ECE 385

Fall 2023

Final Project

Final Project – Overcooked on FPGA

Aarushi Aggrwal and Maya Moy

Lab Section AT
Alara Tin

Introduction

This final project is based on the videogame Overcooked by Team17, Ghost Town Games. The game is displayed using VGA to HDMI conversion and uses a USB keyboard to control character movement. The player must create onion soup by first grabbing an onion, chopping it, cooking it, and serving it on a plate. Each properly prepared soup earns a point, and the goal is to score as many points as possible within the 3-minute time limit. The functionality for character movement was largely based on the ball movement from Lab 6. This project also incorporated sprite drawing concepts from Lab 7 to draw multiple sprites on the screen, as well as text for the timer and scorekeeping.

Written Description of Final Project Features

The game was drawn as a 640x480 pixel map. The background image used for the map can be seen below.

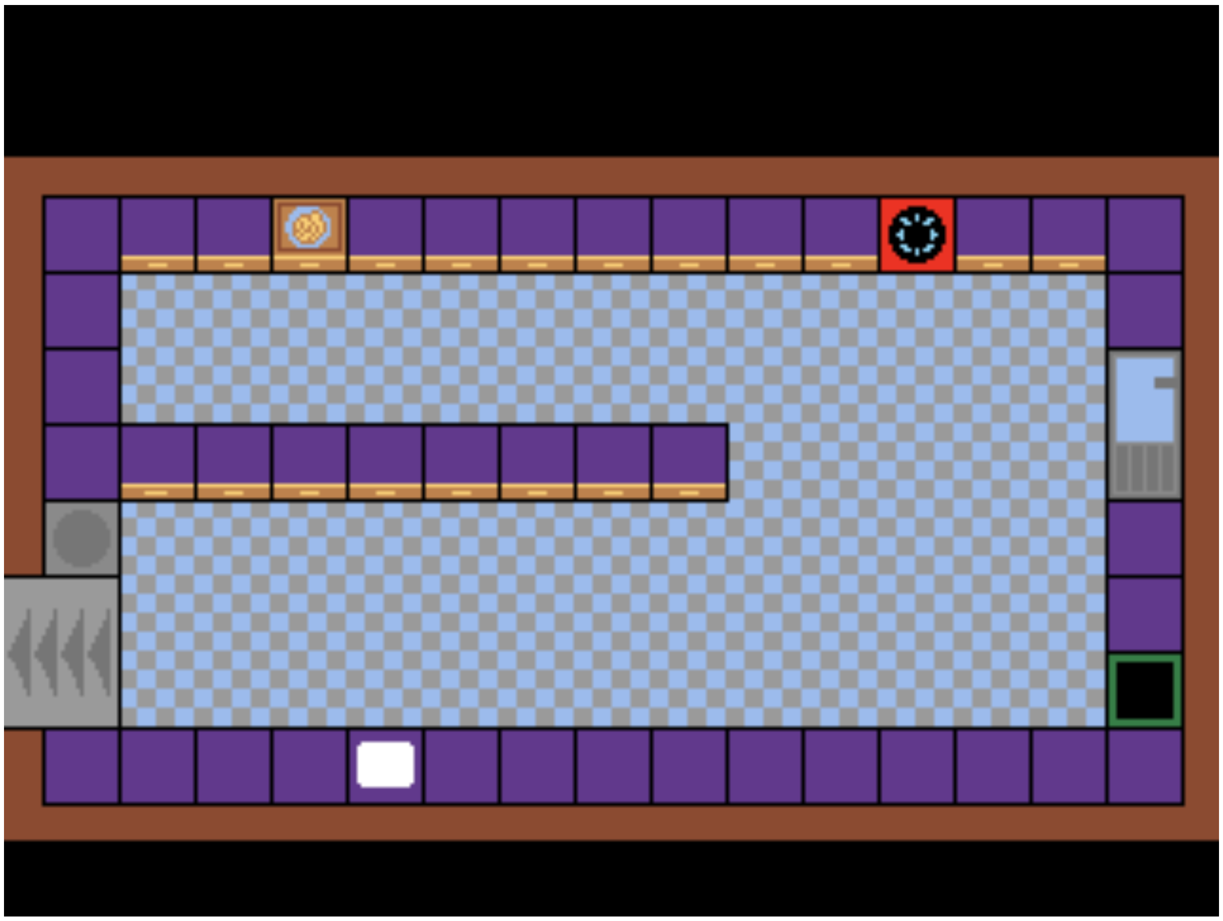
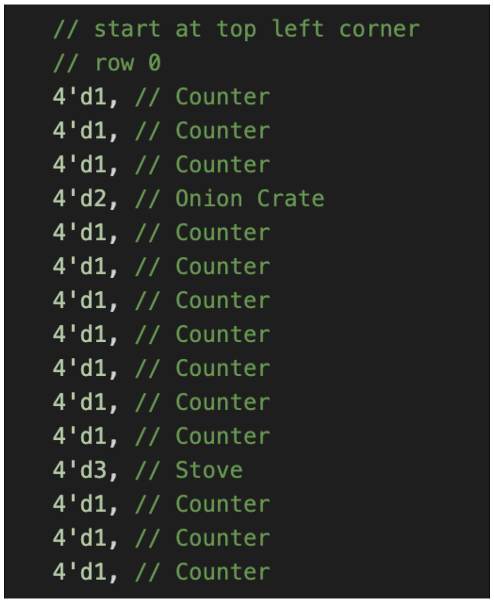


