EE 354 Final Project Report

S. Burton

github.com/54mb/VHDL-Donkey-Kong

Project Description:

Goal:
To create a fun-to-play Donkey Kong Mini-Game.
This will be written in Verilog, will run on the Nexys 3 board, and will be shown on the monitor using a VGA connection.

Limitations:
To make this goal achievable, I will limit the game to one obstacle at a time, and only one map. The screen will be drawn using 3-bit colour to make the scanning and sprite drawing more simple.

Gameplay:
The goal of the game is to reach the top platform (the one with the Princess) without getting hit by a barrel. The player will have three buttons to control their movement: left, right, and jump;

There will be a single moving obstacle in the game, a barrel which will roll down the platforms towards the player.
If the barrel hits the player, they will lose a life, and the barrel will slow down by one speed increment.
If the barrel does not hit the player, and reaches the bottom of the map, it will reappear falling from the top of the screen.

If the player reaches the top platform, they will receive a bonus score before being put back to the start of the map. The barrel speed will also increase by one speed increment.
The map will be made up of platforms, which the player will be able to jump up through, but not fall down through. This makes collision detection slightly easier.

Scoring and Lives:
The player will start off with three lives, and a score of 0.
If they are hit by a barrel, they will lose a life or die if no lives left.
If they make it up a platform level they will receive a small score.
If they make it all the way to the top, they will receive a large score.

Game Design:
The game will look as close as possible to the original Donkey Kong as possible.
The sprites will be drawn to the screen in 3-bit colour. Mario, Peach, the barrel, and the platforms can all be drawn easily. Donkey Kong himself may not be possible to draw convincingly since there is no way to create multiple shades of brown with only 3-bit colour.
Score and lives will be shown in the top corner. Score will be a 5-digit decimal number. The player will start with 3 lives. The game will end when the player dies with no lives left.
To add some flavour, the score will flash red-green-blue when the player earns score, and will count up in real time to give the player a sense of achievement.
If the player dies, they will become invincible for a short time, and will flash visible-invisible for this time to show it.

Implementation: Sprites

Drawing a Sprite:
To make the game look nice, sprites will need to be drawn on the screen.
The sprites will be stored as 2D bitmaps in the form of wire arrays.

By default, the VGA control scans across the screen with CounterX and CounterY. This creates pixels that are too small to properly see using sprites as small as we use.
To make the pixels larger, we assign ScaleX[8:0] = CounterX[9:1], and ScaleX[8:0] = CounterX[9:1]. This doubles the size of the pixels displayed with much less effort than creating larger sprites.

To show the sprite on the screen, we must send the appropriate RGB combination to the VGA for each pixel. The pixel we are meant to be drawing is at position(CounterX, CounterY), where (0,0) is the top left corner of the screen. Our adjusted coordinate system (ScaleX, ScaleY) represents the same, but half the resolution to increase the size of the sprites.

Sprite Scanning:
For each sprite that needs to be drawn, we check to see if the pixel being drawn is within the sprite's border.
If it is, then we index the sprite's data to see if we need to set the R/G/B bit. The index if found by subtracting the position of the sprite, from the position of the counter.

For the heart sprites, we also check if health is above a threshold before drawing. Below is the scanning for the red component of the first heart sprite.

```
heartR = 0;
if (ScaleX >= 10 && ScaleX < 19 && ScaleY >= 30 && ScaleY < 40 && health > 0)
      heartR = heartR | hSpriteR[ScaleY-30][ScaleX-10];
```

The value heartR is then OR'ed with all the other red sprite components to produce vga_r, which is read by the VGA control module.
The same is done for all sprites, and for all three colours. Since some sprites don't use all three colours, we don't need to OR them all in. The platforms are handled in a for loop, since we must refresh the screen very quickly.

```
vga_r <= inDisplayArea & (heroR | scoreR | peachR | heartR | barrelR | platR);
vga_g <= inDisplayArea & (heroG | scoreG | peachG | heartW | barrelG);
vga_b <= inDisplayArea & (heroB | scoreB | peachB | heartW);
```

Sprite Flashing:
To make the hero and score flash was surprisingly simple.
For the hero, we just need to show/hide when the player has just died. To achieve this, we set a wide register to some positive value when the player dies. Then decrement the register's value every frame of the game. Then we can OR the value of the three heroRGB bits with one of the bits of the register to create a flashing effect.

```
heroR = heroR & ~justDied[4];
heroG = heroG & ~justDied[4];
heroB = heroB & ~justDied[4];
```

