ECE 385

Final Project

Spring 2018

Megaman Survival

Sahil Shah/sahils2
Samir Kumar/samirkk2
ABC - Tues, 11:30-2:20
Vibhakar Vemulapati

Introduction:

The goal of this project is to implement a condensed version of the classic NES game, Megaman. Specifically this project will implement portions of Megaman's cutman level. Megaman was released on the NES in 1987 and is considered a classic arcade game today. Instead of implementing the entire level, this project will slightly modify the game into a "survivor" style game. Megaman will be able to run around while enemies constantly assault him. The goal is to try and survive as long as possible. This game will use the NIOS II/e processor to handle the USB protocol. The rest of the game implementation will be handled in SystemVerilog HDL. With regards to keycodes, the arrow keys will be used to move and climb up/down ladders. The z and x keys will be jump and shoot respectively. Multiple key codes are accepted, so Megaman can jump and shoot, move and shoot, and jump, shoot and move.

Written description of the hardware:

Sprite Display: Sprites and backgrounds in this game are instantiated in on-chip memory. The sprite metadata contained color palette numbers that represent an RGB hex value. These color palette numbers are generated using the python script provided. The sprite data is instantiated using a two dimensional array and can be indexed using the horizontal and vertical coordinates of the image to be printed. Because all of the images were stored in the same module, a sprite select signal was used to determine which sprite to be indexed. This is determined in the game logic including the modules that control Megaman, the bullet, and all enemies. Once a sprite is selected, it is indexed and its color palette number is read. If this number returned the alpha color palette number, then the background is printed instead. For this, the module must also accept DrawX and DrawY (the horizontal and vertical components of the pixel trying to be drawn at the moment). Then the color palette number is outputted from the font_rom module to the color mapper. The color mapper decodes the color palette number and translates the specific RGB values. These values are then outputted directly to the VGA pins.

Animation: Animating a sprite requires the switching of sprites at a specific rate. This is accomplished using a 4-bit counter where the animation switches every 16 cycles of the frame clock. If slower animations are required, a simple fix of increasing the image update rate solves the issue. This entails making the counter count higher (6 bit counter instead of a 5 bit counter).

Collision Detection: Collision detection is required to ensure that Megaman does not travel through boundaries defined in the background. To accomplish this, the horizontal and vertical coordinates of each boundary is manually determined by observing the metadata of the sprite, which is defined in the font_rom file. Once these coordinates have been detected, specific rectangles are defined where movement cannot occur. If Megaman is moving and encounters this defined rectangle, his horizontal motion is zeroed while he is in that location.

Gravity: Gravity is implemented through defined movement in specific rectangles defined based on the background. These rectangles refer to areas where Megaman is in the air and are

defined with the coordinates on the screen, which were determined by observing the background metadata in the font_rom module. If in one of the defined rectangles, Megaman's vertical motion gets set to positive two (meaning that his position falls by two pixels per frame). This sequence also accounts for horizontal key inputs. If left or right is pressed, the horizontal motion of Megaman gets set to negative two (for left) and positive two (for right). When combined with vertical movement, horizontal key presses while 'in the air' result in a diagonal falling motion until Megaman is out of the designated rectangle (i.e. he is on the ground). This dictates where "ground" is set.

Movement: Movement requires the output of the keycode reader, which has been extended to work for as many as 4 key presses. The keycode reader outputs boolean values (1 or 0) for each key where 1 means it has been pressed and 0 means it hasn't. If the right or left key has been pressed, Megaman will move that direction by adding or subtracting to its current position. When both left and right is pressed, Megaman does not move. If within the rectangle that defines the ladders. Megaman can climb up or down the ladder using the up and down arrows. When at the top of the ladder, Megaman snaps to the ground and faces the direction of the last horizontal key press (left or right).