Figure 1 - Game map background image

Within this, there is a smaller area of the map designated for player movement. This area consists of 120 tiles of size 40x40 pixels and can be seen as the area enclosed by the brown boarders in the image below. There are 8 types of tiles, and a ROM file was used to represent this area using the corresponding tile types. This was necessary to determine where sprites could be placed and which events would happen with the sprites. A portion of this ROM file can be seen below.



```
// start at top left corner
// row 0
4'd1, // Counter
4'd1, // Counter
4'd1, // Counter
4'd2, // Onion Crate
4'd1, // Counter
4'd1, // Counter
4'd1, // Counter
4'd1, // Counter
4'd1, // Counter
4'd1, // Counter
4'd1, // Counter
4'd3, // Stove
4'd1, // Counter
4'd1, // Counter
4'd1, // Counter
```

Figure 2 - Example of tile type ROM file

There were many sprites involved in the game, including the character, onion, and plate. In the case of the onion and the plate, the location of these sprites at a given time in the game needed to be stored. Sprites could either be held by a character or placed onto a counter. To track the location of the sprites placed on counters, a set of registers was used to represent the previously described 120 tile game movement area. An index was assigned to each tile to represent any sprites stored on the counter. 0 represented empty, 2 represented plate, and 3 represented onion. So, accessing the registers at a given tile index would return either 0 to represent an empty counter, 2 would represent a plate present on that counter, and 3 would represent an onion.

The game required storing sprite transformation as well, for example, the onion converting to a chopped onion when chopping is completed. So, the implementation had multiple versions in order to represent different states for the objects. The onion had three states - not present, present whole, and present chopped. The plate had two states - empty and filled with soup.

The project includes two categories of features. The first set of features are baseline functionality for the game. The second set of features were added to increase the complexity of the project.

1. Baseline Features

a. Drawing

This project utilized graphics heavily. The Background image is a 640 x 480 image, the character sprites are 40 x 60 images, the food sprites and plate sprites are 40 x 40 images. Graphic control was a crucial part of the project, which was supplemented by the

"Image_to_COE" tool provided by some of the CAs. All the pictures are hand drawn using pixel art apps. The usage of this tool is described in this section later.

b. Motion of the player and actions using keyboard

The starting point to control the motion of the sprite was the ball from Lab 6. WASD keys were used to determine movement direction. However, instead of using the parametric form of a circle, we used the image ROM created using the tool and addressed into it using DrawX and DrawY (current coordinates on the screen). The penguin motion is characterized using penguinX and penguinY, that denote the top left coordinates of the player sprite. The last pressed key was used to determine which direction the player would face. This was important in combination with the nearest tile calculation for determining where the player should place and pick up objects.

The E and Q keys were used to control player actions. E was used to pick up and drop sprites, such as grabbing an onion from the plate, or placing it down on the chopping board. The Q key was used to chop an onion after it was placed on the chopping board. In the case of these action keys, a counter was used to prevent spamming behavior when a key was pressed. Since a user pressing a key once would often trigger multiple keycode signals for a single press, the screen would display the player rapidly placing and dropping the object multiple times. After a key press, no further key presses would be considered for the next 3 clock cycles to prevent this undesired behavior. This mechanism is similar to debouncing switches in previous labs.

c. Collision detection using coordinates

Collision detection served two purposes in the game. The first was to limit player movement to ensure the player stays on the screen. The second was to detect when a player was touching a wall to determine the nearest tile. The drawing of the map was done precisely, and the coordinates of each point were known. Thus, restricting the motion was fairly simple. The penguin's coordinates are bounded by the counter coordinates.

d. Nearest Counter calculation

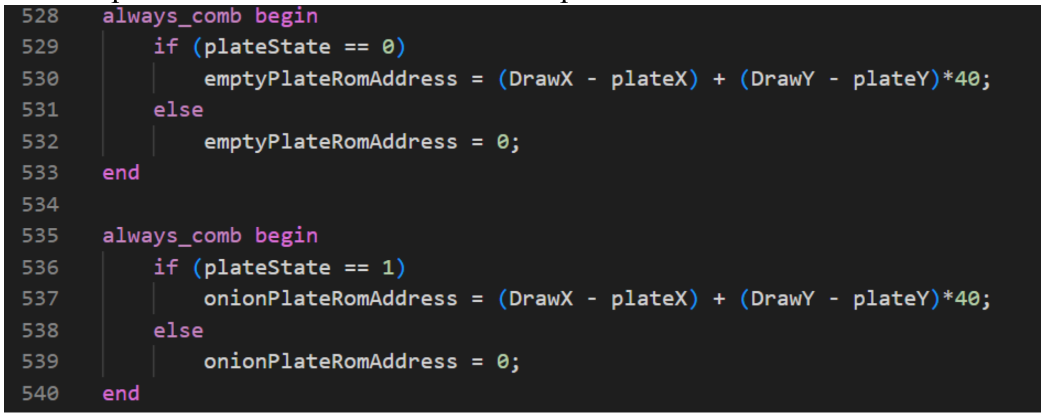
Collision detection would determine if the player was touching a wall. The player X and Y coordinates were then used to calculate X and Y coordinates corresponding to the nearest counter which the player could use to perform an action. In this case, all tile types except for the floor were considered a counter. Another module would convert these coordinates to a tile index, which would then be used to determine the tile type. Once the tile type was determined, there were several different unique scenarios to account for when determining the proper behavior. Some important examples of behavior include -

- A player may place an onion or plate on any plain counter or cutting board
- The plate cannot be placed in the trash, but the onion can be
- The cutting action can only be performed when an onion is on the cutting board tile

e. Sprite transformation

Each sprite has the following associated with it.