For score, we check to see if score is currently counting up (totalScore > ourScore). If it is, then we use DIV_CLK[26:25] to tell is which colour(s) to block out. This blocking makes the colours change.

```
if (totalScore > ourScore) begin
      if (DIV_CLK[26:25] == 0) begin
            scoreG = 0;
            scoreB = 0;
      end
      if (DIV_CLK[26:25] == 1) begin
            scoreR = 0;
            scoreB = 0;
      end
      if (DIV_CLK[26:25] == 2) begin
            scoreG = 0;
            scoreR = 0;
      end
      if (DIV_CLK[26:25] == 3) begin
            scoreG = 0;
      end
end
```

Score Sprites:
To show the correct digit for each of the scores, we do the same as we would for any other sprite, with one extra step. We 'try' setting the bit based on each possible sprite the score digit could use, and then only allow the bit to be set when the value of the digit matches the sprite.

```
scoreR = 0;
if (TitleX >= SCORE_X && TitleX < SCORE_X+6
    && TitleY >= SCORE_Y && TitleY < SCORE_Y+6) begin
      scoreR = scoreR | ((digit5 == 0) & number0[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 1) & number1[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 2) & number2[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 3) & number3[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 4) & number4[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 5) & number5[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 6) & number6[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 7) & number7[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 8) & number8[TitleY-SCORE_Y][TitleX-SCORE_X]);
      scoreR = scoreR | ((digit5 == 9) & number9[TitleY-SCORE_Y][TitleX-SCORE_X]);
end
```

Sprite Data:
The red layer of the heart sprite is shown below, with the ones highlighted to show the pattern.
The heart sprite was 9 bits wide, by 10 bits deep, and was one of the smallest sprites used.

```
wire[8:0] hSpriteR[9:0];
assign hSpriteR[0] = 9'b 011000110;
assign hSpriteR[1] = 9'b 111101111;
assign hSpriteR[2] = 9'b 111111111;
assign hSpriteR[3] = 9'b 111111111;
assign hSpriteR[4] = 9'b 111111111;
assign hSpriteR[5] = 9'b 111111111;
assign hSpriteR[6] = 9'b 011111110;
assign hSpriteR[7] = 9'b 001111100;
assign hSpriteR[8] = 9'b 000111000;
assign hSpriteR[9] = 9'b 000010000;
```

Some sprites are only drawn in white, so we only need one array which is sent to all bits R,G,B of the VGA.

```
wire[31:0] gameOver[14:0];
assign gameOver[0] = 32'b 00111111000111000110001101111111;
assign gameOver[1] = 32'b 01100000001101100111011101100000;
assign gameOver[2] = 32'b 11000000011000110111111101100000;
assign gameOver[3] = 32'b 11000111011000110110101101111100;
assign gameOver[4] = 32'b 11000011011111110110001101100000;
assign gameOver[5] = 32'b 01100011011000110110001101100000;
assign gameOver[6] = 32'b 00111111011000110110001101111111;
assign gameOver[7] = 32'b 00000000000000000000000000000000;
assign gameOver[8] = 32'b 01111110011000110111111101111110;
assign gameOver[9] = 32'b 11000011011000110110000001100011;
assign gameOver[10]= 32'b 11000011011000110110000001100011;
assign gameOver[11]= 32'b 11000011011101110111110001100111;
assign gameOver[12]= 32'b 11000011001111100110000001111100;
assign gameOver[13]= 32'b 11000011000111000110000001101110;
assign gameOver[14]= 32'b 01111110000010000111111101100111;
```

Implementation: Gameplay

Score Calculation:
The player receives a different score when they jump up for each level.

To prevent the player from jumping up and then back down and then up again to get more score, we use a register to represent whether the player has made a specific jump. That way, they only get each score once per level.
This register is cleared when the player reaches the top or loses a life.
Platform 29 is the first on level 2, and platform 30 is the first on level 3.

```
if (positionY <= platY[29] - 16 && ~hasJumped[0]) begin
      totalScore <= totalScore + 125;
      hasJumped[0] <= 1;
end
if (positionY <= platY[30] - 16 && ~hasJumped[1]) begin
      totalScore <= totalScore + 250;
      hasJumped[1] <= 1;
end
```

To make the score count up, we check totalScore in a separate always block and increment ourScore (the one displayed) if necessary.