Shooting: Megaman's shooting capabilities come with the press of the X key on the keyboard. In this version of the game, Megaman's capabilities are limited to one bullet on the screen at a time in order to maximize the need for high precision. The implementation of shooting initially checks if the X key has been pressed. Then the next check is to see if the bullet is *not* on the screen at the instance of the keypress. If it is not, the direction that Megaman is facing at the time of the keypress is recorded then latched until the next bullet is ready to be fired. This ensures that the bullet will travel in the direction that it was fired in despite Megaman changing direction. Once the bullet is turned on, the motion is defined deterministically based on the input direction of the bullet. A step size of nine pixels per frame is either added or subtracted from the position each frame cycle. Lastly, the edge and collision case of the bullet is determined. If the bullet (defined by the horizontal and vertical coordinate of the top-left corner of the image) is at the far right or far left of the screen or has collided with an enemy sprite, the bullet turns off and another bullet is able to be fired. A collision boolean is passed from the ball.sv module where is checks the coordinates of the bullet and the enemy sprite to see if the rectangles that each sprite is drawn in overlaps. If there is overlap, a collision is detected (the collision boolean is set high).

Enemies: describe the operation of the enemy module. This is determined by a separate enemy FSM. The hat enemy flies around Megaman, and is not bound by any of the boundaries that Megaman is save for the bounds of the screen. The enemy will moves at 2 pixels per frame horizontally when moving around. This means that he will move slower than Megaman and will constantly follow him. The enemy will constantly be above Megaman. Once the enemy gets on top of megaman, he will dive bomb down towards him. His Y motion will go from 0 to 10 pixels per frame, and if the enemy manages to touch Megaman, Megaman will be damaged. If the enemy is shot at any point, he will disappear. If the enemy successfully dive bombs Megaman,

once it gets down to 40 pixels below Megaman's head, he will then move back up toward his original position and the process repeats. The collision boolean that is passed from the ball.sv module into the bullet module is also passed into the enemy module. If a collision of the bullet and the enemy is detected, the enemy will disappear just as the bullet does. After a collision is detected, a counter will start, which represents a waiting period for the next enemy to appear. After 3 seconds, or 180 frames after a collision is detected, another enemy will appear for Megaman to kill. This will represent the survival aspect of the game, as Megaman can only take so much damage. Once Megaman has been hit enough times, he will be killed and the game will end.

Scoring: Scoring for this game takes place in the same module where the enemy is controlled. When a collision was detected the score is incremented. The current score is displayed on the hex display and is tracked while Megaman still has heath.

Damage: Damage detection is done similarly to collision detection: the rectangles that define Megaman are checked for overlap. If overlap is detected, the damage signal is sent high. When the damage signal is high, a counter is triggered. This acts as a time after taking damage where Megaman is invincible. At the start of the counter, the amount of life megaman has left is decremented. This value is also displayed on the hex display. If Megaman has no health left, the game resets.

NOTE: see annotated background for specific boundary locations

Written description of the software:

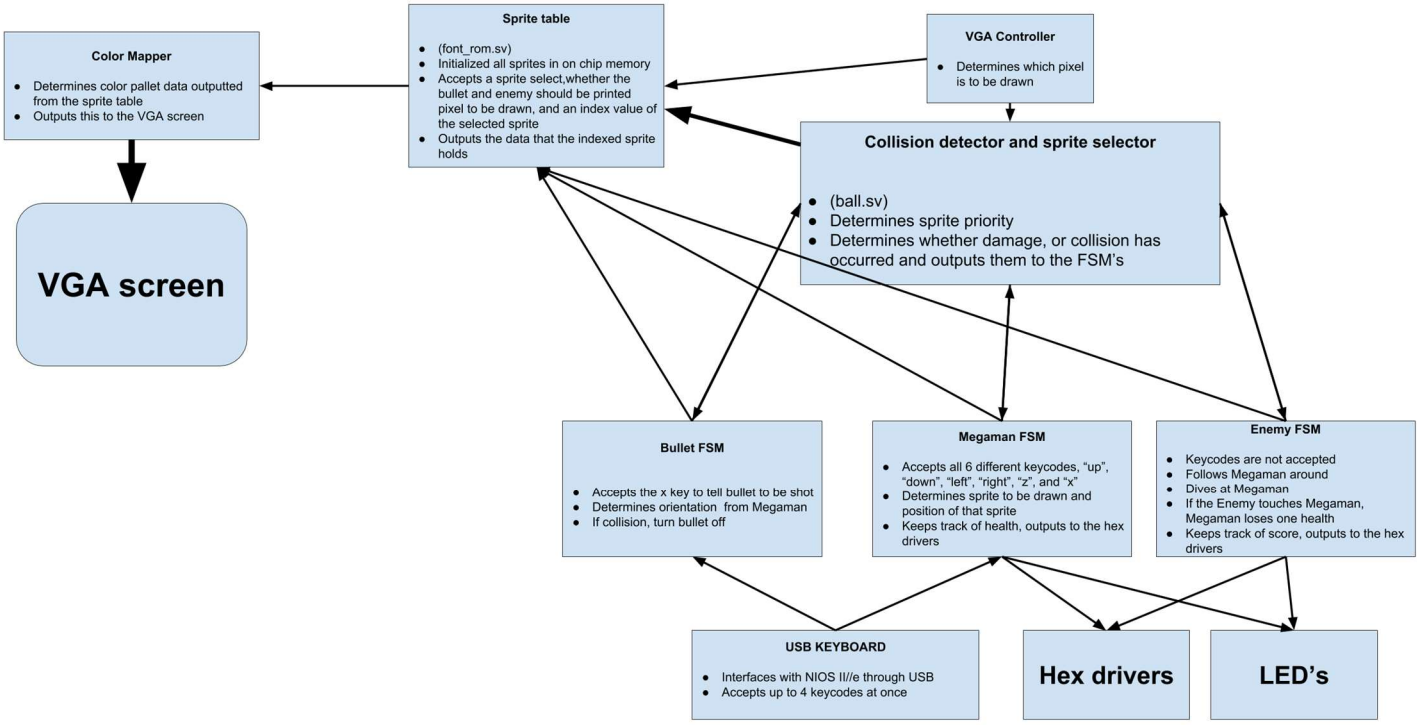
IO_write: Writes data to the HPI data register at the address stored in the HPI address rUess passed to the function. Then, the HPI chip select and HPI write registers are dereferenced and assigned to zero. Now the data HPI data register is assigned the data passed to the function. Lastly, the chip select and write values are set back to one.
wr
IO_read: Reads data from the address specified in the HPI address register. The HPI address register is dereferenced and assigned the target address. Then chip select and read are set to zero and the data in the HPI data register is returned.

USB_write: perform an IO_write to write the address to the address register, then an IO_write to write the data to the data register.

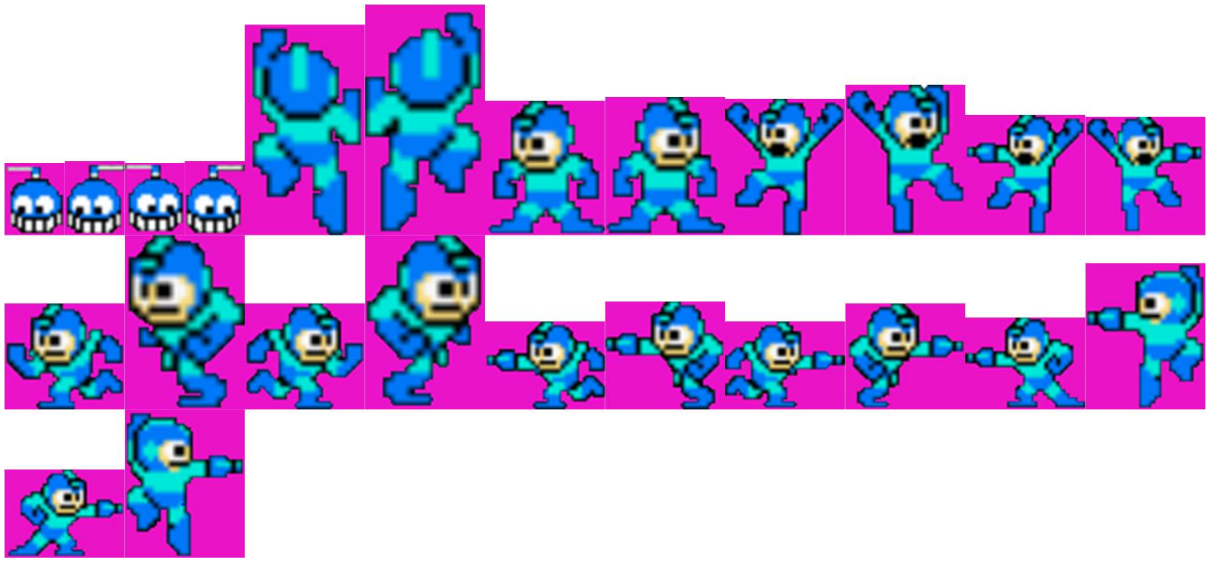
USB_read: perform an IO_write to write the address to the address register, then an IO_read to read the data from the data register.