- spriteState: Each sprite is characterized by a certain number of states it can be in. These are integer values from 0 to 2 (can be extended as per requirement). The onion has a "not present" state (state 0), a "whole onion" state (state 1) and a "chopped onion" state (state 2). The soup plate has an "empty plate" state (state 0) and a "full plate" state (state 1). This feature helped control the drawing of sprites in a systematic way. ROM addresses for the sprite drawing are put through a mux. Below is an example of how the ROM addresses for plate are set:



```
always_comb begin
    if (plateState == 0)
        emptyPlateRomAddress = (DrawX - plateX) + (DrawY - plateY)*40;
    else
        emptyPlateRomAddress = 0;
end

always_comb begin
    if (plateState == 1)
        onionPlateRomAddress = (DrawX - plateX) + (DrawY - plateY)*40;
    else
        onionPlateRomAddress = 0;
end
```

Figure 3 - Code snippet to select the ROM address for each version of the sprite

- spriteX and spriteY: Top left coordinates of the sprite that are initialized to certain values based on different cases. In the picking up case, the sprite coordinates are set to the penguin's coordinates (offset by a constant value to make it appear as if the character is holding it). In the dropping something on a counter case, the sprite coordinates are set to the nearest counter's coordinates.

f. Order drawing at the top

Drawing the order was similar to the rest of the sprites. The only different thing is that it remains static. In the prototype version, we only did one dish per level which is fixed. This order sprite was drawn on the top right of the screen and was used to check the delivered orders. If they matched the order on top, a point is awarded; otherwise, the order is rejected.

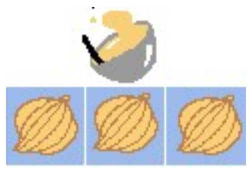


Figure 4 - Order sprite example

2. Additional Features

a. Timer

A single level is 3 minutes long. This timer was implemented by maintaining a counter that resets to zero whenever the level starts. Every clock cycle, the counter is incremented. The clock used in this particular always_ff block is vsync, which is 60 Hz. Thus, 180 seconds is 10800 clock cycles.

The timer also outputs a startFlag and endFlag. The start screen is displayed when startFlag is low and endFlag is low. As soon as the keycode to select a character is pressed, startFlag is set to high. The counter starts counting. When it hits 10800, endFlag is set to high and startFlag is set to low. Now, the end screen is displayed.

This module is also responsible for drawing the timer by converting the counter to minutes and seconds. Then, each number is read and used to index into the fontROM file for numbers. For this project, the bitmaps provided for lab 7 were expanded using a Python script. This fontRom file only contains bitmaps for numbers 0 through 9 and a colon to draw between the minutes and seconds.

b. Start and End screen

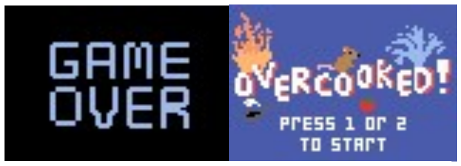


Figure 5 - Start and end screen examples

As above, the start and end flags generated while computing the timer signify when the start screen and end screen should be drawn. In the module that does the background drawing, the RGB values of the background are set to the start screen or end screen's values based on the flags from the timer.

c. Character Selection (Controllable parameter)

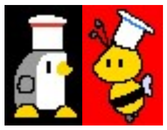


Figure 6 - Character sprites

The character can be selected by the user on the start screen. If 1 is pressed, the penguin is selected. If 2 is pressed, the bee is selected. If any other key is pressed, the game defaults and selects a penguin. This is done by reading the keycode and assigning the signal "character". A penguin is drawn if character is 1 and a bee is drawn if character is 2.

d. Scorekeeping

Score is incremented every time a full plate is put on the "delivery" counter. The sprite that the character is holding right before putting the plate down is examined. If it matches what is on the order, the score is incremented. This score is displayed in a similar fashion to the timer. The same fontRom is used to display the score.

e. Status Bar

The status bar is initialized to empty. It is used to signify progress on activities like chopping an onion. A special counter (qcounter) is used to track how many times the action key (Q) is pressed. Each key press increases the status bar. After 4 key presses, the status bar is full, and the sprite gets transformed. In our baseline design, when the onion is brought to the chopping board, the user is required to press Q four times to completely chop the onion.

3. Portions of lab 6 used:

The entire gameplay was controlled using a USB keyboard. The driver for the USB keyboard is drawn from lab 6. The keycode reading and SPI protocol are implemented through the block diagram that was created in lab 6.2. Bound checking is done a little differently than the ball because the sprites in the game don't necessarily bounce back when they encounter a wall – they simply remain at that same location. The sprite drawing is also similar to the lab 6.2 – only that instead of drawing a circle with fixed colors at all times, a ROM of pixels and a color palette is indexed into to draw sprites.

