ECE 385

Spring 2023

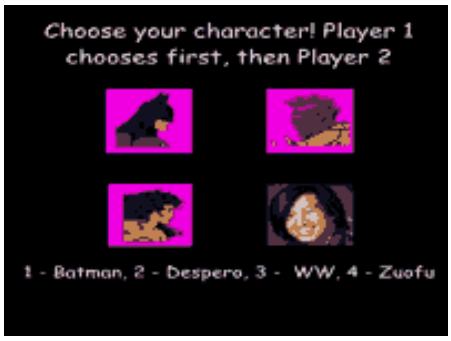
Final Project Report: Justice League Task Force using SoC with VGA and USB Interface

Jaejin Lee/Raj Baala

5/9/2022

Introduction and Explanation of the Functionalities

For our final project we decided to make our own version of Justice League Task Force, a fighting style game that was released on the NES and Genesis gaming systems in 1995.
The game starts with a start screen that says “Press Enter to Start” whereupon pressing Enter goes to the character selection screen.



Here, player1 chooses their character first and player2 chooses their character seconds by using “1234” to pick the representative character on the screen.
The game then starts with a “3 2 1 fight” countdown where players are initialized but cannot move until after the “fight” symbol disappears from the screen.


Players then can move around freely and can makes moves such as jump, crouch, kick and punch by pressing on the assigned keys on the keyboard.


Once the special attack meter is full, players can shoot a red ball which can decrease the other players health significantly if they don't dodge or block.

Both players then fight each other until either player's health bar drops to 0 in which the KO symbol appears and both cannot move anymore.

Once the game is over and can be restarted using the KEY1 button on the FPGA that represents Reset. The original game contains up to 8 different characters, but our implementation only includes 3 characters due to hardware limitations of the DE-10 FPGA board. The original game also includes multiple maps, but ours only contained one. Given more time, and hardware capacity, implementing more sprites and backgrounds would've been possible for a better gaming experience. The graphics from the original and our implementation is different as ours had to utilize lower resolutions and only allow a max resolution of up to 640x480 due to VGA hardware capacity. One unique feature that we implemented is a special attack meter which was not existent in the original game. Once the meter fills up fully, the user is able to make a special attack that will one shot the other player upon impact. Overall, this final project can be thought of as a "street fighter" type game that contains characters from the DC universe. In this report, we will be going over the internals of the design for the game.

Written Description of NIOS-II System

How the NIOS Interacts with both the MAX3421E USB chip and the VGA Components

The DE10-Lite shield includes a USB type-A port and MAX3421E which allows for physical connection with the keyboard to the FPGA board. The way NIOS II interacts with the MAX3421E USB chip is through the code we wrote in four read/write functions in MAX3421E.c. In each function, we called Intel's provided avalon driver, which allows the NIOS II to communicate to the MAX3421E via the SPI protocol. We also instantiated an SPI peripheral block within platform designer in order to connect the hardware to the software that utilizes SPI. Through those functions, NIOS II is able to read and write to MAX3421E registers.

The inclusion of the keycode PIO within platform designer allows for signaling the playermovement and playermovement2 routine in moving the sprites. Based on the value of the

exported keycode signal(QWERASDF/YUIOHJKL), sprite1 and sprite2 will do the corresponding action linked to the keypress. The keycode signal is utilized within the instantiated playermovement/playermovement2 module that controls the position of the sprites based on keycode. Since position logic is updated within player movement based on keycode, color_mapper will also update its pixel colors based on the sprite locations while the vga_controller will allow the sprites to be synced to the screen based on vs and hs signals.

Written Description of the SPI Protocol

The SPI protocol stands for serial peripheral interface protocol which is a protocol in which data is transferred serially between a master and slave device. In this case, the master is the NIOS II and the slave is the MAX3421E USB host chipset. We utilize this protocol in order to interface the MAX3421E with the NIOS2 and SPI peripheral that is created in platform designer. The SPI port consists of four pins that allow for data transfer and receival between the master and slave devices. The four pins are master-in-slave-out(MISO), master-out-slave-in(MOSI), slave select(SS), and serial clock (S_CLK). MOSI transfers data from the master to slave one bit at a time at every rising edge of the S_CLK and MISO transfers data from the slave to master at a time at every rising edge of the S_CLK. When a data transaction is to be made, the SS signal goes low which is sent from the master to slave. In our implementation, the first byte transferred via MOSI is the command byte where the top 5 bits indicate register number for the MAX3421E which has 32 registers, bit 2 is 0 padding, bit 1 is DIR which indicates a R/W and bit 0 which is also 0. The purpose of this command byte is to communicate which register to choose and what operation to do (R/W) based on DIR bit. If DIR is 1, then a write is performed and the second byte of MOSI will contain the data to be written to the register from the previous 8 clock cycles of S_CLK. If DIR is 0, then a read is performed where MISO will transfer the data at the register addressed from the command byte to the master. It is also important to note that the device driver that we utilize in the C code to do the SPI communication for us operates in half duplex mode which indicates that if the read_length = 1 and write_length = 1 for example, then the total number of bytes in transaction will be 2 where write always happens first. A full duplex mode would indicate bidirectional data transfer where both transfers of MOSI and MISO would happen simultaneously.

Details of the VGA operation, and interactions between playermovement modules, VGA controller, and Color Mapper

VGA stands for Video Graphics Array which was the analog video display standard back in the 80s. VGA organizes the display to be a grid of pixels where the pixel dimensions of the visible display is 640 columns by 480 rows or 640 horizontal pixels by 480 vertical lines. Analog voltage signals are sent through the port in order to alter the pixel RGB values that are shown on the display. These analog voltages are also responsible for controlling the intensity of the electron beam which is what is responsible for drawing the image on the screen. The current coordinate positions of the electron beam in the lab are denoted by DrawX and DrawY. There are deflection yokes that can change the positioning of the gun so that the image is displayed in row major order or raster order. The beam will start at the first pixel from the left after finishing each row using horizontal sync or the hs signal in the lab. After the beam is finished drawing the image, the vertical sync signal or vs signal in the lab will reset the electron beam to the top left

corner of the screen in order for the beam to start drawing a new image. Outside the 640 x 480 screen there is a blanking region called the blanking interval which makes the screen size increase to 800 x 525 despite only 640 x 480 pixels being visible. This interval is used for the electron beam to have enough time to reset after a hs/vs signal is given. Finally, the refresh rate of the screen is 60 hz so the vertical sync signal will need to be generated 60 times a second.

The playermovement modules, color mapper, and VGA controller modules are instantiated in the top level file and all have signals that depend on each other in order to work. For example, the DrawX and DrawY signals are outputs from the vga controller module that are inputs to the color mapper module. This allows the color mapper to assign RGB values to pixels based on DrawX and DrawY positions that determine when to draw the sprites or the background. The playermovement modules takes in the keycode signal that is exported from the keycode PIO block. This signal is used to update the position and actions of the sprites based on corresponding keypresses based on the player. The vga controller's output of the pixel_clk is used as the input clock to the color mapper which allows pixels to be drawn at the correct time. The VGA_VS or vertical sync signal is used for most modules that rely on timing or a counter due to VGA_VS being a slower clock compared to the others where its only 60hz vs 25Mhz or 50Mhz. This VGA_VS is used as the frame_clk to allow switching between sprites or creating animations.