2Darray: This is a script designed to reformat the 1D array received by the png_to_palette_relative_resizer program to a 2D array formatted for SystemVerilog HDL.

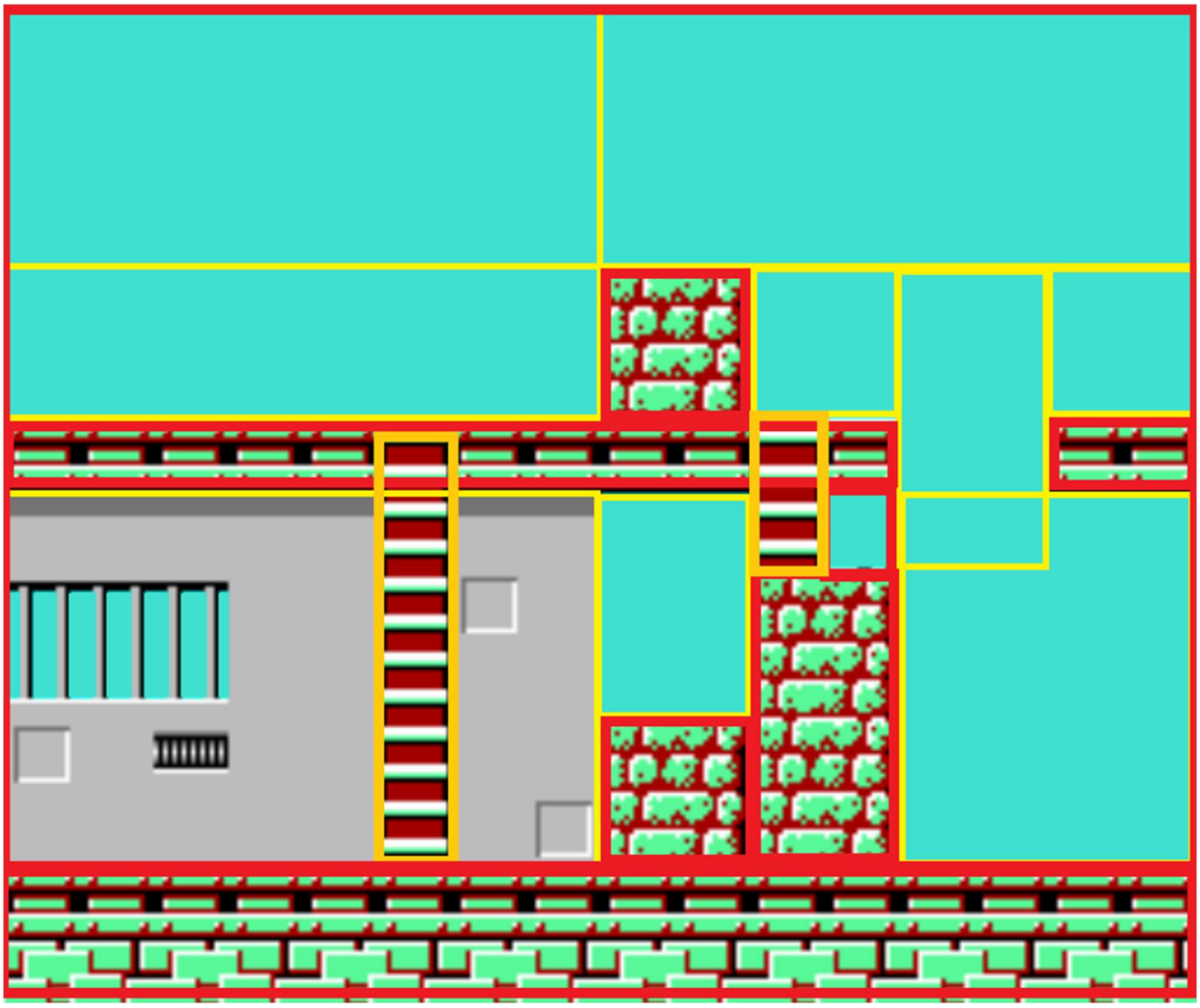
Block Diagram:



Sprites:



Annotated Block Diagram:



Inside of the orange rectangles, Megaman is able to move up and down. The up and down arrow keys being pressed in this region would mean that he is on a ladder. These places have the highest priority. If Megaman is on the ladder, the ladder will override any falling that would otherwise occur. Megaman will be able to jump through the ladder as though it wasn't there if he is not currently on the ladder. When on the ladder, the right and left arrow keys will not do anything other than change which way he would shoot when the x key is pressed. Megaman will snap to the middle of the ladder when on it, and will snap off when he is at the top and bottom of the ladder.

The next highest priority would be falling. This is characterized by the yellow rectangles. If Megaman is not on a ladder and within those yellow rectangles, Megaman will fall. This is met by making his Y motion to be increasing by 3 pixels per frame. He will fall until he hits the end of a rectangle.

The lowest priority is the red rectangles. This means that Megaman will no longer be able to move in that direction. So these determine the screen size bounds, floors, as well as the

block bounds, so megaman can't walk or jump through objects. If he tries to move past a boundary, his motion in that direction is zeroed. Right of the big block on the lower level in the original NES version of the game led to another screen but in our game it is essentially a trap. To overcome this, we allow the user to jump up the side of the block, essentially adding wall-climbing ability. Also the smaller ladder in the real version of the game is essentially useless. Megaman cannot climb or jump off this ladder. Therefore, we decided to eliminate its functionality from this game.

Module Description:

Module: Color_mapper.sv
Inputs: [3:0] sprite_data
[9:0] DrawX, DrawY
Outputs: [7:0] VGA_R, VGA_G, VGA_B
Description: This module takes color palette data from the sprite data and translates it to an RGB value to output to the computer
Purpose: This module acts as a buffer from the actual spite metadata and the colors needed to output to the VGA display.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: Creates the interface to the hex displays
Purpose: Allows the program to output data to the hex displays

Module: hpi_io_intf.sv
Inputs: Clk, Reset, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset
[1:0] from_sw_address
[15:0] from_sw_data_out, OTG_DATA
Outputs: OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N
Description: Assign the value of the "On the Go" signals
Purpose: Take the values of the "On the Go" signals and pass them to the on board USB chip. The output is set to high impedance unless the bus is being written to.

Module: VGA_controller.sv
Inputs: Clk, Reset, VGA_CLK
Output: VGA_HS, VGA_VS, VGA_BLANK, VGA_SYNC_N
[9:0] DrawX, DrawY
Description: Determines the coordinates to draw
Purpose: Controls how the screen is drawn by outputting the horizontal and vertical coordinates

Module: font_rom.sv
Inputs: Clk, print_background

[4:0] sprite_select
[9:0] DrawX, DrawY
[10:0] addr_x, addr_y

Output: [3:0] data

Description: This module contains all of the sprite metadata and will output a color palette number using the address inputted and the sprite selected. Each sprite is stored using a 2-D array, and is accessed as such. There is a case statement that determines which sprite is to be drawn, and if that sprite should be defaulted to background, or the actual sprite.

Purpose: This module is used to house all background and sprite data. The address inputted refers to an index of the sprite that needs to be selected. Once indexed, this module outputs the color palette number to the Color Mapper for translation to RGB values.

Module: keycode_reader.sv

Inputs: [31:0] keycode

Outputs: up_on, left_on, right_on, z_jump_on, x_shoot_on

Description: This module translates multiple keycodes and outputs boolean values for each button that is pressed for a max of 4 buttons. This module was implemented as described on KTTech.

Purpose: This module provides the user with the capability of multiple key presses. This can be vital in this game as multiple actions at the same time are often necessary.