Then, we increment digit 1 (the 1's place in score) and ripple that increment up the digits if necessary. totalScore, ourScore, and all digits are reset at transition between the TitleScreen and GamePlay states.

```
if (totalScore > ourScore) begin
      digit1 <= digit1 + 1;
      ourScore <= ourScore + 1;
end
if (digit1 == 9) begin
      digit2 <= digit2 + 1;
      digit1 <= 0;
      if (digit2 == 9) begin
            digit3 <= digit3 + 1;
            digit2 <= 0;
            if (digit3 == 9) begin
                  digit4 <= digit4 + 1;
                  digit3 <= 0;
                  if (digit4 == 9) begin
                        digit5 <= digit5 + 1;
                        digit4 <= 0;
                        if (digit5 == 9) begin
                              digit5 <= digit5;
                        end
                  end
            end
      end
end
```

Player Movement:
The player can move left and right, and can also jump.
Left and right movement is simple, wait for the user to press a button and then just check that we are not going to go off the screen.
We can't check (positionX < 0) so instead we check (positionX > [some big number]) to see if player is off the screen to the left, since wrapping will make the position large.

```
if(btnR && ~btnL && (positionX < 309)) begin
      positionX <= positionX + 3;
      lookLeft <= 0;
end
else if(btnL && ~btnR && (positionX > 0)) begin
      positionX <= positionX - 3;
      lookLeft <= 1;
end
if (positionX > 309)
      positionX <= 309;
if (positionX > 350)
      positionX <= 0;
```

Jumping is more complex. Specifically, landing is difficult.

The issue is that we have to check which platform we have/will land on.
For now, we will assume that the register collide is '1' when we are on a platform, and '0' when we are not. We will also assume that the wide register platPos represents the y-position of the last platform we are stood on. We will set these registers in the collision section later on.

Once we have these registers, it becomes fairly simple to figure out how to move the player in the y direction.
If we are on a platform, make the y-position of the player the position of the platform + the height of the player (since (0,0) is the top left corner of the sprites).
If we are not on the platform, then decrement vertical velocity, and subtract vertical velocity from y-position of the player. This gives a nice parabolic jump.
If the player wants to jump and is touching the floor, then we set vertical velocity to a positive number. We use the convention that 7 is 0 vertical speed, 0 is -7 vertical speed, and 14 is +7 vertical speed since negative numbers are tricky.

```
if (velocityY > 0 && ~collide)
      velocityY <= velocityY - 1;
if (collide) begin
      velocityY <= 7;
      positionY <= platPos - 16;
end
if (btnU && collide) begin
      velocityY <= 14;
end
if (~collide)
      positionY <= positionY - velocityY + 7;
```

Barrel Movement:
To move the barrels, we check the barrelDir register to see which direction to move the barrel, and then increment barrelX (x-position) by barrelS (speed). We also check barrelCol to make sure that the barrel is on a platform before moving it, which forces the barrel to fall straight down when it is falling. When the barrel turns from moving left to right we also check to see if it is on the first platform. If it is, we send it back to the top.
We use the convention that 5 is 0 speed, 0 is -5, and 10 is +5 to avoid negative numbers.

```
if (barrelCol) begin
      barrelV <= 5;
      barrelY <= barrelPlat - 16;
end

if (~barrelCol) begin
      barrelY <= barrelY - barrelV + 5;
      barrelX <= barrelX;
end

if (barrelDir) begin
      if (barrelV > 3)
            barrelX <= barrelX + barrelS;
      if (barrelX > 300) begin
            barrelDir <= 0;
            barrelX <= 300;
      end
end
else begin
if (barrelV > 3)
      barrelX <= barrelX - barrelS;
      if (barrelX < 10 || barrelY >= platY[0] - 16) begin
            if (barrelY >= platY[0] - 16) begin
                  if (barrelX > 320) begin
                        // Put barrel back to top
                        barrelDir <= 0;
                        barrelV <= 5;
                        barrelX <= platX[NUM_PLATS - 2];
                        barrelY <= -20;
                  end
            end
            else begin
                  barrelDir <= 1;
                  barrelX <= 10;
            end
      end
end

if (barrelV > 0 && ~barrelCol)
      barrelV <= barrelV - 1;
```

Barrel Movement:
To check if the barrel has hit the player, we must ensure the following are all true:
1. Player's right edge is to the right of barrel's left.
2. Player's left edge is to the left of barrel's right.
3. Player's bottom edge is below the barrel's top.
4. Player's top edge is above the barrel's bottom.