4. Portions of lab 7 used:

Text drawing was done in a simpler way than lab 7 since the text didn't come from a software program. However, indexing into the fontROM file and reading the bitmaps to draw the numbers was exactly like lab 7. A new bitmap was created to draw 16x32 big characters rather than 8x16 ones from the lab. This was accomplished using a python script.

5. Creating COE files and color palette using "Image_to_COE" tool

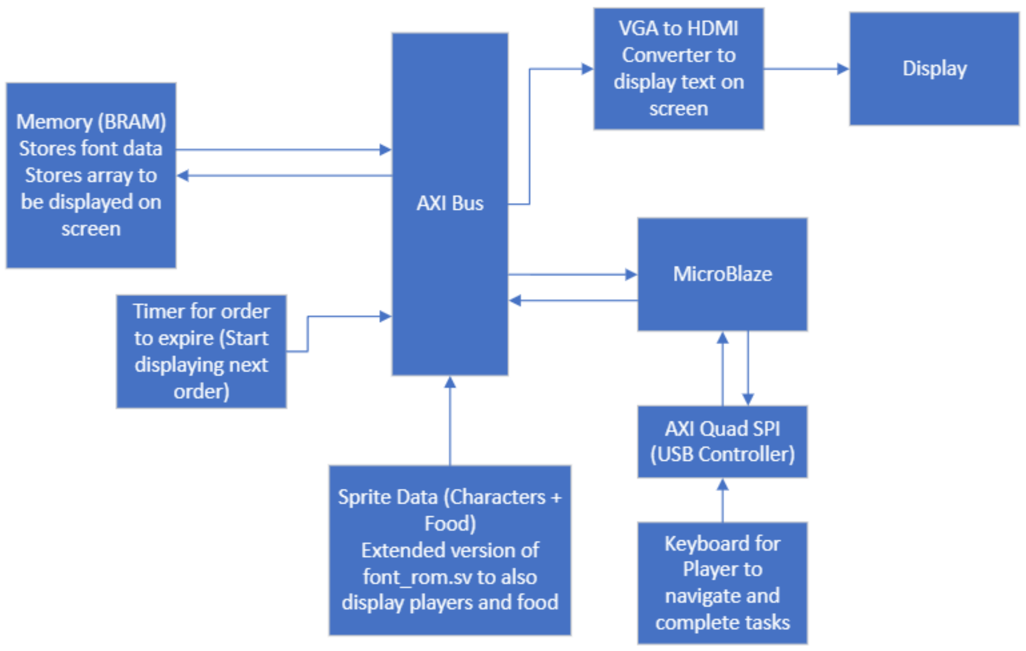
Here is a link to the tool that was used to convert pixel images to COE files and palettizing: https://github.com/amsheth/Image_to_COE/tree/master. The python script inputs an image, the desired horizontal and vertical resolution, and the number of bits to store the colors (2 for 4 colors, 3 for 8 colors, etc.).

6. How to use COE files / ROM to display sprites

All sprites in the game were created from hand drawn images. The image to COE conversion tool was provided by the CAs and allowed the sprites to be drawn within the game. Using the tool and displaying the sprite within the game involved the following components -

1. An on-chip memory block was used to store the data from the COE file. This memory block was set to always enabled as the data must be continuously read to continuously display the proper images on the screen.
2. A rom file was used for accessing this memory.
3. A palette file contained the 12-bit RGB color mappings for the image.
4. A top-level file was needed to instantiate the rom file and access the data from the COE file as well as the palette.

Block Diagram
A high-level breakdown of the project and its parts is seen below.



VGA to HDMI Converter to display text on screen
Display
Memory (BRAM) Stores font data Stores array to be displayed on screen
AXI Bus
MicroBlaze
Timer for order to expire (Start displaying next order)
AXI Quad SPI (USB Controller)
Sprite Data (Characters + Food) Extended version of font_rom.sv to also display players and food
Keyboard for Player to navigate and complete tasks

Figure 7 – High-level flow of project

Below is the block diagram from Vivado that implements the SPI protocol and helps the keyboard interface with the microprocessor, MicroBlaze.