Module: FSM.sv

Inputs: up_arrow, left_arrow, down_arrow, right_arrow, z_jump_key, x_shoot_key, frame_clk Clk, RESET

Outputs: LED0, LED1, LED2, LED3, LED4, last_horizontal
[4:0] sprite_select
[9:0] Megaman_x_position, Megaman_y_position

Description: Megaman's motion is defined by multiple actions all with a certain priority. The highest priority is climbing a ladder. This, like all other motion is defined by a specific rectangle. That is, Megaman can only climb up and down the ladder if he is within the defined rectangle. If within this rectangle and on the ground, Megaman's horizontal position will be snapped to the center of the rectangle (i.e the center of the ladder). Once Megaman is snapped to the center of the ladder, his animation is switched to the climbing animation and up and down movement is enabled. Once at the top of the ladder (the top of the rectangle), Megaman snaps off the ladder and faces the last horizontal direction that was entered by the player. The same happens when Megaman reaches the bottom of the ladder. Also while on the ladder, if the X key is pressed, Megaman will enter the shooting animation. The X key press boolean is also passed to the bullet controller that generates and moves the bullet. If Megaman is not on the ladder, this state machine checks whether or not Megaman is in the air. This is also done using defined rectangles in the background. If Megaman is in one of these rectangles, his incoming vertical motion to 3, meaning that his vertical position is increased by 3 pixels per frame. While in the falling state, this module checks whether the horizontal keys have been pressed. If so, the horizontal motion will be set accordingly. Similarly, if the shoot key is pressed, the falling while shooting animation is triggered. Next on the priority ladder is the jumping state. Implementation

for jumping is similar to falling in that the Y motion is changed every cycle and the X motion is changed based on the user's horizontal key press. The Y motion is set on a counter to simulate negative acceleration. For fourteen cycles, Megaman moves at 4 pixels per frame, for the next eight cycles he moves at 2 pixels per frame, and for the last nine frames, he moves at 1 pixel per frame. This is accomplished using a 5-bit counter. If Megaman is not on climbing, falling, or jumping, he is standing or running. In this state, Megaman is on the ground and the up and down keys are ignored. His motion only depends on horizontal key presses and, as always, the shoot key triggers the shooting while running or standing animation (depending whether or not he is running or not).
Purpose: This module controls the movement of the main character, Megaman. It accepts the outputs of the keycode reader and controls Megaman's movement based on those signals and his location with respect to his background.

Module: Bullet_FSM.sv
Inputs: x_shoot_key, last_horizontal, frame_clk, Clk, RESET, collision
[9:0] Megaman_x_position, Megaman_y_position
Outputs: LED0, bullet
[9:0] x_position, y_position
Description: This module is the bullet controller for this game. On the press of the shoot key, the bullet status is checked. If the bullet is not on, the direction that Megaman is facing at the time of the press is latched and the bullet is turned on. Once the bullet is turned on, it travels in the direction that was latched at the time of the bullet being fired. The last check that is made is a boundary and a collision check. If the bullet has shot an enemy or is at the boundary, it turns off (i.e. it is taken off the screen).
Purpose: The purpose of this module is to control the motion of the bullet on the screen once the shoot key has been pressed

Module: Hat_Enemy_FSM.sv
Inputs: Clk, RESET, frame_clk, is_collision
[9:0] Megaman_x_position, Megaman_y_position
Outputs: hat_on, LEDG7
[9:0] hat_enemy_position_x, hat_enemy_position_y
Description: This module describes the motion of the Blader. He will move towards Megaman and dive bomb him once he gets close. This is done through the same rectangle system used in Megaman's FSM. We create a rectangle around Megaman that Blader will go to. Once inside that rectangle, Blader will go into the dive bomb sequence, where he will move down rapidly, until it moves 40 pixels below Megaman's head. Once it has finished that, it will move back to its original y coordinate position, 100 pixels above megaman. This will make it hard for Megaman to shoot the enemy. If Megaman and Blader have overlapping pixels, a collision is detected and Megaman will take damage, reducing his HP. It takes one bullet to collide with Blader to kill it. Once a collision is detected, the score is increased.
Purpose: This module is used to control one of Megaman's enemies, specifically, "Blader".