If the player and barrel have collided, we check if the player has any lives. If they do, we remove one and reset the barrel and player to their starting positions. If they do not, we go to the game over screen.
Since the sprites are indexed from the top left, we must add the width and height of each sprite to some of the position values before checking collision.

```
if (positionX + 10 >= barrelX && positionX < barrelX + 16 && positionY + 14
       > barrelY && positionY < barrelY + 16 && justDied == 0) begin
      if (health > 0) begin
            health <= health - 1;
            // Set appropriate registers for reset
      end
      else begin
            gameState <= 2;
      end
end
```

Platform Collision:
To prevent the player from falling through the floor, we can check to see if there is a collision between each platform and the player.
This is done by first checking that the player is in line with the platform, and then that they are at least half way through it (since the player can jump up through platforms). We also check that the player is falling before deciding if there was a collision, since you cannot collide with something you are moving away from.
The index j represents the specific platform that we are checking.

```
if (positionX + 12 >= platX[j] && positionX <= platX[j] + 20) begin
      if (positionY + 8 <= platY[j] && positionY + 16 >= platY[j]) begin
            if (velocityY <= 7) begin
                  collide = 1;
                  if (platY[j] < platPos)
                        platPos = platY[j];
            end
      end
end
```

platPos is the index of the highest platform that the player is on, so that when they go up or down steps they will appear to be stood on the higher step, not in it.

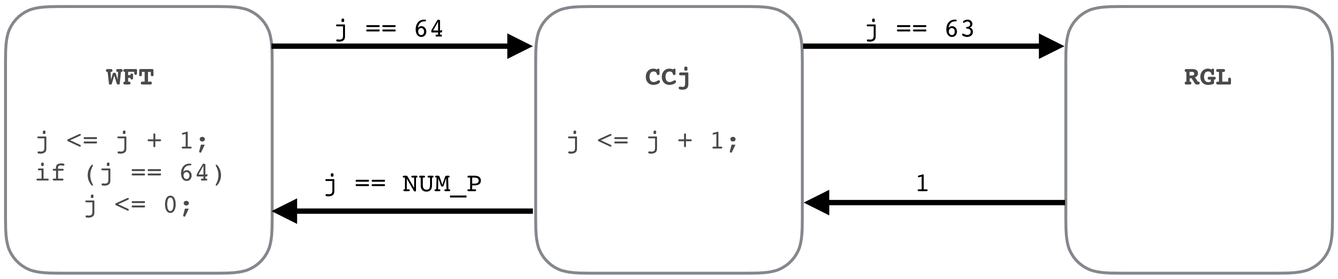
This algorithm takes O(n) time given n platforms to check. If we were to store the platform data as a binary array with 0 representing no platform, and 1 representing some platform, then we could index this array by the player's position to figure out if we have collided or not in constant time.
This approach would allow for many more platforms to be used more easily, but also requires more storage.
Since we are not using too many platforms, O(n) is fine.

Platform Data:
The (x,y) position of each platform is stored in a wire array.
The number of platforms is stored in a parameter.

```
parameter NUM_PLATS = 46;
wire[9:0] platX[NUM_PLATS-1:0];
wire[9:0] platY[NUM_PLATS-1:0];
assign platX[0] = 0;
assign platY[0] = 230;
assign platX[1] = 20;
assign platY[1] = 230;
. . .
assign platX[45] = 240;
assign platY[45] = 68;
```

This data is then read by the collision loop and sprite scanning sections.

Collision Loop:
To check all the platforms, we have to loop through them in some fashion.
If we were to use a for loop, then the synthesiser would copy/paste the circuitry needed for checking each platform, and we would end up with a rather frustrating error in the implementation phase telling us that the design is too big for the board.
Instead, we must implement a state machine, with three states.
In CCj we check collision between the player and platform index j as well as the collision between the barrel and platform index j.
In RGL we run the game logic described earlier.
In WFT we wait until j reaches 64 such that the RGL state is reached every 64th clock.

