 2'b10)
        begin if (count3 == 1848)
        begin   state <= idle;
                        busy_dec1 <= 0;
                        count0 <= 0;
                        count1 <= 0;
                        count2 <= 0;
                        count3 <= 0;
                        count4 <= 1;    end
        else    begin   state <= wait3;
                        busy_dec1 <= 1;
                        if (count1 < 42)
                                begin   count1 <= count1 + 1;
                                        count2 <= count2 + 3;   end
                        else if (count1 == 42)
                                begin   count1 <= 0;
                                        count2 <= count2 + 258; end
                end
```

```
                                                            end
                                                    end
                                            end
                                    else if (count0 == 3)
                                        begin   if (M == 4)
                                                    begin   if (count3 ==
16639)
                                                        begin   state <=
idle;
    busy_dec2 <= 0;
    count0 <= 0;
    count1 <= 0;
    count2 <= 0;
    count3 <= 0;
    count4 <= 1;
    count5 <= 0;    end
                                                            else
    begin   state <= wait3;
        busy_dec2 <= 1;
        if (count1 < 15)
            begin   count1 <= count1 + 1;
                    count2 <= count2 + 2;   end
        else if (count1 == 15)
            begin   count1 <= 0;
                    count2 <= count2 + 34;  end
    end
                                                    end
                                                else if (M == 6)
                                                    begin   if (count3 ==
16504)
                                                        begin   state <=
idle;
    busy_dec2 <= 0;
    count0 <= 0;
    count1 <= 0;
    count2 <= 0;
```

```
            count3 <= 0;

            count4 <= 1;

            count5 <= 0;        end
                                                                                    else
            begin     state <= wait3;

                      busy_dec2 <= 1;

                      if (count1 < 10)

                                begin     count1 <= count1 + 1;

                                                    count2 <= count2 + 2;

                                                    count5 <= count5 + 2;       end

                      else if (count1 == 10)

                                begin     count1 <= 0;

                                                    count2 <= count2 + 24;

                                                    count5 <= 0;                        end

            end
                                                                                    end
                                                                  end
                                          end
                      default: state <= state;
            endcase
end
endmodule
```

6.2c Overall Resizer Module

```
module finalproj(clock, reset, addr_rom, addr_ram, size, le_size, cs_rom, oe_rom, we_rom,
            cs_ram, oe_ram, rw_ram, data_ram);
input clock, reset, le_size;
input [2:0] size;
output cs_rom, oe_rom, we_rom, cs_ram, oe_ram, rw_ram;
output [14:0] addr_rom, addr_ram;
inout [7:0] data_ram;

wire start_copy, start_int, start_dec1, start_dec2, busy_copy, busy_int, busy_dec1, busy_dec2;

majorfsm majorfsm1(clock, reset, start_copy, start_int, start_dec1, start_dec2, busy_copy,
            busy_int, busy_dec1, busy_dec2, size, le_size, oe_driver);
minorfsm minorfsm1(clock, reset, start_copy, start_int, start_dec1, start_dec2, busy_copy,
            busy_int, busy_dec1, busy_dec2, addr_rom, addr_ram, size, cs_rom, oe_rom, we_rom, cs_ram,
            oe_ram, rw_ram, data_ram, oe_driver);

endmodule
```

6.2d Filter Coefficients

```
WIDTH = 8;  % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE %
DEPTH = 64;  % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL VALUE %


ADDRESS_RADIX = HEX;  % Address and data radixes are optional, default is hex %
DATA_RADIX = HEX;     % Valid radixes = BIN,DEC,HEX or OCT  %

CONTENT BEGIN
        00      :       00;  % L=1 %
        01      :       00;
        02      :       00;
        03      :       00;
        04      :       40;
        05      :       00;
        06      :       00;
        07      :       00;
        08      :       00;
        09      :       15;  % L=2 %
        0A      :       00;
        0B      :       15;
        0C      :       2A;
        0D      :       54;
        0E      :       2A;
        0F      :       15;
        10      :       00;
        11      :       15;
        12      :       00;  % M=1 %
        13      :       00;
        14      :       00;
        15      :       00;
        16      :       40;
        17      :       00;
        18      :       00;
        19      :       00;
        1A      :       00;
        1B      :       06;  % M=2 %
        1C      :       0A;
        1D      :       06;
        1E      :       06;
        1F      :       20;
        20      :       06;
        21      :       06;
        22      :       0A;
        23      :       06;
        24      :       05;  % M=3 %
        25      :       0C;
        26      :       05;
        27      :       05;
        28      :       20;
        29      :       05;
        2A      :       05;
        2B      :       0C;
        2C      :       05;
        2D      :       00;       % start 0 values %
        2E      :       00;
```

