Frogger

David Tucker

CMPE 100/L — Lab 7
Section 2 (T/Th 2:00pm)
June 7, 2012

Descriptions

- The purpose of this lab was to use our acquired knowledge of logic design in a large application and to familiarize ourselves with driving a VGA display.
- The goal was to make a simplified version of the classic arcade game Frogger.
- This project consisted of 5 major[1] components:

[1] For the purposes of this report, it is assumed the reader knows such basic & common parts as the clock, GSR, counter, adder, multiplexer, and shift register.

Results

Part 1: State Machine

In essence, the state machine *is* the game. It defines (in Verilog) when the game should change its state based on the position of objects on the screen. There are 3 legal states: playing, won, and dead. The latter two are considered “game over” states, and the only way to transition back to “playing” from either is through GSR.[2] A “game over” state is indicated by an inability to move the frog.

To win, a player must guide Tom (the frog) from the bottom border[3] to the top border (across the highway) without being hit by a car. As soon the frog enters the top border, the game enters the “won” state, and the frog begins to blink to signify this to the user. Blinking is accomplished in a strobe controller that works using a 20-bit counter and a T flip-flop. The flop’s output is toggled once every 2^{20} cycles. This signal is later used to determine when the color of the frog should change (see Part 6). If the frog hits or is hit by a car, or if it is moved out of bounds, it dies (i.e. the games moves to the “dead” state). This is indicated by a red frog.

In order for the state machine to function according to this specification, it must know if the frog’s position overlaps with another object. In this implementation, this is achieved using comparisons between the pixel being updated and the position of each component. These comparisons occur in each component’s respective controller. If the two overlap, a logic 1 is asserted on a signal that the state machine accepts. In all, this unit intakes 4 of these signals: frog, car, finish (top border), and boundary (left or right border). If “frog” and any other signal is high, the game changes its state. An important realization when defining transitions is that since any input may be asserted in any state, all states must be able to handle all possible input combinations.

The machine indicates and remembers its current state using D flip-flops and one-hot encoding. Therefore, there must be 1 flop for each state (3 total), the output of which is an output for the state machine indicating whether the game is in that state. Only a single flop should be high at a time. When the game begins (after GSR is asserted), none of these flops indicate a current state. Surrounding the “playing” state flop with inverters solves this problem by restoring a stored (inverted) value before it is used elsewhere. Any other value forced into the flop is only inverted once (at output) creating a logic 1 when GSR is enabled.

A state transition diagram and the characteristic equations are included in the attached notes and Verilog code, respectively.

[2] In this implementation, GSR is used as an on/off switch. This is because all 4 push buttons are used in gameplay and the FPGA’s power switch causes the loaded game to be cleared which means it must be reloaded before playing again after the board is turned back on.

[3] There is a 32-pixel border that outlines the highway (see Part 4). It is implemented in a way very similar to the synchronization windows described in the VGA driver (see Part 2).

Part 2: VGA Driver

The game is played on a 640x480 pixel VGA LCD monitor; however, the driver controls a region that measures 800x525. The visible area is called the active region and is the only area where color outputs may be enabled. The invisible region is used for synchronization between the FPGA and the display.

The driver revolves around 2 counters. One tracks the horizontal component (X) of the pixel being updated and the other tracks the vertical component (Y). The X component is incremented every cycle to ensure the screen is refreshed as often as possible. Each time the X component reaches the end of the display (800) it is reset and the Y component is incremented. When the Y reaches the last line AND the X reaches the end of that line, both counters are reset and a new frame begins.