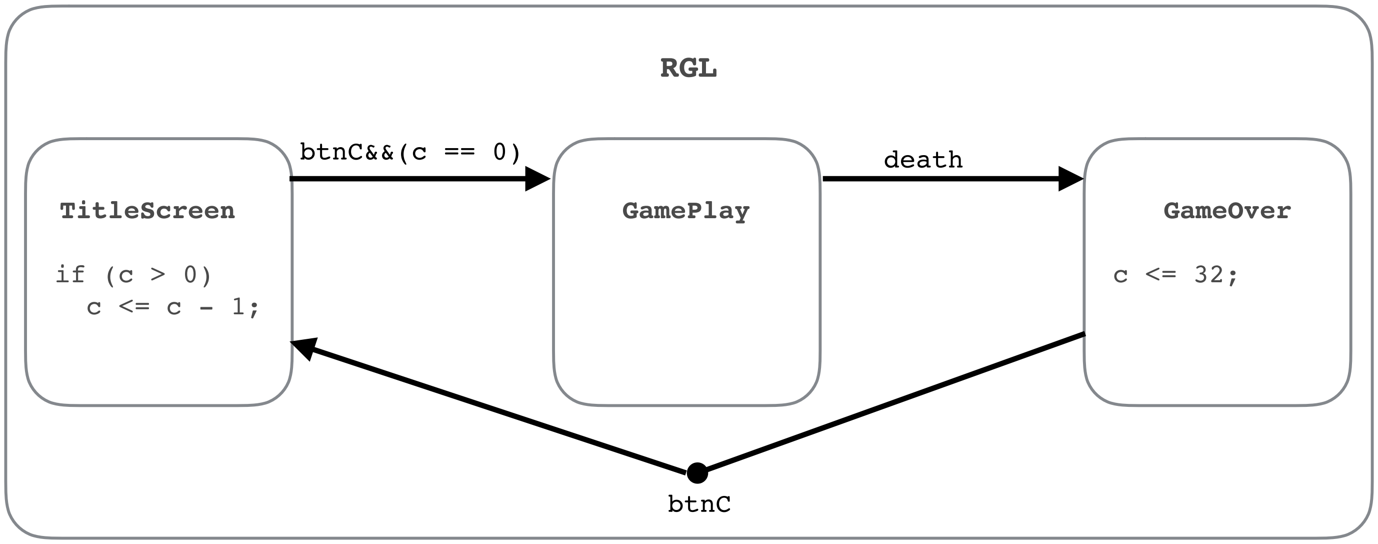


We want the game logic to run at a clock rate of DIV_CLK[20]. This rate gives a good balance between player movement speed, and smoothness.
Since the RGL state occurs once every 64 clocks, we must run the state machine at a clock rate of DIV_CLK[20-6] = DIV_CLK[14].
With this setup, we can have up to 63 platforms on the screen. Currently, we use only 46.
If we needed more than 64, we can change the 64's in the state machine to 128's, and change the DIV_CLK[14] to DIV_CLK[13], and we will be able to use 127 platforms. We can repeat this process 13 more times, if we wanted to.

Game State:
We use the register[1:0] gameState to store the game state: TitleScreen, Play, GameOver.
When we are in the RGL state, we check to make sure gameState == Play before running the play logic.

If gameState is TitleScreen or GameOver, we just look for a button press to decide whether to move to the next state. We debounce the TitleScreen button press by waiting for a counter to count up to a value before allowing the state to change.

gameState is also checked in the sprite scanning section, to decide whether or not to draw all the platforms, player, etc. or only the Title or GameOver sprite.



Future Improvements:

Multiple Barrels:
If we wanted to use multiple barrels, then we could move the player/barrel collision logic out of the RGL state. We could then add another state to the collision loop to loop through each barrel and check collisions between that barrel and the player, we could call this CCk.

There would be a link coming out of CCj going to CCk where we would increment k to check if each barrel has hit the player. Once k reaches the number of barrels, it would reset and we would return to CCj and continue as before.

For every time we double the number of barrels, we would need to decrement the X in DIV_CLK[X] by one.

Multiple Maps:
To include multiple maps in the game, we can store an array of platforms for each map, and then display the appropriate platforms for the map.

If we ensure each map has less than 63 platforms, we can use the same collision logic, and index the platform data by map as well as by platform number to run the correct map.

When the player reaches the end of the map we increment the map index and the next map will be shown.

High Score:
To track the high score we can include another score register, and write to it when totalScore is greater than highScore.
Then we can create more digit registers and scanning logic to draw the highScore to the screen.

Unfortunately, the score will be lost when the board is shut down. But players will be able to see who got the best in any particular session.