```
        2F      :       00;
        30      :       00;
        31      :       00;
        32      :       00;
        33      :       00;
        34      :       00;
        35      :       00;
        36      :       00;
        37      :       00;
        38      :       00;
        39      :       00;
        3A      :       00;
        3B      :       00;
        3C      :       00;
        3D      :       00;
        3E      :       00;
        3F      :       00;

END;
```

6.3 Video Controller Unit

6.3a 6bit Reg

```
module 6bitreg(clk,rgb_in,rgb_out);

input clk,reset;
input [5:0] rgb_in;
output [5:0] rgb_out;

always @ (posedge clk)
begin
        rgb_out[5:0] <= rgb_in[5:0];
end

endmodule
```

6.3b Line Reg

```
module line_reg (clk,reset,width,x_coor,frame_done,xt,ld,rom_addr_in,rom_addr_out,x_hit);

input clk, reset;
input frame_done;
input [8:0] width,x_coor,xt;
input [18:0] rom_addr_in;
output [18:0] rom_addr_out;
input ld;
output x_hit;


reg [8:0] w_reg;
reg [8:0] x_reg;
reg [18:0] rom_addr_out;
reg x_hit;

//the "commented out" portions below will be put in another module.
```

```
//assign rom_addr = mem_addr + (yt - y_coor) * width - x_coor;

//assign y_hit = (yt <= y_coor + height) && (yt >= y_coor);

always @ (posedge clk) begin

if ((xt >= x_reg) && (xt < x_reg + w_reg)) x_hit <= 1;
        else x_hit <= 0; end

//assign x_hit = (xt >= x_reg) && (xt < x_reg + w_reg); //x_coor is temporary and only appears when ld is
enabled.

          //the 2nd part is xt "<" NOT "<=" because then add one extra pixel
always @ (posedge clk)                                                                  //because the 1st
part is xt >= x_reg;
begin
        if (reset || frame_done) begin //when reset, want to clear the registers, x_reg want to be large so
won't accidentally get a xhit.
                w_reg <= 0;
                x_reg <= 9'b111111111;
                rom_addr_out <= 0;
        end else
        if (ld) begin
                w_reg <= width; //store the following 3 lines into registers to hold/remember the values,
when finding xhit,etc.
                x_reg <= x_coor;
                rom_addr_out <= rom_addr_in;
        end
end

endmodule
```

6.3c Overall

```
module overall
(clk,reset,ready_sprite,sprite_no,size,x_coor_in,y_coor_in,process_done,y_hit,count,rom_addr_in,
rom_addr_out, width,srt_sprite,frame_done,xt,yt, hsync, vsync,rgb_in, rgb_out,oe1,oe2,oe3,oe4,ld1,ld2,
ld3,ld4,ld5,ld6,ld7,ld8,ld9,ld10,enemy_enable,hreset,x_hit1,x_hit2,x_hit3,x_hit4);
input clk,reset,ready_sprite,process_done;
//input [18:0] mem_addr;
input [4:0] sprite_no;
input [8:0] x_coor_in;
input [7:0] y_coor_in;
output [5:0] rgb_out;
output[8:0] xt;
output[9:0] yt;
output [8:0] width;
input [2:0] size;
input [5:0] rgb_in;
output [18:0] rom_addr_in,rom_addr_out;
output srt_sprite,frame_done,y_hit,hsync,
vsync,oe1,oe2,oe3,oe4,ld1,ld2,ld3,ld4,ld5,ld6,ld7,ld8,ld9,ld10*/,hreset,x_hit1,x_hit2,x_hit3,x_hit4;
output [3:0] count;
reg ld1,ld2,ld3,ld4,ld5, ld6,ld7,ld8,ld9,ld10;

reg [4:0] sprite_no_reg;
```

```
reg [8:0] x_coor;
reg [7:0] y_coor;

wire [18:0] rom_addr_out1, rom_addr_out2, rom_addr_out3, rom_addr_out4, rom_addr_out5;
wire [18:0] rom_addr_out6, rom_addr_out7, rom_addr_out8, rom_addr_out9, rom_addr_out10;

wire ld,hblankon,vblankon;
wire [8:0] width,xt;
wire [7:0] height;
wire [9:0] yt;
wire [4:0] sprite_no;
wire [18:0] mem_addr;
reg srt_sprite;

reg [18:0] rom_addr_in;