Sprite Generation

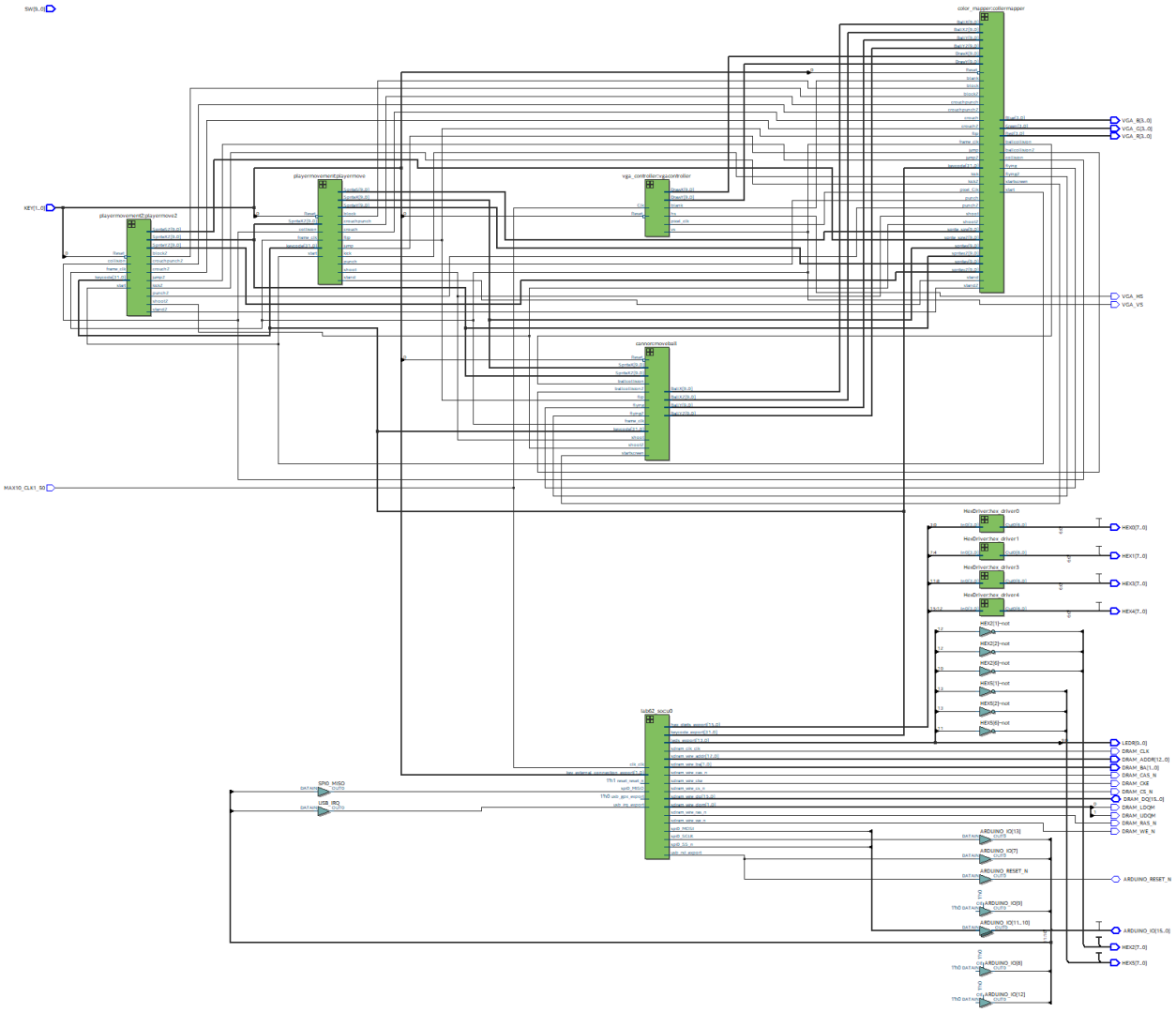
For generating our sprites on the screen, we used a helper tool made by CA Ian Dailis that converts jpg images into SystemVerilog modules that store a palettized image into inferred M9k on-chip memory. The tool is a python script that utilizes the k-means clustering algorithm to convert the images. After running the script, the script asks for a .jpg image as input then asks for the number of bits for the image as well as the x and y resolutions. To first get our sprite images, we used "spriters-resource.com" which provided sprite sheets to download for each character. We then cropped each sprite we needed and applied a hot pink background to the image in order to easily make the background of the sprite transparent in the game through color mapper. After giving all the inputs, the script generates a folder containing a rom.sv, palette.sv, image_example.sv, and .mif file in order for the rom to hold the pre-initialized image data. It also tells you how many bits the image is going to use up on the M9k and thus can plan accordingly how many sprites it is possible to have as well as compromising between resolution and color accuracy. The image_example.sv module instantiates the image's rom and palette in it and sets the rgb values accordingly based on the output from the palette module. Instead of having an image_example.sv for each sprite, we simply had a player1_example.sv and player2_example.sv in which we instantiated all sprite's rom and palettes. This allowed for adding new sprites and character selection implementation to be easier as well as making the code look cleaner. The number of bits for the image as input specifies how many colors the image can use. The higher the number of bits, the more colors the generated palette module for the image will contain. All our sprites utilized 4 bits of color thus containing 16 different colors in the palette to represent the image. As for resolution we chose 64x64 as the resolution for our sprites. This color plus resolution combination for the sprites allowed us a good compromise between storage space on the on-chip memory and visual quality. As for the background, start

screen, and character selection screen, we opted for 3 bits of color and a 200x150 resolution as resolution was more important for this as the image needed to cover the entire screen and would look very blurry if stretching a lower resolution image across the whole screen.

```
Input image (eg: cat.jpg): background.jpg
Number of bits per pixel to store: 3
Desired output horizontal resolution: 320
Desired output vertical resolution: 240
Using 230400 / 1490944 available M9k memory bits
Design may still not fit. M9k block usage is weird.
Resizing image... Done
Palettizing image... Done
Generating MIF (Memory Instantiation File)... Done
Generating ROM module... Done
Generating palette module... Done
Generating example module... Done
Generating .qip file... Done
Generating output image... Done
Output files are in ./background/
```

Image of Ian's tool in terminal

Top-Level Block Diagram

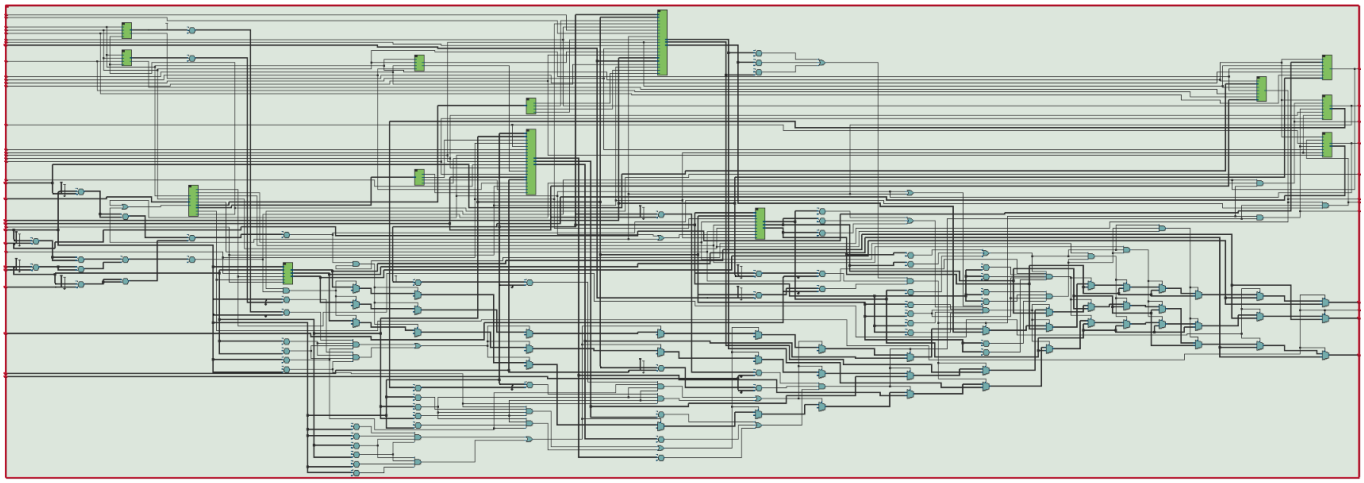


Top-Level Block Diagram of the Final Project

Written Description of All .sv Modules

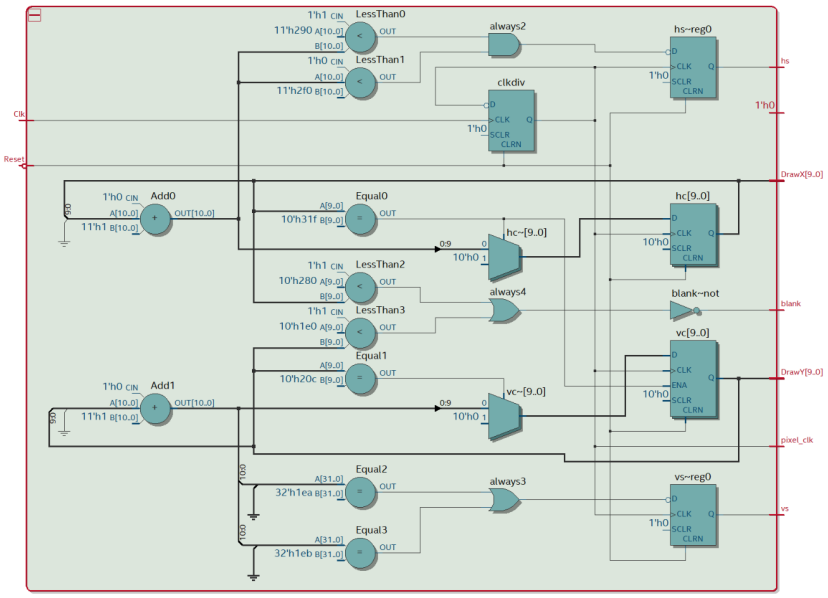
1. Module: jltf.sv
    a. **Inputs:** MAX10_CLK1_50, [1:0] KEY, [9:0] SW,
    b. **Outputs:** DRAM_CLK, DRAM_CKE, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [9:0] LEDR, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
    c. **Description:** top level module
    d. **Purpose**: Top level module for the project that instantiates the vga controller, color mapper, character 1 and 2 movement modules, and special attack cannon module.
    e. **RTL Diagram**: The Top-Level Diagram is provided above.