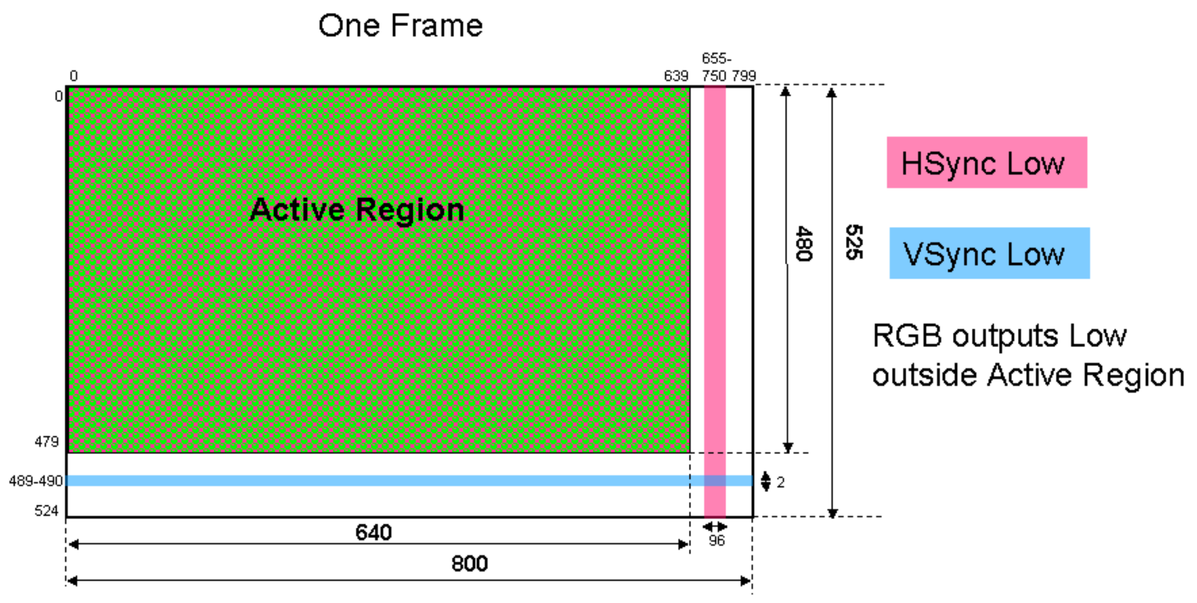
Synchronization occurs when $654 < X < 751$ (96 cycles) or when $488 < Y < 491$ (2 cycles). It is detected by comparators (implemented in Verilog) that indicate the relation between two 10-bit values. During synchronization the sync signal of the appropriate component goes low (via a NAND of the above conditions). This scheme ensures the developer is working with the correct "current" pixel (i.e. the pixel being updated).

800 is the highest value either counter must count. In binary, this requires 10 bits. Since $2^{10}-1$ (the highest value a 10-bit counter can reach = 1023) > 800, another comparator must be employed to recognize when either counter should be reset (equality is neither less than NOR greater than). Since X and Y value comparisons like these are made elsewhere in the design (including in active region detection), convenience dictates a (mostly) 10-bit design on top of this 10-bit coordinate system.

Relevant Pin Mapping:

HS → P39

VS → P35



Part 3: Frog

Like the VGA driver, the frog's position is tracked using two 10-bit counters. In particular, they track the coordinate of the upper left corner of the frog, (frogX, frogY). Using this and the frog's width (x00A) and height (x00A), the coordinates of the other 3 corners (boundaries) can be calculated. These positions are compared with the coordinate of the current pixel to tell if the frog is currently being updated. If the current pixel is to the right of (greater than) the left side of the frog (frogX), left of (less than) the right side (frogX + x00A), below (greater than) the top (frogY), and above (less than) the bottom (frogY + x00A), it is a within the frog. The truth value of this comparison is used to detect the end of the game (see Part 1) and for coloring (see Part 6).

This controller is also responsible for moving the frog when 1 of the 4 push buttons is pressed. Since the frog needs to move more than 1 pixel at a time (16 in this implementation), the counters are custom-built to accommodate this in a single cycle. Also, the frog may move either direction on either axis, so they must be decrement-able. They work by separating the two basic functions of a counter: memory and addition. Naturally, the memory is handled by an array of D flip-flops. This is where the initial position of the frog, (290,450), is encoded in the same way as the initial game state: by sandwiching the appropriate flops with inverters. Instead of a count enable which gets incorporated as logic directly on the flops, a 10-bit ripple carry adder (RCA) adds an offset to the frog's position and a multiplexer chooses between the new position and the unchanged one. This essentially makes the select line of the multiplexer the count enable. The offset is a constant (x010) that represents the frog's positive jump distance.[4] If the frog is moved in a negative direction (up or left), the offset is complimented before being added.