//the only thing that is changing when you're sweeping across the screen in the above line is yt. Note that
y_coor
//(from the game controller) assumes that the vertical is 240. yt from sync_gen goes to 480. Also my sprites
in the
//roms are assuming that vertical is 240. Therefore, inorder to have the double the lines to 480, you divide
yt by 2
//or shift one bit. doing that means i.e. when yt is at 0, it's 0. and when yt is at 1, also 0. so you get same
//stuff from rom.

always @ (posedge clk) begin
srt_sprite <=0;
if (yt < 479 && process_done && hblankon)
        srt_sprite <= 1;
 end

assign frame_done = vblankon; //at reset, vblankon is high therefore frame_done is high, so lynne can
process. (see sync_gen)

assign y_hit = ready_sprite? (yt[9:1] >= y_coor) && (yt[9:1] <= y_coor + height) && !(sprite_no_reg==0):
0; //sprite_no = 0 means no data in sprite.

//reg y_hit_reg;
reg [3:0] count;

line_reg line_reg1(clk,srt_sprite,width,x_coor,frame_done,xt,ld1,rom_addr_in,rom_addr_out1,x_hit1);
//have srt_sprite as reset, because have to
line_reg line_reg2(clk,srt_sprite,width,x_coor,frame_done,xt,ld2,rom_addr_in,rom_addr_out2,x_hit2);
//clear the registers each time you go to a new
line_reg line_reg3(clk,srt_sprite,width,x_coor,frame_done,xt,ld3,rom_addr_in,rom_addr_out3,x_hit3);
//new line
line_reg line_reg4(clk,srt_sprite,width,x_coor,frame_done,xt,ld4,rom_addr_in,rom_addr_out4,x_hit4);
line_reg line_reg5(clk,srt_sprite,width,x_coor,frame_done,xt,ld5,rom_addr_in,rom_addr_out5,x_hit5);
line_reg line_reg6(clk,srt_sprite,width,x_coor,xt,ld6,rom_addr_in,rom_addr_out6,x_hit6);
line_reg line_reg7(clk,srt_sprite,width,x_coor,xt,ld7,rom_addr_in,rom_addr_out7,x_hit7);
line_reg line_reg8(clk,srt_sprite,width,x_coor,xt,ld8,rom_addr_in,rom_addr_out8,x_hit8);
line_reg line_reg9(clk,srt_sprite,width,x_coor,xt,ld9,rom_addr_in,rom_addr_out9,x_hit9);
line_reg line_reg10(clk,srt_sprite,width,x_coor,xt,ld10,rom_addr_in,rom_addr_out10,x_hit10);

sixbitreg mysixbitreg(clk,rgb_in,rgb_out);
```

```
sprite_table sp_table(.sprite_no(sprite_no), .clk(clk),
.size(size),.mem_addr(mem_addr),.width(width),.height(height));
sync_gen mysync_gen(clk,reset,hsync,vsync,hblankon,vblankon,xt,yt,hreset);



always @ (mem_addr or yt or y_coor or width or x_coor or count) begin
rom_addr_in[16:0] <= mem_addr[16:0] + ((yt[9:1] - y_coor) * width) - x_coor; //not width - 1 (which is
changed in line_reg)
rom_addr_in[18:17] <= mem_addr[18:17];
                        //changing "width" here changes the address getting in the rom.