2. Module: color_mapper
    a. **Inputs:** pixel_Clk, frame_clk, blank, Reset, crouch, crouchpunch, stand, jump, kick, punch, crouchpunch2, crouch2, stand2, jump2, kick2, punch2, flip, shoot, shoot2,block,block2,
    input [9:0] spritex, spritey, spritex2, spritey2, DrawX, DrawY, sprite_size, sprite_size2, BallX, BallY, BallX2, BallY2,
    input [31:0] keycode
    b. **Outputs:** collision, ballcollision, ballcollision2,startscreen,flying,flying2,start, gameover, gameover2, input [3:0] Red, Green, Blue
    c. **Description:** Sets background, foreground color, and sprite colors using combinational logic based on two players positions.
    d. **Purpose**: Sets all necessary RGB values in order for the VGA to display the correct colors / visuals for the players and background and is instantiated in the top level
    e. **RTL Diagram**:

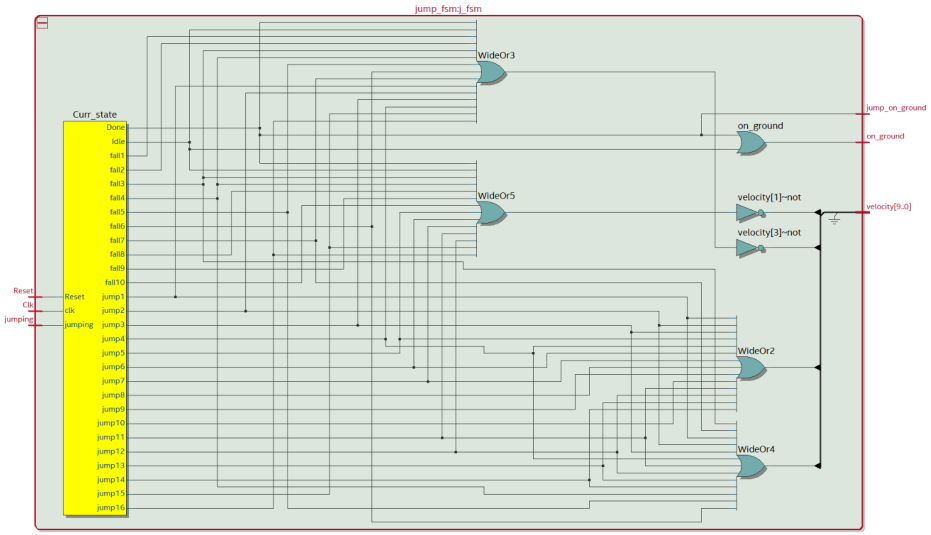


3. Module: vga_controller
    - Inputs: Clk, Reset

- Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, DrawY
- Description: Utilizes sequential logic to set vertical and horizontal sync signals
- Purpose: Sets multiple signals (vs, hs, pixel_clk, DrawX, DrawY) that are used as inputs to other instantiated modules in the top level (color_mapper, ball)
- RTL Diagram:

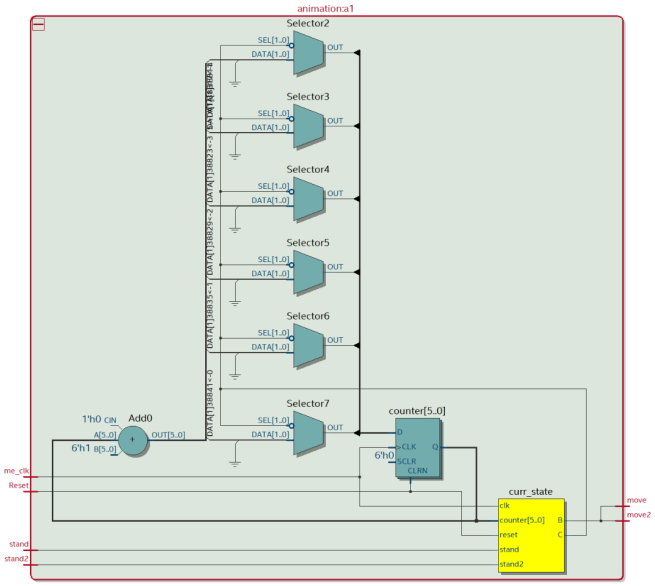


4. Module: jump_fsm
    a. **Inputs:** Clk, Reset, jumping
    b. **Outputs:** jump_on_ground, on_ground, [9:0] velocity
    c. **Description:** state machine for setting velocities
    d. **Purpose**: Main module to implement the jumping physics for character
    e. **RTL Diagram**:

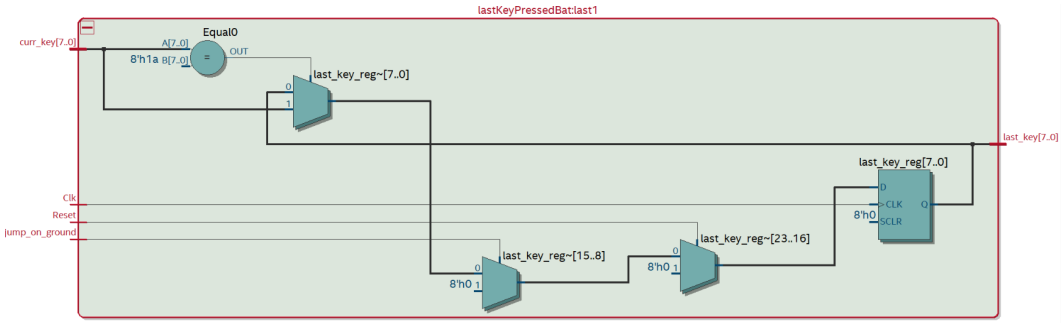


5. Module: animation
    a. **Inputs:** frame_clk, Reset, stand, stand2,
    b. Outputs: move, move2

c. **Description:** This animation state machine module has a counter that counts based on the frame clock. It has 3 states in total with one initial state and 2 other states that assigns whether the output “move” is high or low.
d. **Purpose**: This state machines outputs move and move2 for player 1 and player 2 which is used to assign different sprites for the character. That makes the characters have a walking motion instead of being static.
e. **RTL Diagram**:



6. Module: lastKeyPressedBat, lastKeyPressedWW
   a. **Inputs:** Clk, Reset, jump_on_ground, [7:0] curr_key
   b. **Outputs:** [7:0] last_key
   c. **Description:** A parallel loaded register based on various inputs
   d. **Purpose**: Needed in order to retain jump key press for player1 and player2 respectively until their jump state machine finishes
   e. **RTL Diagram**:

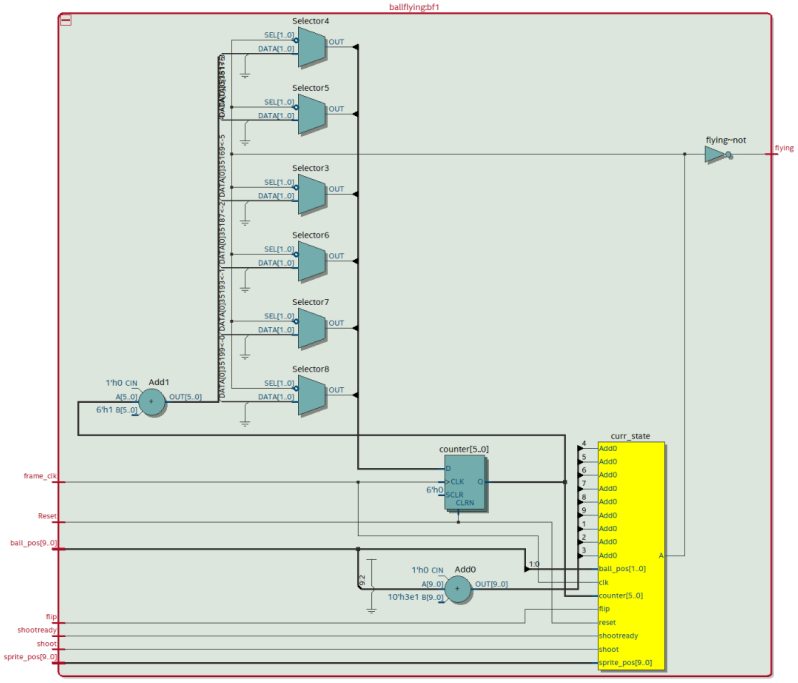


7. Module: ballflying, ballflying2
   a. **Inputs:** frame_clk, Reset, shootready, shoot, flip, input[9:0] sprite_pos, ball_pos
   b. **Outputs:** flying
   c. **Description:** This state machine modules contains 3 state that are based on the special attack key and the special attack meter. Each state assign the output

"flying" with correct value (1 or 0). The state machine also contains a counter so that the ball doesn't fly over the range.

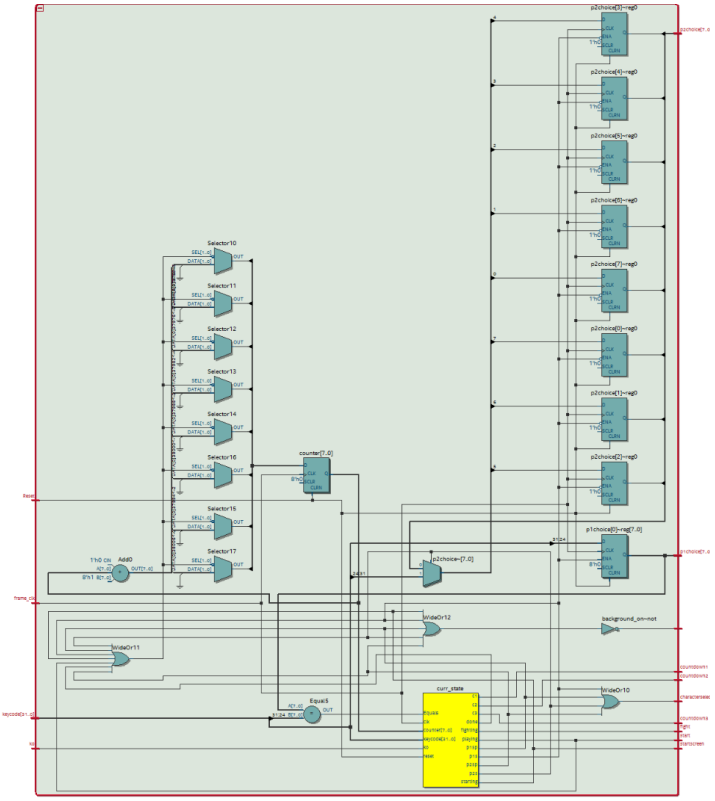
d. **Purpose**: It controls the when the ball is printed on the screen and makes sure the ball only shows up when the speical attack key is pressed.

e. **RTL Diagram**:



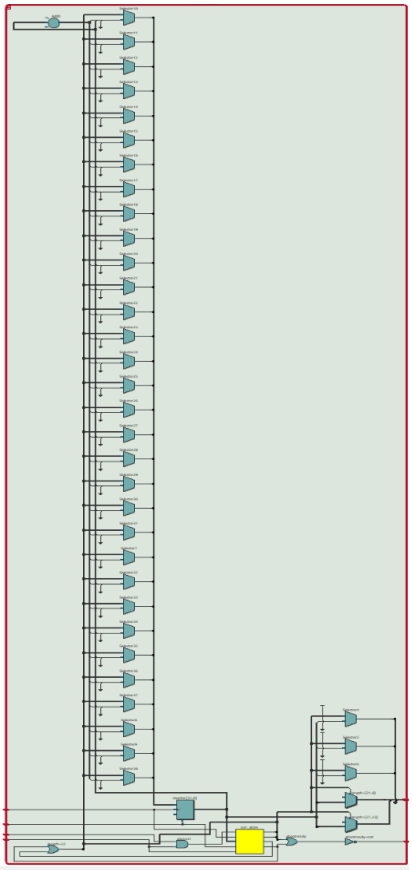
8. Module: initialscreen
    a. **Inputs:** frame_clk, Reset, ko, [31:0] keycode
    b. **Outputs:** background_on, startscreen, characterselect, countdown3, countdown2, countdown1, fight, start, [7:0] p1choice, p2choice
    c. **Description:** state machine that outputs signals in each state given various conditions
    d. **Purpose**: Needed to hold the state of the game and when to transition to next states of the game.

e. **RTL Diagram**:



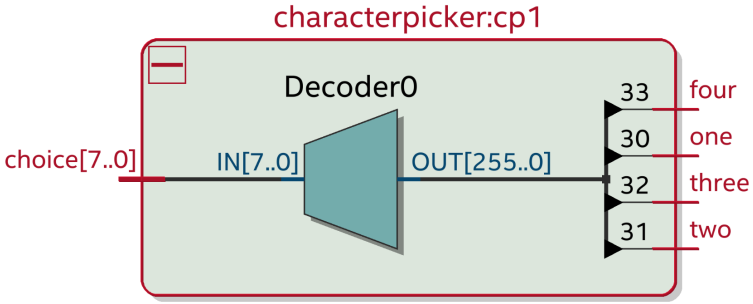
9. Module: specialattack
    a. **Inputs:** frame_clk, Reset, shoot, start
    b. **Outputs:** shootready, length
    c. **Description:** This module uses counter and frame clock to fill up the length of the special attack bar. There are four states that controls the output of the length and the “shootready” logic.
    d. **Purpose**: This state machine module controls the length of the special attack bar

e. **RTL Diagram**:



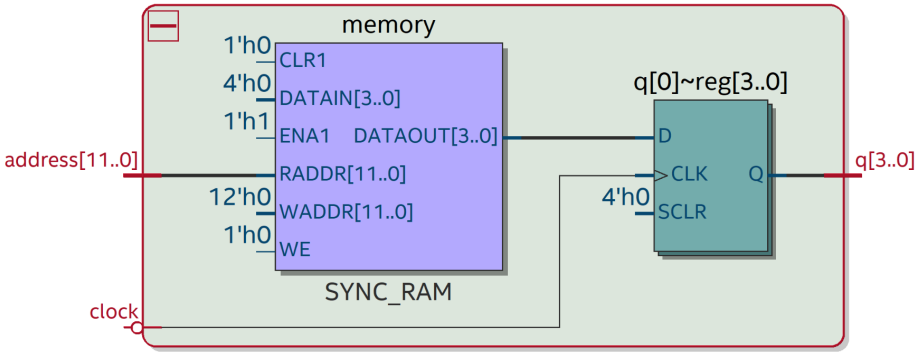
10. Module: characterpicker
    a. **vgaInputs:** [7:0] choice
    b. **Outputs:** one, two, three, four
    c. **Description:** 4-1 MUX
    d. **Purpose**: Module to output character number selected based on what player1 and player2 select by keycode

e. **RTL Diagram**:

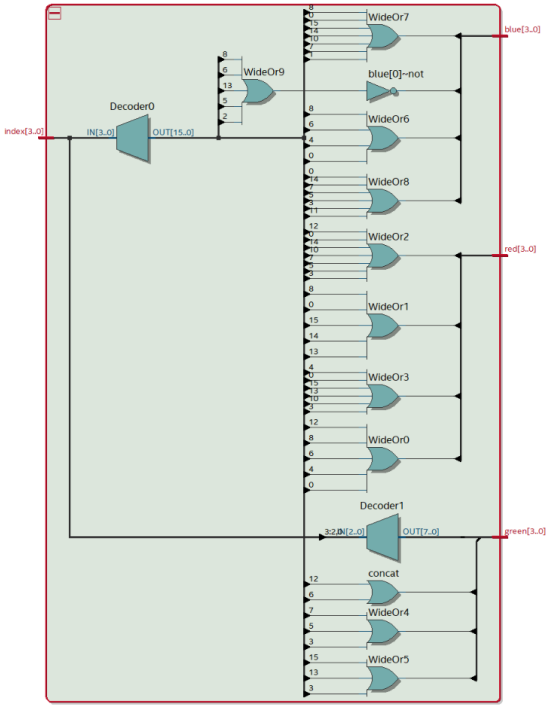


11. Module: zforom.sv, wwstand_rom.sv, batmanstand_rom.sv, desp_rom.sv
    a. **Inputs:** clock, [11:0] address
    b. **Outputs:** [3:0] q
    c. **Description:** contains on-chip memory ROMs
    d. **Purpose**: contains all sprite roms

e. **RTL Diagram**:

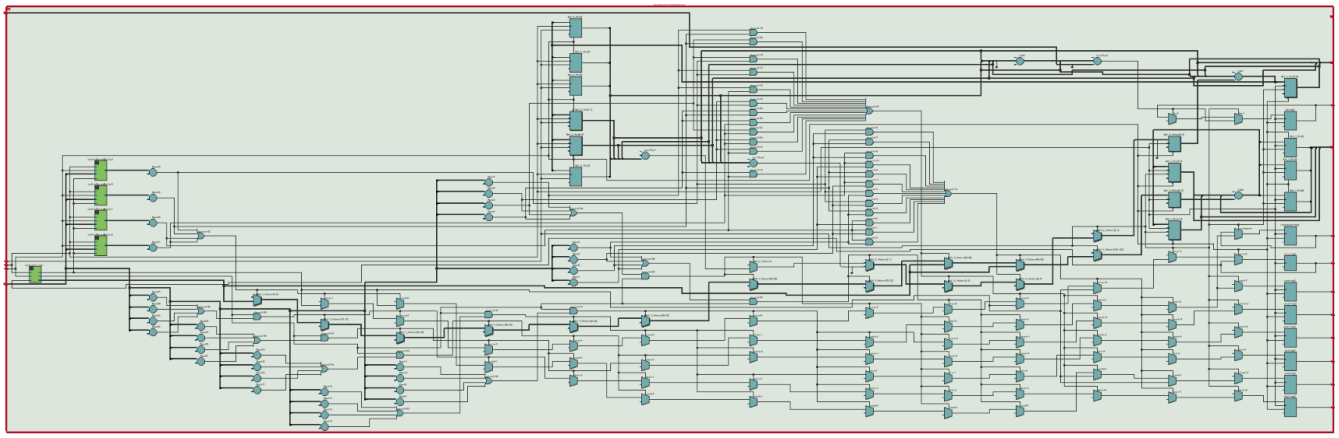


12. Module: zfopalette.sv, desp_palette.sv, wwstand_palette.sv, batmanstand_palette.sv
    a. **Inputs:** [3:0] index
    b. **Outputs:** [3:0] red, green, blue
    c. **Description:** Contains an array of rgb colors to index into
    d. **Purpose**: Needed to index into palette from ROM output index to access sprite colors
    e. **RTL Diagram**:

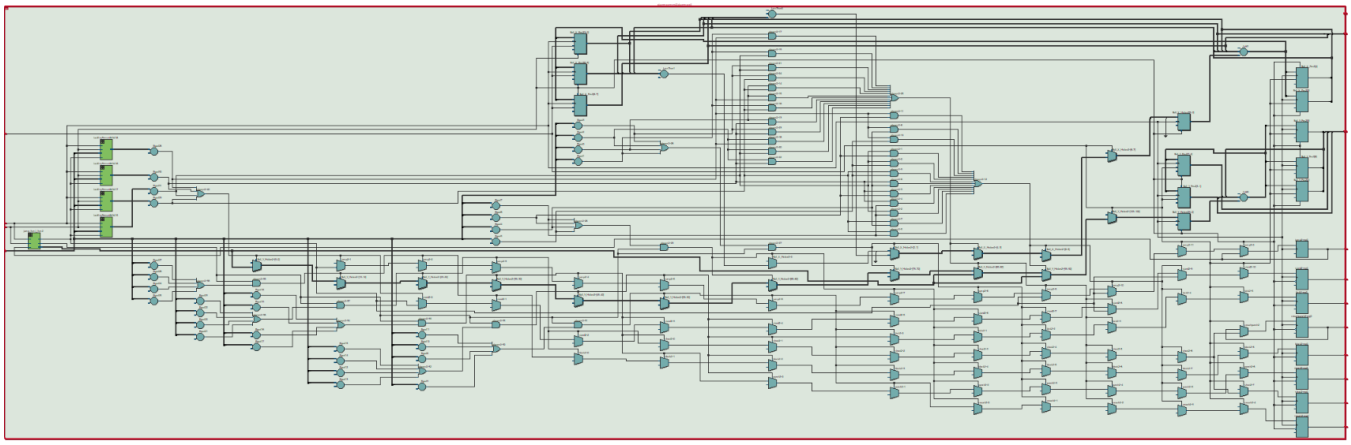


13. Module: playermovement
    a. **Inputs:** Reset, frame_clk, collision, start, [9:0] SpriteX2, [31:0] keycode
    b. **Outputs:** [9:0] SpriteX, SpriteY, SpriteS, stand, crouch, kick, punch, flip, shoot, block, crouchpunch
    c. **Description:** Contains always_ff logic to set output signals based on keycode inputs

d. **Purpose**: ball.sv module from lab 6.2 but modified to incorporate multiple keycodes, stop an action on release of keypress, and output action signals based on certain keypresses / set x and y motion of player1's sprite.
e. **RTL Diagram**:

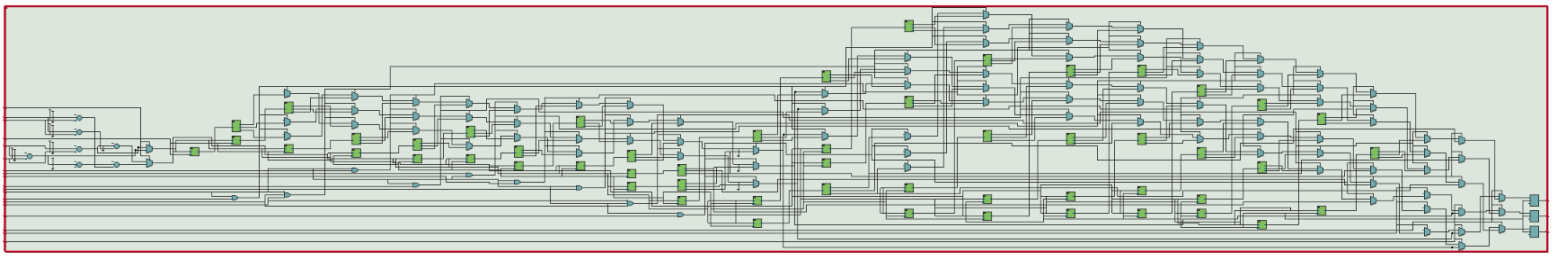


14. Module: playermovement2
    a. **Inputs:** Reset, frame_clk, collision, start, [31:0] keycode
    b. **Outputs:** stand2, crouch2, jump2, kick2, punch2, shoot2, block2, crouchpunch2, [9:0] SpriteX2, SpriteY2, SpriteS2
    c. **Description:** Contains always_ff logic to set output signals based on keycode inputs
    d. **Purpose**: ball.sv module from lab 6.2 but modified to incorporate multiple keycodes, stop an action on release of keypress, and output action signals based on certain keypresses / set x and y motion of player2's sprite.
    e. **RTL Diagram**:

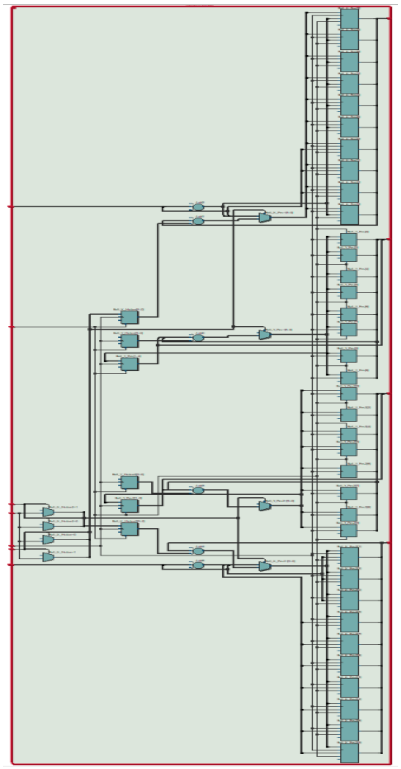


15. Module: player1_example, player2_example
    a. **Inputs:** vga_clk, blank, batchoose, despchoose, wwchoose, zfochoose, stand, crouch, jump, kick, punch, flip, crouchpunch, move, block, dead, [9:0] DrawX, DrawY, spritex, spritey
    b. **Outputs:** [3:0] red, green, blue

c. **Description:** Uses always_ff logic to assign rgb register values based on input signals
d. **Purpose**: Contains all instantiated sprite roms and palettes. Module used to assign properly assign rgb values based on character's action to switch sprites
e. **RTL Diagram**:

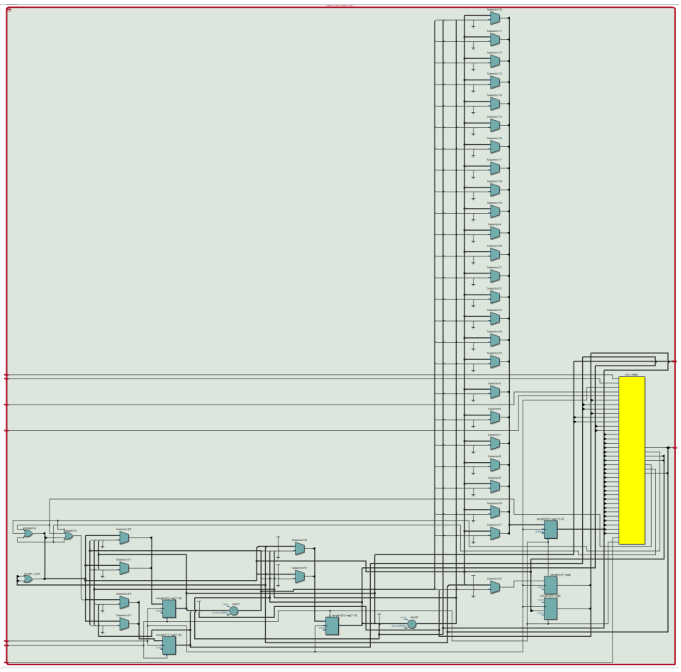


16. Module: cannon
    a. **Inputs:** Reset, frame_clk, ballcollision, ballcollision2, shoot, shoot2,startscreen,flying,flying2,flip, input [9:0] SpriteX, SpriteX2, input [31:0] keycode
    b. **Outputs:** BallX, BallY, BallX2, BallY2
    c. **Description:** Contains always_ff logic to set output signals based on the output from the flyingball state machine and few other conditions.
    d. **Purpose**: This module controls the motion of the special attack flying ball.
    e. **RTL Diagram**:

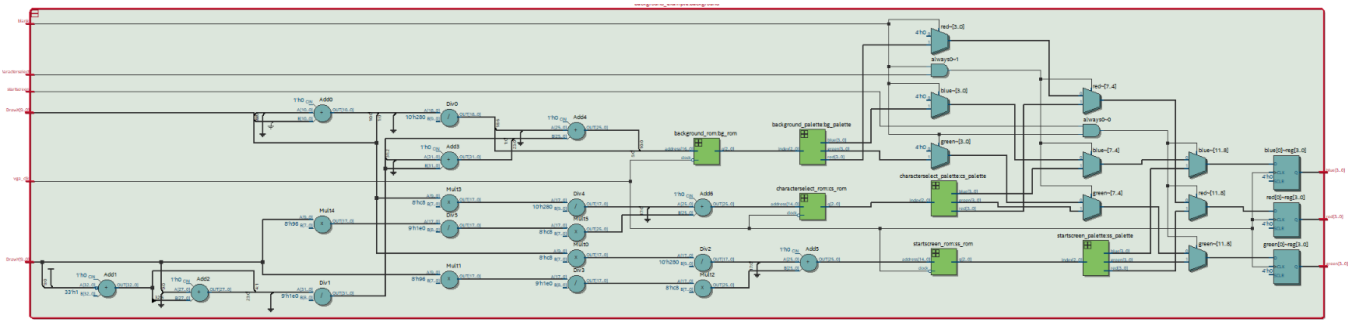


17. Module: healthbar
    a. **Inputs:** pixel_Clk, Reset, kick, punch, collision, ballcollision, block
    b. **Outputs:** length, gameover

c. **Description:** This state machine is created so that the output length is lowered when is is collision between the two players or the cannon from one player hits the other player. It also has a gameover state which is set when the length equals 0.
d. **Purpose**: It determines the length of the health bar and whether or not if the game is over.
e. **RTL Diagram**:



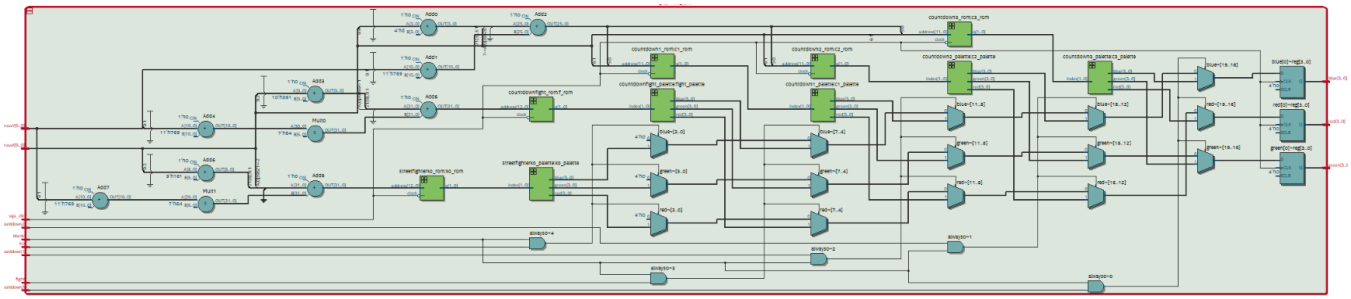
18. Module: background_example
    a. **Inputs:** vga_clk, blank, startscreen, characterselect
    b. **Outputs:** [3:0] red, green blue
    c. **Description:** Uses always_ff logic to assign rgb register values based on inputs
    d. **Purpose**: Holds roms and palettes for the start screen and character select screen and sets rgb values from the corresponding palette based on which one is high.
    e. **RTL Diagram**:



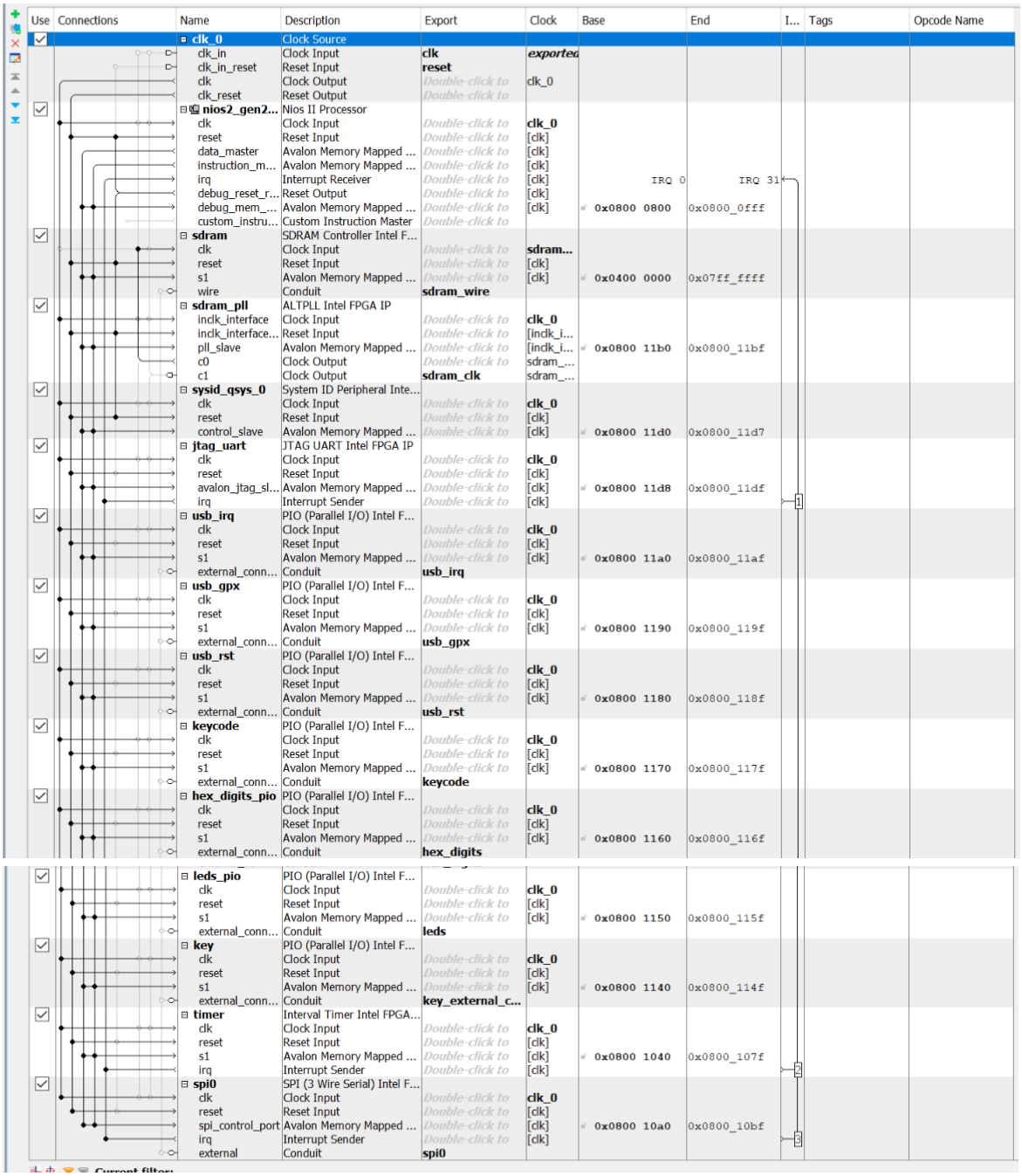
19. Module: graphics
    a. **Inputs:** vga_clk, blank, countdown3, countdown2, countdown1, fight, ko
    b. **Outputs:** [3:0] red, green, blue
    c. **Description:** Uses always_ff logic to assign rgb register values based on inputs

d. **Purpose**: Holds all roms and palettes for the countdown animations before fighting and sets the rgb values to corresponding palette colors based on inputs

e. **RTL Diagram**:



System Level Block Diagram



clk_0: This is the 50MHz clock source that is connected to other blocks on the platform designer

nios2_gen2_0: This is the processor which is the economy version of NIOS II which has instruction and data master that provide the instruction and data to other modules. It handles the compilation of C code.
sdram: The SDRAM is utilized since the on-chip memory has a limited amount of space. This block, just like few other blocks exports a wire ‘sdram_wire’ to the top-level module.

sdram_pll: This block provides the clk signal to the SDRAM block. The clock it generates is different from the controller clk as it is phase shifted by -1ns.

sysid_qsys_0: This component checks whether the software and hardware of this lab is compatible or not.

Jtag_uart: Allows printf statements and is helpful for debugging. Uses the terminal of the host to communicate with the NIOS II processor

Usb_irq: 1-bit interrupt for usb

Usb_gpx: 1-bit usb mux

Usb_rst: 1-bit usb reset signal

Hex_digits_pio: Used for displaying the keycode onto hex digits

Keycode: 32-bit packet that represents the key being pressed on the keyboard, used in the playermovement and playermovement2 modules to control player1 and player2 movement

Timer: module to track time-outs that are necessary for usbs

Spi0: SPI peripheral block with a clock signal connected to the 50 Mhz clock. Communication to this block happens through the provided device driver that is used within the read/write functions in MAX3421E.c where the protocol is used to communicate between the host chip and NIOS II.

Description of the Software Components of the Lab

```
29 //writes register to MAX3421E via SPI
30⊖void MAXreg_wr(BYTE reg, BYTE val) {
31      //psuedocode:
32      //select MAX3421E (may not be necessary if you are using SPI peripheral)
33      //write reg + 2 via SPI
34      //write val via SPI
35      //read return code from SPI peripheral (see Intel documentation)
36      //if return code < 0 print an error
37      //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
38      int ret_code;
39      alt_u8 spi_bytes[2];
40      spi_bytes[0] = reg + 2;
41      spi_bytes[1] = val;
42      ret_code = alt_avalon_spi_command(0x080010a0,0,2,spi_bytes,0,NULL,0);
43      if(ret_code < 0){
44          printf("Error");
45      }
46      return;
47 }
48⊖//multiple-byte write
49 //returns a pointer to a memory position after last written
50⊖BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
51      //psuedocode:
52      //select MAX3421E (may not be necessary if you are using SPI peripheral)
53      //write reg + 2 via SPI
54      //write data[n] via SPI, where n goes from 0 to nbytes-1
55      //read return code from SPI peripheral (see Intel documentation)
56      //if return code < 0  print an error
57      //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
58      //return (data + nbytes);
59      int ret_code;
60      alt_u8 spi_bytes[nbytes + 1];
61      spi_bytes[0] = reg + 2;
62      for(int i = 1; i < nbytes + 1; i++){
63          spi_bytes[i] = data[i - 1];
64      }
65      ret_code = alt_avalon_spi_command(0x080010a0, 0, nbytes + 1, spi_bytes, 0, NULL, 0);
66      if(ret_code < 0){
67          printf("Error");
68      }
69⊖//    ret_code = alt_avalon_spi_command(0x080010a0, 0, nbytes, data, 0, NULL, 0);
70 //    if(ret_code < 0){
71 //        printf("Error");
72 //    }
73      return data + nbytes;
74 }
```

SPI C functions

```
76 //reads register from MAX3421E via SPI
77 BYTE MAXreg_rd(BYTE reg) {
78     //psuedocode:
79     //select MAX3421E (may not be necessary if you are using SPI peripheral)
80     //write reg via SPI
81     //read val via SPI
82     //read return code from SPI peripheral (see Intel documentation)
83     //if return code < 0 print an error
84     //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
85     //return val
86     int ret_code;
87     alt_u8 spi_read[1];
88     alt_u8 spi_write[1];
89     spi_write[0] = reg;
90     ret_code = alt_avalon_spi_command(0x080010a0,0,1,spi_write,1,spi_read,0);
91     if(ret_code < 0){
92         printf("Error");
93     }
94     //printf("%c",spi_read[0]);
95     return spi_read[0];
96 }
97 //multiple-byte write
98 //returns a pointer to a memory position after last written
99 BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
00     //psuedocode:
01     //select MAX3421E (may not be necessary if you are using SPI peripheral)
02     //write reg via SPI
03     //read data[n] from SPI, where n goes from 0 to nbytes-1
04     //read return code from SPI peripheral (see Intel documentation)
05     //if return code < 0 print an error
06     //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
07     //return (data + nbytes);
08     int ret_code;
09     alt_u8 spi_write[1];
10     spi_write[0] = reg;
11     ret_code = alt_avalon_spi_command(0x080010a0, 0, 1 , spi_write, nbytes, data, 0);
12     if(ret_code < 0){
13         printf("Error");
14     }
15     return data + nbytes;
16 }
```

SPI C functions

The MAXreg_wr function takes in BYTE reg and BYTE val as arguments and is responsible for writing val to reg. An array of size 2 is made called spi_bytes to be used as a buffer to pass into the driver. We need to write reg + 2 and val so we put both values into index 0 and index 1. Reg + 2 is put into the buffer instead of reg because the +2 will set the dir bit to 1 indicating that a write is happening. After passing in the values to correct values into the driver it will return an int code that checks if there is no error. The if statement checks if the ret code means error by checking if it's < 0, then print error. The values in the driver, respectively, are the SPI block starting address, 0 for flags, 2 for writing 2 bytes, spi_bytes containing the bytes to write, 0 for bytes to read, NULL for read buffer, and finally 0 for slave select since we only have one slave device.

The MAXbytes_wr is the same thing as MAXreg_wr except now that multiple bytes need to be written. The parameters are BYTE reg, BYTE nbytes, and BYTE* data. Data holds the data that needs to be written and nbytes is the number of bytes that need to be written to reg. This time we need to initialize spi_bytes to a size of nbytes + 1 to account for writing nbytes + the register. In the first location we store reg + 2 then we loop through data to store all nbytes of data in the rest of the spi_bytes buffer. After done, we simply fill in the driver accordingly as explained above and check the ret code to make sure there are no errors. We then return data + nbytes.