Ensuring the frog never leaves the screen is another feature that is essential to gameplay. This is especially true on FPGAs with worn, damaged, or highly sensitive push buttons because these problems can cause multiple presses to be detected when only 1 is indicated—even with an edge detector! Locking in the frog is achieved by comparing the current position to the highest (x002), lowest (x1C2), leftmost (x002), and rightmost (x262) values the frog may be at. If the frog's position is equal to any of these positions and a move that would push the frog past the boundary of the screen is detected, count enable is disabled.

Relevant Pin Mapping:

BTN0 → P69
BTN1 → P47
BTN2 → P48
BTN3 → P41

[4] Constants throughout the design are represented in hexadecimal. Since this is a 10-bit design and a hexadecimal digit represents a 4-bit value, all constants must specify at least 3 digits.

Part 4: Cars

The 6 cars on the highway all operate using identical, customizable controllers. They are tracked in a way similar to the frog (i.e. counters maintain the position of the top left corner of the car); however, for smooth motion, incrementing these counters 1 pixel at a time is desirable. This is not the only way in which car tracking is simpler than frog tracking. Since cars only move laterally across the screen, the vertical component of their position is constant (x024, x064, x0A4, x102, x142 & x182) and based on the car height (x03C) and padding. So, only a single standard 10-bit counter is required to manage each car's horizontal location. Note that like the counter in the VGA driver, the maximum car position is less than the highest value of a 10-bit counter (1023). When and to what value the counter needs to be reset depends on the direction of the car. The lower 3 cars move left to right (LTR) and must begin again at 0 (x000) when they hit 800 (x320). On the other hand, the upper 3 cars move right to left (RTL) which means their counters decrement from their initial position, and they must wrap around to 799 (x31F) when they hit 1023 (x3FF)[5]. The reload value is determined using two 2 to 1 multiplexers which evaluate the direction of the car and whether the current car is wrapping or a new car is being generated. Regardless of the direction, a car begins at 720 which is exactly halfway through the inactive region (see Part 2). This is so the distance from the start to the active region is the same for cars in either direction and a single constant (x2D0) may be loaded whenever a new car is generated. It also means cars in both directions end at the same position. The only differences between the two are when they wrap and whether the counter increments or decrements.

All lanes operate cars with variable presence, speed, and size. This is accomplished through the use of a pseudo-random timer. At the top level of this design[6] are two 16-bit linear feedback shift registers (LFSRs) that combine to create a varying 32-bit value every cycle. From this, 16 different bits are bussed into a timer[7] in each car creating a different "random" pattern for the 6 instances. The timer emits 2 signals that indicate the mode of the lane: driving or generating.[8] The timer consists of a loadable counter that only counts when it is not at TC (a.k.a. generating). When a new car is generated, a new value is loaded from the LFSRs and the timer is no longer at TC. While it counts, so do width and speed counters that cycle from 62 (x03E) to 248 (x0F8) pixels and 32512 (x7F00) to 65535 (xFFFF), respectively. Notice that the speed must be 16 bits to represent an acceptable speed range. This is admissible for the same reasons as the 16-bit timer; it is an independent subsystem.

[5] Since 0 is a valid horizontal (and vertical) position and a counter enters TC when decremented from 0, TC is the first invalid position for a car and indicates a need for reset.

[6] It is necessary for this to be above any car logic since the same design is used for each car and different random values are desired for different lanes.

[7] 16 bits are used to widen the range of possible random generation periods. It is not necessary for this to be 10 bits since it is not a position on the coordinate system. In fact, the actual time from the timer is not used at all!

[8] These signals are inverses of one another.