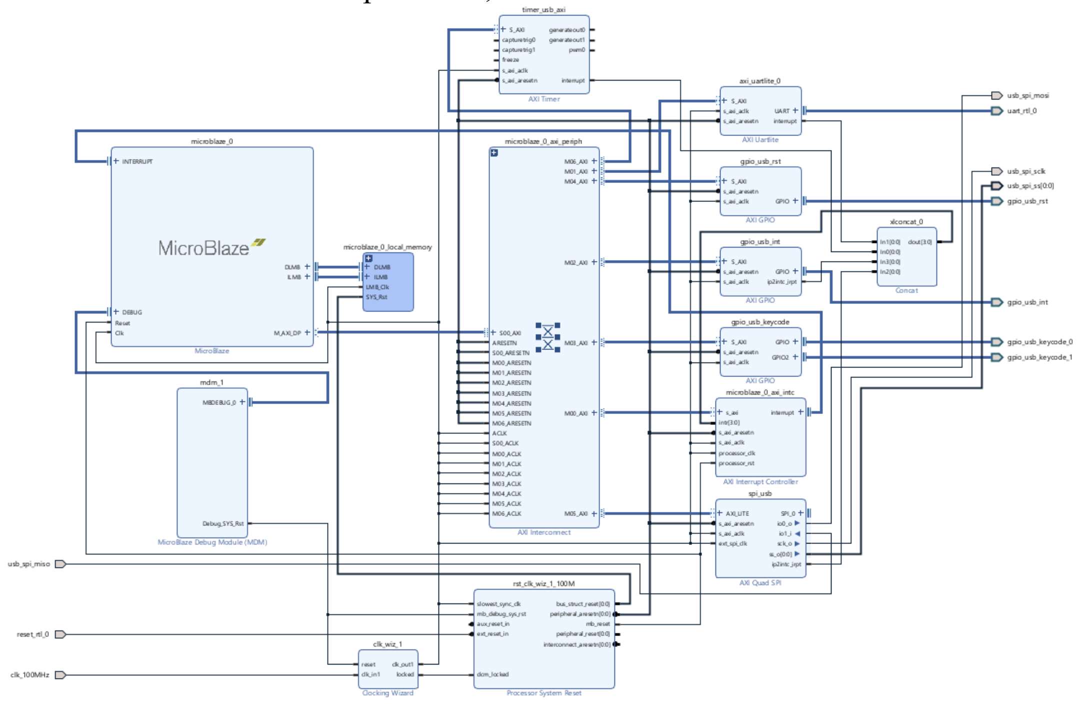


Figure 8 - Block Diagram from Vivado

Below is the elaborated design for the modules in our project. It is worth nothing that most of the design is inside the backgroundImage_example module, the expansion of which is shown later.

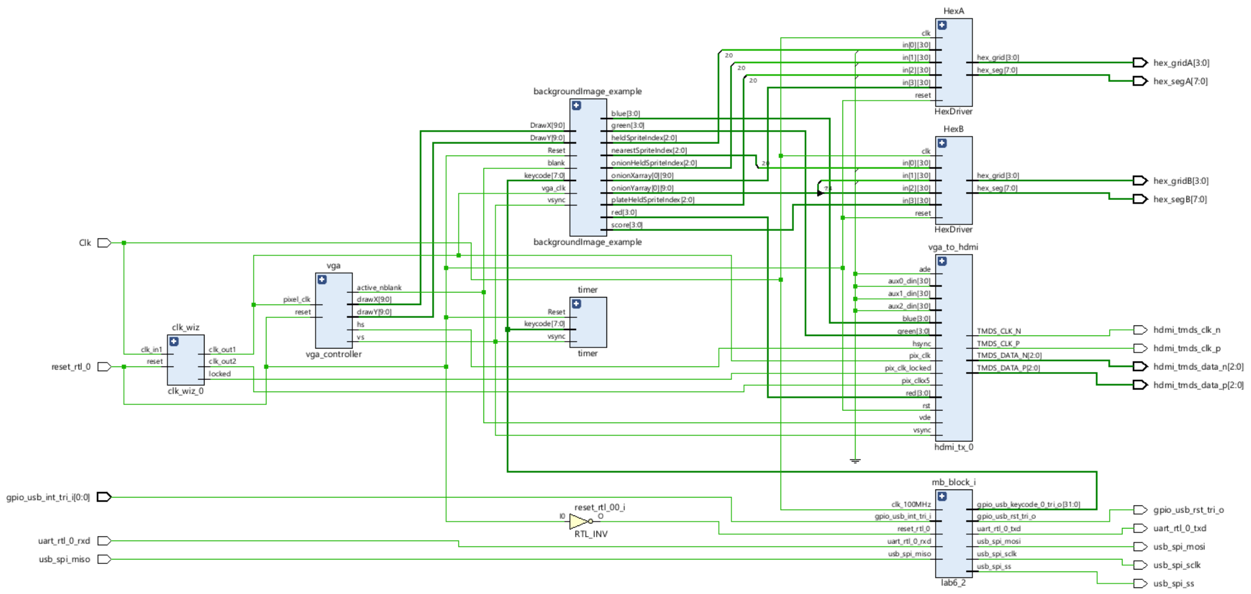


Figure 9 - Elaborated design from Vivado demonstrating the top level design

Finally, below is a block diagram of how the logic inside the backgroundImage_example is implemented to facilitate drawing on the HDMI screen.