Module: ball.sv (this is an extended version of the lab 8 ball.sv version)

Input: Clk, RESET, bullet_on, hat_on
[9:0] DrawX, DrawY, Megaman_x_position, Megaman_y_position, bullet_x_position, bullet_y_position, hat_enemy_position_x, hat_enemy_position_y,
Output: print_background, print_bullet, print_hat_enemy, is_collision, damage, LEDG6
[10:0] addrx, addry
Description: This module handles the address finding and collision detection among sprites. First it checks if the rectangle that defines the bullet overlaps with the rectangle that defines the enemy. If so, a collision has been detected and the collision signal gets set to 1. Next, the same overlap is checked between Megaman and the enemy. If they overlap, Megaman takes damage and the damage signal gets set to 1. Then this module checks if the pixel being drawn (defined by DrawX and DrawY) is within the Megaman's rectangle. If so, addrx and addry are set to index one of Megaman's images (the specific image is determined by the sprite select signal). If not, the same is checked for the bullet and then the enemy. Out of these three, the enemy has the highest priority, followed by the bullet and Megaman after that.
Purpose: The purpose of this module is provide the address of pixel of the sprite (stored in font_rom) to be displayed.

Module: lab8.sv
Inputs: CLOCK_50, OTG_INT
[3:0] KEY
[15:0] OTG_DATA
[31:0] DRAM_DQ
Outputs: OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, DRAM_RAS_N, DRAM_CAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS
[1:0] OTG_ADDR, DRAM_BA
[3:0] DRAM_DQM
[6:0] HEX0, HEX1
[7:0] VGA_R, VGA_G, VGA_B, LEDG
[12:0] DRAM_AtDDR
[15:0] OTG_DATA
[17:0] LEDR
[31:0] DRAM_DQ
Description: This module is the top level design. It instantiates all modules.
Purpose: Connects all modules together in hardware such that the final product is a combination of Megaman, enemies, and background displayed on a VGA monitor.

Design Resources and Statistics:

LUT	35,333
DSP	0

Memory (BRAM)	55,296
Flip Flops	2406
Frequency	42.73 MHz
Static Power	109.21 mW
Dynamic Power	237.84 mW
Total Power	430.22 mW

Conclusion:

This project came with a variety a large milestones that needed to be passed along the way. The first was the ability to print and move a sprite (without animation or a background) on the screen. This did not come easily as the notion of printing sprites on the screen was unfamiliar to us. Initially we attempted to instantiate ROM using the rom.sv file given to us. We realized fairly quickly that the best way to instantiate memory on-chip was to use a two-dimensional array that could be addressed using the horizontal and vertical coordinates of the pixel. Initially, the alpha in the sprites would not be ignored. The screen would display the alpha color instead of the background. The simple fix to this was to check the color pallete number before outputting the data. If it was zero (or the number that corresponded to the apha), the background was addressed using DrawX and DrawY and that data was outputted instead.

Another issue with this report was the boundary conditions. The only way for us to determine whether Megaman collided or interacted with something in the background was to hardcode rectangles around the objects. Rectangles refer to a range of horizontal coordinates and vertical coordinates. This proved to be tedious but effective. Another proposed method was to draw a unique color around all the boundaries on the original image and regenerate the color palette data array. This proved to be difficult, however, because the 'png_palette_relative_resizer' tool given to us used a 'find nearest neighbor' search in order to compress or enlarge the original image. This distorted the resulting image and gave unwanted results. This could have been fixed by adjusting the size of the unique color boundary on the original image but time was of the essence so we continued with the rectangle method instead.

On-chip memory was used to store sprite and other image data. This method allows for smoother and faster access but exponentially increases the size of the generated hardware (and therefore compile time). Another option is to store sprites and images in SRAM. This not only decreases hardware and compile time, but also allows for more storage. The downside to SRAM, however, is that it can be harder to use. We committed to on-chip memory earlier in the project and decided to stick with it throughout. Because using on-chip memory increased the hardware, the graphics sometimes became extremely slow and laggy. This was fixed by finding inefficient hardware and streamlining it.

Enemy AI was difficult to implement. The act of following Megaman was fairly easy, but for some reason, bounding the divebomb action and subsequent reset of the enemy proved to be difficult. He would constantly either not divebomb or just run away from megaman. After a lot of trial and error, we got the enemy to follow and divebomb properly.

We feel that because the compilation time was so long, that debugging was made exponentially more difficult, because by the time we made the changes and waited for the program to compile, we forgot what changes we made. So small, incremental changes were necessary to solve problems, which were very time consuming and made even small problems seem more difficult than they really were.