ld1 <= (count==0) & y_hit;//before had count start at 1 and go to 10, but then the 1st sprite that gets
ld2 <= (count==1) & y_hit;//hit, would not be loaded, cause count isn't in effect, until clk cycle after,
ld3 <= (count==2) & y_hit;//but by then the next sprite will come up since lynne gives me one every clk
cycle
ld4 <= (count==3) & y_hit;//for 32 clk cycles. to solve the problem start at count equals 0.
ld5 <= (count==4) & y_hit;
ld6 <= (count==5) & y_hit;
ld7 <= (count==6) & y_hit;
ld8 <= (count==7) & y_hit;
ld9 <= (count==8) & y_hit;
ld10 <= (count==9) & y_hit;

end

always @ (posedge clk) begin
x_coor <= x_coor_in; //make everything in rom_addr_in above delayed by one clk cycle (i.e.
x_coor,y_coor,sprite_no)
y_coor <= y_coor_in;
sprite_no_reg <= sprite_no;

if (srt_sprite)
        count <= 0;
else if (ready_sprite && y_hit)
        count <= count + 1;

end
//end of clock


//reg [18:0]rom_addr; //rom_addr defaults to a wire if don't declare it as a register and just one bit. o.w. if
>1 bit, declare as wire.
reg [18:0] rom_addr_out;

//always @ (x_hit1 or x_hit2 or x_hit3 or x_hit4 or x_hit5) /*or x_hit6 or x_hit7 or x_hit8 or x_hit9 or
x_hit10)*/  begin
 always @ (posedge clk) begin
        if (x_hit1)
                rom_addr_out<= rom_addr_out1 + xt;
        else if (x_hit2)
                rom_addr_out <= rom_addr_out2 + xt;
        else if (x_hit3)
                rom_addr_out <= rom_addr_out3 + xt;
        else if (x_hit4)
                rom_addr_out <= rom_addr_out4 + xt;
```

```
        else if (x_hit5)
                rom_addr_out <= rom_addr_out5 + xt;
        else if (x_hit6)
                rom_addr_out <= rom_addr_out6 + xt;
        else if (x_hit7)
                rom_addr_out <= rom_addr_out7 + xt;
        else if (x_hit8)
                rom_addr_out <= rom_addr_out8 + xt;
        else if (x_hit9)
                rom_addr_out <= rom_addr_out9 + xt;
        else if (x_hit10)
                rom_addr_out <= rom_addr_out10 + xt;*/

        else begin rom_addr_out[16:0] <= xt+yt[9:1]*320; //if no hits, pixel is background
                  rom_addr_out[18:17] <= 2; end


end

/*reg oe;
reg oe1,oe2,oe3,oe4;

always @ (posedge clk) begin ' //the following was initially used to register the outputs of the
                                                        //pixel data because there were many lines
on the screen, which was
                                                        //initially thought of as a timing problem
of the video controller.
                                                        //it turned out that it was the game
controller's problem, and the
                                                        //registering of the outputs was not as

neccesary as once thought.
if (rom_addr_out[18:17] == 0)  begin
        oe1 <= 0; oe2 <= 1; oe3 <= 1; oe4 <= 1;  end
if (rom_addr_out[18:17] == 1)  begin
        oe1 <= 1; oe2 <= 0; oe3 <= 1; oe4 <= 1;  end
if (rom_addr_out[18:17] == 2)  begin
        oe1 <= 1; oe2 <= 1; oe3 <= 0; oe4 <= 1;  end
if (rom_addr_out[18:17] == 3)  begin
        oe1 <= 1; oe2 <= 1; oe3 <= 1; oe4 <= 0;  end

end */

assign oe1 = (rom_addr_out[18:17] == 0)? 0:1;
assign oe2 =  (rom_addr_out[18:17] ==1)? 0:1;
assign oe3 = (rom_addr_out[18:17] ==2)? 0:1;
assign oe4 = (rom_addr_out[18:17] ==3)? 0:1;


//end //
/*always @(rom_addr_in)
rom_addr_out <= rom_addr_in;*/ //did these 2 commented lines out before because we commented out the
whole x_hit block
                                                        //block of code and the block of
instantiations of line_reg code because we were trying
                                                        //to figure out something that was wrong
(not requiring those codes) and compilation
```