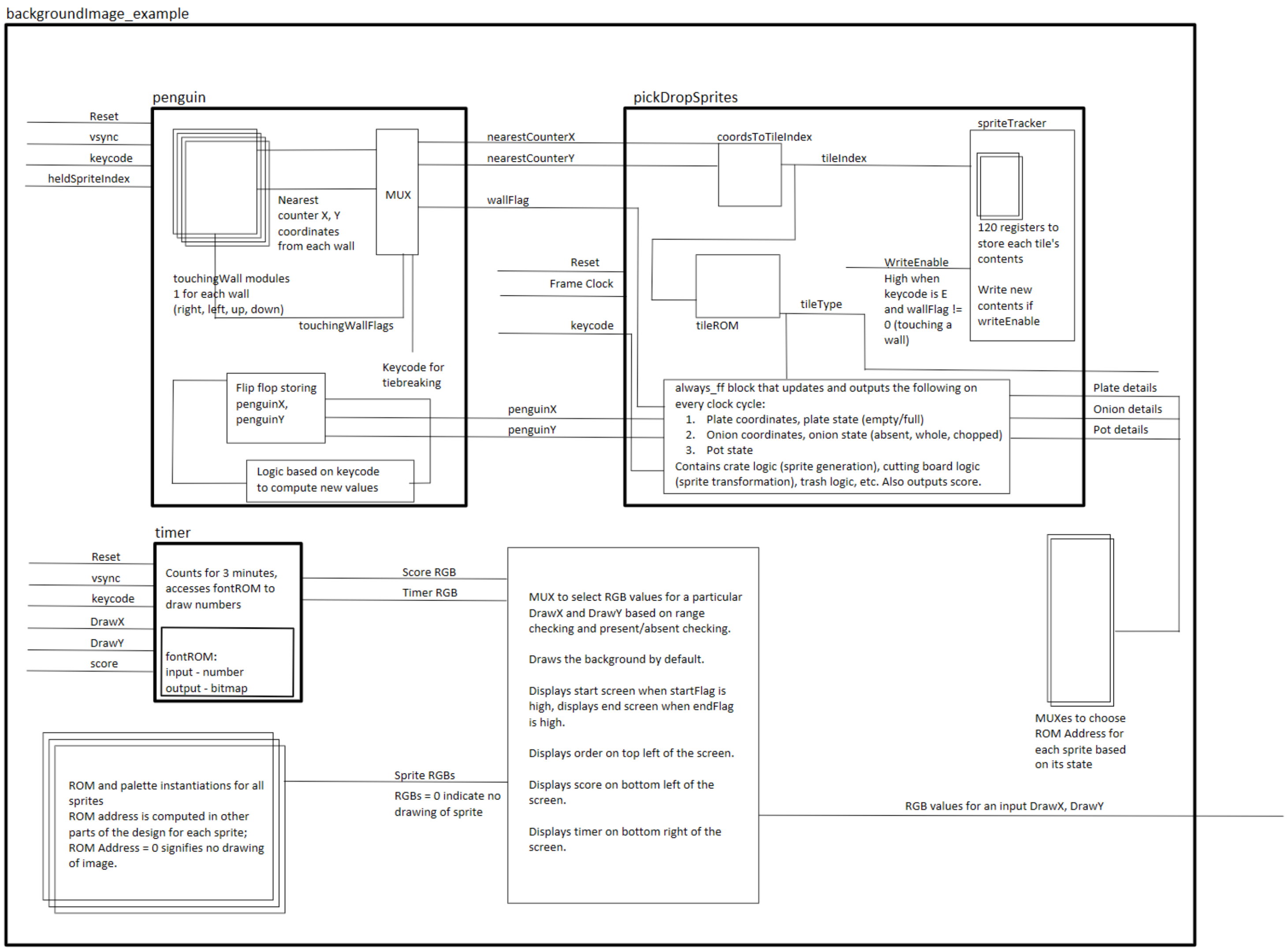

backgroundImage_example
penguin
Reset
vsync
keycode
heldSpriteIndex
Nearest counter X, Y coordinates from each wall
MUX
touchingWall modules 1 for each wall (right, left, up, down)
touchingWallFlags
Keycode for tiebreaking
Flip flop storing penguinX, penguinY
Logic based on keycode to compute new values
nearestCounterX
nearestCounterY
wallFlag
penguinX
penguinY
pickDropSprites
coordsToTileIndex
tileIndex
spriteTracker
120 registers to store each tile's contents
Write new contents if writeEnable
WriteEnable High when keycode is E and wallFlag != 0 (touching a wall)
Reset
Frame Clock
keycode
tileROM
tileType
always_ff block that updates and outputs the following on every clock cycle:
1. Plate coordinates, plate state (empty/full)
2. Onion coordinates, onion state (absent, whole, chopped)
3. Pot state
Contains crate logic (sprite generation), cutting board logic (sprite transformation), trash logic, etc. Also outputs score.
Plate details
Onion details
Pot details
timer
Reset
vsync
keycode
DrawX
DrawY
score
Counts for 3 minutes, accesses fontROM to draw numbers
fontROM: input - number output - bitmap
Score RGB
Timer RGB
MUX to select RGB values for a particular DrawX and DrawY based on range checking and present/absent checking.
Draws the background by default.
Displays start screen when startFlag is high, displays end screen when endFlag is high.
Displays order on top left of the screen.
Displays score on bottom left of the screen.
Displays timer on bottom right of the screen.
MUXes to choose ROM Address for each sprite based on its state
ROM and palette instantiations for all sprites
ROM address is computed in other parts of the design for each sprite;
ROM Address = 0 signifies no drawing of image.
Sprite RGBs
RGBs = 0 indicate no drawing of sprite
RGB values for an input DrawX, DrawY

Figure 10 - Block diagram of backgroundImage_example showing its logical design

Block Diagram Components

1. **Microblaze**
   The MicroBlaze embedded processor is a reduced instruction set computer (RISC) core. It can be configured to act as a 32-bit processor ot a 64-bit processor as per the user's requirements. It has some fixed features like thirty-two general purpose registers (each register 32- or 64-bit as configured), 32-bit instructions, 32-bit address bus and a single-issue pipeline.