The MAXreg_rd function takes in BYTE reg as a parameter. For this, we need to first sent reg through MOSI in order to read the data from the reg on the next 8 cycles of S_CLK. To

do this we have a spi_write and spi_read buffer of both size 1 since we are only doing a single read. We store reg instead of reg + 2 in spi_write since the dir bit should be 0 for a read. The spi_read buffer will then have the value from the register after returning from the driver function. After making sure there is no error we return spi_read[0] to return the value from the register that we wanted to read from.

The MAXbytes_rd function reads multiple bytes from a reg and takes in parameters BYTE reg, BYTE nbytes, and BYTE* data. We write the register through spi_write[1] and read nbytes of data from data through the driver without needing a buffer as data will have the data to read. We then return data + nbytes after checking the ret code to make sure there is no error.

```
128 void setKeycode(DWORD keycode)
129 {
130     IOWR_ALTERA_AVALON_PIO_DATA(0x08001170, keycode);
131 }
132 int main() {
133     BYTE rcode;
134     BOOT_MOUSE_REPORT buf;      //USB mouse report
135     BOOT_KBD_REPORT kbdbuf;
136
137     BYTE runningdebugflag = 0;//flag to dump out a bunch of information when we first get to USB_STATE_RUNNING
138     BYTE errorflag = 0; //flag once we get an error device so we don't keep dumping out state info
139     BYTE device;
140     WORD keycode;
141
142     printf("initializing MAX3421E...\n");
143     MAX3421E_init();
144     printf("initializing USB...\n");
145     USB_init();
146     while (1) {
147         printf(".");
148         MAX3421E_Task();
149         USB_Task();
150         //usleep (500000);
151         if (GetUsbTaskState() == USB_STATE_RUNNING) {
152             if (!runningdebugflag) {
153                 runningdebugflag = 1;
154                 setLED(9);
155                 device = GetDriverandReport();
156             } else if (device == 1) {
157                 //run keyboard debug polling
158                 rcode = kbdPoll(&kbdbuf);
159                 if (rcode == hrNAK) {
160                     continue; //NAK means no new data
161                 } else if (rcode) {
162                     printf("Rcode: ");
163                     printf("%x \n", rcode);
164                     continue;
165                 }
166                 printf("keycodes: ");
167                 for (int i = 0; i < 6; i++) {
168                     printf("%x ", kbdbuf.keycode[i]);
169                 }
170                 setKeycode(kbdbuf.keycode[0] << 24 | kbdbuf.keycode[1] << 16 | kbdbuf.keycode[2] << 8 | kbdbuf.keycode[3]);
171                 printf("\n");
172             }
173
```

An important change we needed to make in the C code from lab 6.2's main() function is to pass in 4 bytes into the setKeycode function since in lab 6.2 we exported 1 byte of keycode so only 1 byte needed to be set whereas for the final project we export 4 bytes to utilize multiple keycodes at once so now 4 bytes of keycode need to be set by the software. We use simple bit shifting logic to pack bytes into a 32 bit WORD as highlighted in blue. In the setKeycode function signature we needed to change the data type of the keycode argument from WORD to DWORD to change from 16 bits to 32 bits unsigned.

Design Resources and Statistics

LUT	8345
DSP	15
Memory (BRAM)	1557504
Flip-Flop	3017
Frequency	122.67 MHz
Static Power	96.18 mW
Dynamic Power	0.70 mW
Total Power	106.19 mW

Explanation of Game Functionality and Design

In our top level file, jltf.sv, we have our vga controller, color mapper, playermovement, playermovement2, cannon, and lab62soc instantiated. The vga controller and lab62 instantiations and ports are kept the same as from lab6.2, except that the soc needed to be regenerated to account for the change from 8 bit keycode width to a 32 bit keycode width. The playermovement and playermovement2 modules are ball.sv from lab 6.2, but modified heavily to account for several different keycodes, velocity changes, and detecting multiple keycodes at once as well. The playermovement modules output x/y velocities and actions based on keycode input. Playermovement corresponds to player1 and playermovement2 corresponds to player2, thus the keycodes used in both modules are different. For example, for player1, Q is kick, W is jump, S is crouch, A is left, D is right, E is punch, F is block, and R is special attack. For player2, O is kick, I is jump, K is crouch, J is left, L is right, U is punch, H is block, and Y is special attack. Based on these keycodes within each module, the output action that pertains to the keypress is set to high as well as their x and y motion variables if there needs to be movement for the action. The action outputs from both playermovement and playermovement 2 are then sent to the player1_example and player2_example modules respectively as inputs. The player1_example and player2_example modules contain all instantiated sprite roms and palettes. In these modules, if an input action is high, then it will set the RGB values to the corresponding action's palette colors. The address into the sprite roms we used is (DrawX-spritex) / 2 + ((DrawY - spritey) / 2) * 64. The formula is defined in raster order and it is multiplied by 64 due to the sprites being 64x64 pixels. The division by 2 is to double the size of the image on the screen, making it 128x128 size on the screen.

The player1_example and player2_example modules are then instantiated within color mapper so that the RGB outputs can be used as outputs within the color mapper module which ultimately sets the RGB values for the game. The color mapper module contains the logic for drawing sprites/shapes as well as setting the RGB values for them. The color mapper module is arguably our most important module as it has the most modules instantiated within it to ultimately determine what should be drawn on the screen.

Our game implemented multiple state machines to handle different tasks. One of the main ones was the initialscreen module which made transitions based on the state of the game. Specifically, the start screen, character selection screen, countdown before fighting, fighting, and KO. Based on outputs from this module, the color mapper uses this in order to know what to draw to the screen based on what state the state machine is in. Another state machine involved was the jump_fsm module which implemented the jumping physics. This module contained several states where the y motion was hardcoded to replicate increasing negative velocity on the way up, to 0 at the top of the jump, to increasing positive velocity on the way down. A LastPressedBat and LastPressedWW module had to be made that stores the jump key as last pressed so that the state machine is able to continue to the end without any breaks. The healthbar module state machine also used similar logic to the jump_fsm where the length of the health bar in each state is hardcoded and will transition to the next state with a smaller length if the opposing player attacks you. Other state machines we created include ballflying, special attack meter and healthbar. The special attack meter and health bar are very similar where we declare output “length” and decrement it by 10 for every collision for health bar and increment by 1 every clock cycle and divide it by 3 for the special attack meter. Ballflying module has “flying” as an output which is used in color_mapper to correctly print the ball when all conditions are met such as the press of key and the range on the screen.

In order to implement multiple keycodes, we utilized a 32-bit keycode register rather than the 8-bit register that was using in lab 6.2. This was done through expanding the keycode PIO in platform designer from 8 bits to 32 bits and regenerating the soc. Since the register now holds 4 keycodes instead of 1, we simply checked each byte in the keycode register if it contains the keycode that a player has pressed. For example, if player 1 hits E, then all 4 bytes will be checked for the keycode for E in order to process that command. That way when both players are playing, this will take care of the keycodes appearing in different combinations of locations within the register. This type of logic will take care of both players being able to play without one player’s action canceling the other’s out. The lastKeyPressedBat and lastKeyPressedWW modules are used for player1 and player2 respectively and is used to handle cases for jumping to the right and left such as W + D and W + A which requires players to press two keys at once to function properly. The combination of outputs of the last key pressed and the current key press are checked for such situations similar to singular key presses to account for the way the keycodes could be stored within the keycode register.

Conclusion

The final project is by far our favorite lab/project that we have worked on this semester. Using the skill and knowledge we've gained throughout the semester, we were able to put everything together to create a fully functional street fighter game which is one of our favorite childhood video game. We were able to successfully implement all baseline functionalities of our game that we proposed alongside one additional feature which is the character selection screen. If we had more time, we would have implemented audio output during attacks or longer background music that would need to utilize the larger SDRAM. We had a difficult time debugging our code during the early stages as well as getting all the sprites and editing them took up a lot of time as well. Some of the difficult parts about making this game was implementing the jumping physics and allowing the player to shoot a ball as part of the special attack. All the game's logic and visuals were done on hardware using SystemVerilog and only used C code from lab 6.2 for the SPI drivers and setKeycode function which is necessary in order for the keyboard to work.