```
                                                                    //was taking forever, that why we did this
adhoc method.
//wire [2:0] size;
output enemy_enable;
//assign enemy_enable = (y_coor > height + yt) && (sprite_no == 6);
//assign size = ((yt == y_coor + height) && sprite_no == 6) ? size + 1
assign enemy_enable = (yt == y_coor + height + 1) && (sprite_no == 6);


endmodule
```

6.3d Sprite_Table

```
module sprite_table(sprite_no, clk, size, mem_addr, width, height);
input [3:0] sprite_no;
input clk;
input [2:0] size;

output [18:0] mem_addr;

output [8:0] width;
output [7:0] height;

reg [18:0] mem_addr, mem_addr_size;

reg [8:0] width, width_size;
reg [7:0] height, height_size;

always @(posedge clk) begin

case (sprite_no)
15: //road frame 1
begin
mem_addr[18:17] <= 2'b00;
mem_addr[16:0] <= 0;
//width <= 4;
//height <= 3;
width <= 240;
height <= 240;
end
14: //road frame 2

begin
mem_addr [18:17] <= 2'b00;
mem_addr[16:0] <= 57600;
width <= 240;
height <= 240;
end
13: //road frame 3
begin
mem_addr[18:17] <= 2'b01;
mem_addr[16:0] <= 0;
//mem_addr <= 115201;
width <= 240;
height <= 240;
end
```

```
12: //road frame 4
begin
mem_addr[18:17] <= 2'b01;
mem_addr[16:0] <= 57600;
//mem_addr <= 172801;
width <= 240;
height <= 240;
end
11: //powerup frame
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 76800;
//mem_addr <= 94604;
width <= 12;
height <= 12;
end
10: //enemy frame
begin
mem_addr <= mem_addr_size;
width <= width_size;
height <= height_size;
end
9: //bullet frame
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 77952; // 1205;
//mem_addr <= 78005;
width <= 7;
height <= 7;
end
8: //log frame
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 78001; // 1009;
//mem_addr <= 77809;
width <= 14;
height <= 14;
end
7: //mit frame
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 78197;
//mem_addr <= 90604;
width <= 80;
height <= 50;

end
6: //beaver frame
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 76944;
//mem_addr <= 76801;
width <= 42;
height <= 24;
end
5: //life frame #1
```

```
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 92872;
//mem_addr <= 78054;
width <= 45;

height <= 15;
end
4: //life frame #2
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 92422; //1929;
//mem_addr <= 78729;
width <= 30;
height <= 15;
end
3: //life frame #3
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 92197; //2379;
//mem_addr <= 79179;
width <= 15;
height <= 15;
end
/*14: //score digit: 0  //the following commented out lines got deleted because
begin                                                    //we opted out of displaying the score in the end.
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 2604; //
//mem_addr <= 79404;
width <= 8;
height <= 15;
end
15: //score digit: 1
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 2724;
//mem_addr <= 79524;
width <= 8;
height <= 15;
end
16: //score digit: 2
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 2844;
//mem_addr <= 79644;
width <= 8;
height <= 15;
end
17: //score digit: 3
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 2964;
//mem_addr <= 79764;
width <= 8;
height <= 15;
end
```

```
18: //score frame: 4
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 3084;
//mem_addr <= 79884;
width <= 8;
height <= 15;
end
19: //score frame: 5
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 3204;
//mem_addr <= 80004;
width <= 8;
height <= 15;
end
20: //score frame: 6
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 3324;
//mem_addr <= 80124;
width <= 8;
height <= 15;
end
21: //score frame: 7
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 3444;
//mem_addr <= 80244;
width <= 8;
height <= 15;
end
22: //score frame: 8

begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 3564;
//mem_addr <= 80364;
width <= 8;
height <= 15;
end
23: //score frame: 9
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 3684;
//mem_addr <= 80484;
width <= 8;
height <= 15;
end */
2: //"you win" frame
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 82197;
//mem_addr <= 85604;
width <= 100;
height <= 50;
```