2. **Microblaze Local Memory**

This block component serves as storage for data and program instructions. In this lab, it was set to 32KB. It connects to the MicroBlaze through the ILMB (instruction local memory bus) and the DLMB (data local memory bus).

3. **Microblaze Debug Module**
   This module enables the interface to MicroBlaze processor for debugging. With this module, the user can debug the processor using the Xilinx System Debugger.

4. **Clocking Wizard**
   Generates two clock signals with different frequencies (25MHz and 125 MHz). The 25MHz clock is used for the VGA controller, while both frequencies are used for the VGA to HDMI converter.

5. **Processor System Reset**
   This module uses the external reset signal from the FPGA board and synchronizes it with the clock. It generates synchronous reset signals for all parts of the circuit.

6. **AXI Interconnect**
   The AXI Interconnect is an IP that connects one or more memory-mapped Master devices to one or more memory-mapped Slave devices. For this lab, the AXI4-Lite control register interface was utilized.

7. **AXI Uartlite**
   The AXI Universal Asynchronous Receiver Transmitter Lite core transmits data between UART signals and the AXI bus. It interfaces with the AXI4-Lite protocol. This allows the user to access registers and transfer data.

8. **AXI Interrupt Controller**
   Using the interrupt signal from one or multiple sources (one source in week 1, multiple sources from the concat module in week 2) and provides a single interrupt signal to the processor.

Module Descriptions

1. **backgroundImage_example.sv**

   Inputs: vga_clk, Reset, blank, vsync, [7:0] keycode, [9:0] DrawX, DrawY,

   Outputs: [9:0] penguinXOut, penguinYOut, nearestCounterX, nearestCounterY, onionXarray[1], onionYArray[1], [3:0] red, green, blue, tileType, score, [6:0] tileIndex, [2:0] heldSpriteIndex, nearestSpriteIndex, onionHeldSpriteIndex, plateHeldSpriteIndex, [1:0] presentMask[1]

   Description: Controlls all the drawing to the display. Instantiates the rom and palette for each image file for bacgkground, sprites, and start / end screens. Takes in DrawX and DrawY coordinates to select the proper RGB values for each pixel on the screen. Based on pixel location, determines if pixel should represent the background or a sprite. Implements start, end, and character flags to determine when to display the start screen and end screen, and which character to display based on user selection.

Purpose: Draws the entire game display, including background, spites, and start and end screens. Also controls character selection.

2. coordsToTileIndex.sv

Inputs: [9:0] xCoordinate, yCoordinate

Outputs: [6:0] tileIndex

Description: Takes an x and y coordinate and calculates which tile number that coordinate belongs to with the playable area on the screen.

Purpose: Calculates tile index based on position, which is used as a helper for other modules to determine what type of tile the character is near, and if that tile has any sprites on it such as a plate or onion.

3. fontRom.sv

Inputs: [8:0] addr

Outputs: [15:0] data

Description: Contains bitmaps for drawing the characters 0-9 and :. Takes in an address calculated externally and indexes to the proper 16 pixel wide row of font data.

Purpose: Information for how to draw characters 0-9 and : for the timer and score displays

4. hex.sv

Inputs: Clk, Reset, [3:0] in[4]

Outputs: [7:0] hex_seg, [3:0] hex_grid

Description: Takes 4x4-bit input values and uses the nibble_to_hex module to determine how to map each hex digit to a 7-segment LEDs. 4 separate 7-segment LEDs are used to display all the register values, so this module also determines which of the 4 LEDs to write each new hex value to for display. hex_grid determines which of the 4 LEDs will be used to display the current register. hex_seg contains the actual bitmap to display a hex number on a 7-segment LED.

Purpose: The hex driver displays the hex values for the output value from any program on the FPGA LEDs. It converts the hex number to a 7-bit value that can be displayed on the 7-segment displays.

5. mb_usb_hdmi_top.sv

Inputs: Clk, reset_rtl_0, uart_rtl_0_rxd

Outputs: uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, hdmi_tmds_data_p

Description: This module is responsible for creating an instance of the lab7 block module containing all the modules for the microblaze and the actual top level that sends DrawX and DrawY signals. It integrates the hdmi signals with the uart, clock, and reset signals.

Purpose: Top level module for instantiating main lab7 block module.

6. nibble_to_hex.sv

Inputs: [3:0] nibble

Outputs: [7:0] hex

Description: Takes the binary nibble for a number, determines the 0-F hex equivalent, then chooses the proper bitmap needed to display this hex number on a 7-segment LED.

Purpose: Converts the input value into a bitmap to show the hex value on a 7-segment LED display. Submodule to HexDriver.

7. penguin.sv

Inputs: Reset, frame_clk, [7:0] keycode, [2:0] heldSpriteIndex

Outputs: wallFlag, [9:0] penguinX, penguinY, nearestCounterXOutput, nearestCounterYOutput

Description: Takes the inputed keycode to determine the direction the penguin / character should move, determines the new X and Y position of the character, and outputs the new coordinates. Based on the new position, also calculates the coordinates of the nearest counter as well as whether the character is currently touching a wall.