When the timer hits TC, the car begins "driving," and both the variable speed and width counters hold their value. At this point a fifth counter begins incrementing. Each time it reaches the established speed, the car advances 1 pixel by asserting CE on the position counter, and the count is reset. This repeats until the car wraps back around to its starting point at which time a new car size and width begins generating for a new pseudo-random delay.

The final component of each car's design is a set of comparisons that are completely analogous to those made to determine if the current pixel is a within the frog (see Part 3). The origin of the values used in the comparisons is the only difference between the two. The upper bound of the car is established by its constant vertical position. The lower bound is simply that value added with the height of a car. The left bound is tracked by the car's position counter, and the right bound is that value plus the dynamic width calculated during generation. If the pixel being updated is within these bounds, it is updating the car. If it is updating the car and frog at the same time, the game ends and Tom the frog "dies."

Part 5: Layering and Coloring

As discussed earlier, the visible portion of the screen is not the total area the driver controls. No color outputs should be active unless the active region is being updated. The display takes multiple inputs to allow for shading and a wider range of hues. For simplicity, all red, green, and blue inputs are tied together leaving 1 shade of each color to work with. Each visual component (border, frog, & cars) outputs a signal indicating whether the pixel being updated on the VGA display is within the bounds of the object it represents. These bits are used to determine which color to make the current pixel. This creates an interesting effect when more than one is on at a time.

To understand how color mixing works on the VGA display, it is helpful to understand the difference between light and pigment color. Specifically, the way they interact to create other colors is inverted. Pigments mix to make the color black while combined light creates white. This design drives the latter. To ensure that objects do not cause color-mixing when they overlap, logic must be added to implement layering. This problem particularly affects the border element. Having a white border helps combat this since white is a mixture of all the colors; however, it is inadequate as a permanent solution because the frog requires red and blue to be off to display properly on top of the border. Object layering is achieved through converging the signal of the object to show above AND NOT of the signal to show below. For this application, Tom the frog shows above the border and below the cars which appear below the border. When the player wins, the frog blinks. This works by asserting the border color (white) every other time the strobe toggles (see Part 1). Otherwise, the default green is revealed. All scenarios when a given color should be enabled are added to an OR gate that drives the output.

Relevant Pin Mapping:

RED0 → P70	GRN0 → P52	BLUE0 → P44
RED1 → P68	GRN1 → P51	BLUE1 → P43
RED2 → P67	GRN2 → P50	

Conclusion

The included schematics form a working game as described above. I began the implementation by building and testing the VGA driver in isolation. In many ways, it is the most important component because without it, none of the design will be seen (working or not). From there I added the border which was a very simple way to begin locating objects on screen. Since the frog and cars are found the same way, getting them to show up was simple. Making them move was more difficult. I first got the frog to move 1 pixel at a time, then I built the special counter that increments 16 pixels at a time. I decided on a smaller frog to match the narrower cars since there were to be 6. Next I locked the frog into the screen because it was easy to see how it could be a problem if I didn't. The frog hops very fast; so fast, sometimes, that he appears to jump to different areas. Having a smaller frog also seems to help with this problem. I then built the state machine which was rather rudimentary. It is neither a Moore nor a Mealy machine since it does not drive outputs directly. It only manages states. All along the way I colored and layered each object as necessary. The cars took the longest by far. One problem that delayed development significantly was getting the cars running in the first few cycles. The solution I discovered was a GSR detector (see attached schematic). When GSR is detected, an internal signal (newCar) is asserted and everything begins running as if a car before it had just finished. Also, I added a second safe zone in the middle of the road (the meridian) to adjust to the downsized frog and cars and to give the player some help since the game turned out to be a challenge to beat! I then began to construct separate LFSRs for each car, but when I realized how inefficient that approach would be, I decided on one larger LFSR that the cars would share instead. To decide which taps a given car should take, I had random sets of [0-31] generated on http://www.random.org. Modularity was *extremely* helpful during this lab as many units were reused repeatedly (e.g. m20_10, add10, cb10cled, comp10, etc.) Another useful tool I used was the flash base converter at http://www.mathsisfun.com/flash.php?path=/images/convert-binary-hex.swf.

Experiments & Questions

① *N/A*