```
end
1: //"game over" frame
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 87197;
//mem_addr <= 80604;
width <= 100;
height <= 50;
end
/*16: //"background" frame //not needed, taken care of in overall module
begin
mem_addr[18:17] <= 2'b10;
mem_addr[16:0] <= 0;
width <= 320;
height <= 240;
end*/
default: begin
mem_addr[18:0] <= 19'b10xxxxxxxxxxxxxxxx; //in default, better to do "xxxx..." then
width <= 9'bxxxxxxxxx;                                          //actual numbers, because the timing/logic
cells less.
height <= 8'bxxxxxxxx;
end
endcase

end

always @(size)
case (size)
  3'b000: begin width_size <= 64;
                                        height_size <= 64;
                                        mem_addr_size[18:17] <= 2'b11;
                                        mem_addr_size[16:0] <= 0; end
  3'b001: begin width_size <= 43;
                                        height_size <= 43;
                                        mem_addr_size[18:17] <= 2'b11;
                                        mem_addr_size[16:0] <= 0; end
  3'b010: begin width_size <= 32;
                                        height_size <= 32;
                                        mem_addr_size[18:17] <= 2'b11;
                                        mem_addr_size[16:0] <= 0; end
  3'b011: begin width_size <= 22;
                                        height_size <= 22;
                                        mem_addr_size[18:17] <= 2'b11;
                                        mem_addr_size[16:0] <= 0; end
  3'b100: begin width_size <= 16;
                                        height_size <= 16;
                                        mem_addr_size[18:17] <= 2'b11;
                                        mem_addr_size[16:0] <= 16384; end
  3'b101: begin width_size <= 11;
                                        height_size <= 11;
                                        mem_addr_size[18:17] <= 2'b11;
                                        mem_addr_size[16:0] <= 16384; end
  default: begin width_size <= 9'bxxxxxxxxx;
              height_size <= 8'bxxxxxxxx;
              mem_addr_size <= 19'bxxxxxxxxxxxxxxxxxxx; end
endcase
```

endmodule

6.3e Sync_Gen

```
module sync_gen(clk,reset,hsync,vsync,hblankon,vblankon,hcount,vcount,hreset);
  input clk; //12 mhz
  input reset;
  output hsync, vsync, hblankon, vblankon,hreset;
  //output [5:0] rgb;
  output [8:0] hcount;
  output [9:0] vcount;

  reg hsync,vsync,hblank,vblank;
  reg [8:0] hcount;      // pixel number on current line
  reg [9:0] vcount;              // line number
  //screen is 320 by 480;

  wire en;
  assign en = 1;

  // horizontal: 381 pixels = 31.76us  // 381/31.76us = 12mhz
  // display 320 pixels per line
 wire hsyncon,hsyncoff,hreset,hblankon;

 assign hblankon = en & (hcount == 307);
 assign hsyncon = en & (hcount == 313);
 assign hsyncoff = en & (hcount == 358);
 assign hreset = (en & (hcount == 380));

 // vertical: 528 lines = 16.77us
 // display 480 lines
 wire vsyncon,vsyncoff,vreset,vblankon;

 assign vblankon = (hreset & (vcount == 479)); //|| reset; //you could of added the reset signal in this line
except
                                                        //since good practice to initilize states when reset,
might as well

          //make vblankon high implicitly in reset (see below lines of code)
 assign vsyncon = hreset & (vcount == 492);
 assign vsyncoff = hreset & (vcount == 494);
 assign vreset = hreset & (vcount == 527);

  // sync and blanking
  always @(posedge clk) begin
          if (reset) begin //Lynne gives me a "fake" reset, a reset that she sets, which happens when I give
her frame_done.
                    hcount <= 380; //ALWAYS initialize all your states (registers) when reset.
                    vcount <= 479; //when hcount is 380 and vcount is 479, vblankon becomes high, and
done_frame is high--> see overall
          end
          else begin
             hcount <= en ? (hreset ? 0 : hcount + 1) : hcount;
          hblank <= hreset ? 0 : hblankon ? 1 : hblank;
          hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;   // hsync is active low
```

```
            vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
            vblank <= vreset ? 0 : vblankon ? 1 : vblank;
            vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;   // vsync is active low
             end
            end
//////////////////following used to test random output onto screen/////////////////////////////
//reg [5:0] rgb;

//color bars
//always @ (hcount) begin
//if (vblank | (hblank & ~hreset)) rgb <= 0;
//else
   //rgb <= hcount[7:2]; //by moving it down each color bar gets skinnier and therefore repeats itself more.
i.e. from
                                                    //hcount[8:3] to hcount[7:2] to hcount[6:1]
//end
////////////////////////////////////////////////////////////////////////////////////////////

  endmodule
```