Purpose: Controls movement of the character sprite, as well as determining if it's touching and counter and the nearest counter's coordinates

8. pickDropSprites.sv

Inputs: Reset, frame_clk, wallFlag, [7:0] keycode, [9:0] penguin, penguin, nearestCounterX, nearestCounterY

Outputs:[2:0] nearestSpriteIndex, heldSpriteIndex, [3:0] tileType, [6:0] tileIndex

Description: Takes the current character position, and if the player presses the E key, first checks if the player is touching a wall, and if so determines if there is a sprite on that counter. Also tracks if the character is currently holding a sprite. Based on this, either places the currently held sprite on the counter if it's empty, or picks up the sprite on the counter if holding nothing are empty.

Purpose: Controls picking up and dropping sprites for the plate and the onion, including getting an onion from the crate. Ensures plates and onions cannot be placed on each other, and which tile types they can be placed on.

9. spriteTracker.sv

Inputs: Reset, writeEnable, clk, respawnPlate, correctOrder, [6:0] tileIndex, [2:0] spriteIndexIn

Outputs: [2:0] spriteIndex

Description: If the write enable signal is high, writes the incoming sprite index into the incoming tile index to track which locations on the screen have sprites on them. Reads always; meaning, checks the input tile index and outputs which sprite, if any, is located there.

Purpose: Tracks when and where onions and plates are placed on the screen as they are picked up, moved around, and placed back down.

10. tileROM.sv

Inputs: [6:0] tileIndex

Outputs: [3:0] tileType

Description: Register file that stores the usable area within the background map by dividing it into 40x40 pixel tiles. Each tile is assigned an index corresponding to what type of tile it is – cutting board counter, counter, onion crate, stove, trash, floor, vents, sink, etc.

Purpose: Helper module for picking up and placing down sprites by validating what type of counter the nearest counter is and whether a sprite can be placed there / what behavior can happen when a sprite is placed there.

11. timer.sv

Inputs: Reset, vsync, [3:0] score, [7:0] keycode, [9:0] DrawX, DrawY

Outputs: StartFlag, EndFlag, [13:0] timerValue, [3:0] minuteOnes, secondTens, secondOnes, timerRed, timerGreen, timerBlue, scoreRed, scoreGreen, scoreBlue

Description: Uses vsync clock to count the number of clock cycles that have passed and convert this to a number for the timer in terms of minutes and seconds. Uses the number of each digit to index into font rom and display the timer on the screen.

Purpose: Controls the timer displaying showing the remaining time in the level

12. touchingDownWall.sv

Inputs: [9:0] penguinX, penguinY

Outputs: touchingDownWallFlag, [9:0] nearestCounterX, nearestCounterY

Description: Based on the penguinX and penguinY character coordinates, determines if the character is touching any walls below it.

Purpose: Determines if character it touching a wall below it, used to determine nearest counter positions and pick drop logic

13. touchingLeftWall.sv

Inputs: [9:0] penguinX, penguinY

Outputs: touchingDownWallFlag, [9:0] nearestCounterX, nearestCounterY

Description: Based on the penguinX and penguinY character coordinates, determines if the character is touching any walls to the left of it.

Purpose: Determines if character it touching a wall to the left of it, used to determine nearest counter positions and pick drop logic

14. touchingRightWall.sv

Inputs: [9:0] penguinX, penguinY

Outputs: touchingDownWallFlag, [9:0] nearestCounterX, nearestCounterY

Description: Based on the penguinX and penguinY character coordinates, determines if the character is touching any walls to the right of it.

Purpose: Determines if character it touching a wall to the right of it, used to determine nearest counter positions and pick drop logic

15. touchingUpWall.sv

Inputs: [9:0] penguinX, penguinY

Outputs: touchingDownWallFlag, [9:0] nearestCounterX, nearestCounterY

Description: Based on the penguinX and penguinY character coordinates, determines if the character is touching any walls above it.

Purpose: Determines if character it touching a wall above it, used to determine nearest counter positions and pick drop logic

16. VGA_controller.sv

Inputs: pixel_clk, reset

Outputs: hs, vs, active_nblank, sync, [9:0] drawX, drawY

Description: Controls the VGA display by mapping the input drawX and drawY coordinates to the pixel within the VGA's array of representing the screen and outputting the correct vertical or horizontal synchronizations signals.

Purpose: Synchronizes signals for the VGA for use with the VGA HDMI converter.

Design Resources and Statistics
After running implementation on the top level in Vivado, design resource statistics were collected from the utilization and power reports. This information is recorded in the table below.

LUT	4979
DSP	14
Memory (BRAM)	27
Flip-Flop	3032
Latches	0
Frequency	100.664 MHz
Static Power	0.077 W
Dynamic Power	0.413 W
Total Power	0.489 W

Table 1: Design resources table from Vivado implementation

Conclusion

The baseline version of the project worked successfully. Picking up and dropping sprites and the nearest counter calculation were the pillars of our project. The design is scalable now and can be extended to multiple levels and multiple dishes.

Our biggest challenge was coming up with a systematic and efficient way to pick and drop sprites. We began with a separate module for each sprite (onion.sv, plate.sv, pot.sv, etc.). However, we later realized that different modules cause timing issues. This was identified because of the observation of some unpredictable and random glitches in the design. These timing issues were resolved by consolidating the design into a single module.

Debugging this design was most efficiently done by displaying flags on the screen since it was heavily based on graphics. The timing was not an issue for most of the project since the graphics were optimized.

Conclusively, the design was a success. The biggest advantage of our final design was its scalability.