FROGGER

Nathan Vantzelfde
Cory Zue

6.111 Introductory Digital Systems Laboratory
Final Project
May 13, 2004

Abstract

Video games, introduced with the advent of computer technology have become extremely popular in today's society. The old video game Frogger is designed and implemented on a Field Progammable Gate Array (FPGA). The goal of the game is to navigate an animated frog across a highway avoiding a series of cars, and then a river, jumping on logs and bobbing turtles. The interface to the game is an Atari joystick, and the game can be displayed on any VGA compatible monitor. The game consists primarily of two modules. The game module controls the logic of the game including the location of objects, movement of the frog, and end of game conditions. The video module displays the game data on the monitor, using images stored on a read only memory (ROM). Both modules use a major/minor FSM structure. Design and testing strategies are discussed.

1 Introduction

We designed and implemented the popular arcade game Frogger. The game consists of an animated frog that first crosses a highway and then a river. The goal of the game is to make it safely from the bottom of the screen to the top.

The frog must cross the highway without colliding with a series of cars and trucks that move across the screen, alternating directions and varying speeds with each successive line of traffic. Each lane of the highway is a set of identical cars that move with the same speed. Between the highway and the river is one row of grass with no obstacles. Logs and lily pads float across the river, and the frog can only cross on these objects. If the frog reaches the top of the screen, the level is completed and the speed increases. Also, The number of cars on the freeway increases with each level, just as the number of logs and lily pads on the river decreases. If the frog collides with a car or falls in the water the game starts over with the frog at the bottom of the screen. The version of the game that we attempted to mimic is shown in Figure 1.

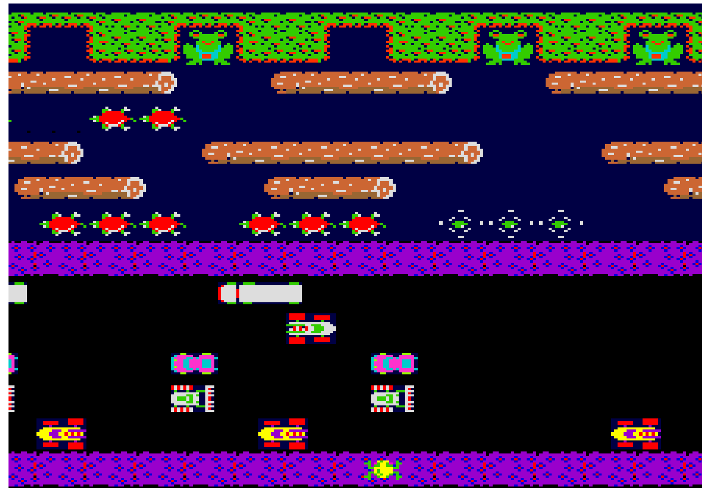

Figure 1: The Frogger Arcade Game

The input to the game is a standard Atari Joystick. This controller consists of four directional buttons (up, down, left, and right), and a fire button. The directional buttons control the movement of the frog, with each press corresponding to a movement of 1 unit (one lane of the highway). During gameplay, the fire button pauses the game, causing the screen to freeze. Releasing the fire button resumes play.

The output of the game is a video signal for interfacing with the color TV monitors in the 6.111 lab. The video outputs are three color bits—red, green, and blue—and four synchronizing and blanking bits—horizontal and vertical synch and blank.

2 Gameplay Unit

The gameplay unit is responsible for controlling the flow of the game. It processes user input from the joystick, moves the frog accordingly, and places and moves various other objects on the screen such as logs and cars. It tests for collisions between the frog and the other objects and acts accordingly, depending on whether the frog is on the road or over the river. It also tests whether

the frog has safely reached the opposite side of the river, and if so, increases the level and the corresponding speeds of the cars and logs.

A block diagram for the gameplay unit is shown in figure 2. A number of smaller modules have been omitted for clarity. The gameplay unit only has two inputs, a global reset signal and a five-bit input from the joystick. There are two large finite state machines, one to control the frog movement and one to control the flow of the game. All of the objects, including the frog, are stored in the ram, which has two read ports, one of which is used by the video unit to access the type and location of all the objects. The gameplay unit also outputs a signal `num_objs` to the video unit, which signals how many of the locations in the object ram contain valid objects to be displayed. Each of the major modules of the gameplay unit will be discussed in subsequent sections.

2.1 Synchronizer and Divider

The synchronizer synchronizes asynchronous inputs from the joystick and reset signal to the system clock by delaying each bit through two or more registers. The divider produces a 15 Hz enable signal from the 27 MHz system clock. This enable signal is used by the frog FSM, the gameplay FSM and the several of the other modules to signify to start a new game cycle, since each object is updated only once every 1/15 second. The divider also takes the `pause` input from the joystick interpreter; when `pause` is high, the divider stops produces the enable signal, effectively stopping the every element of the game in its current state.

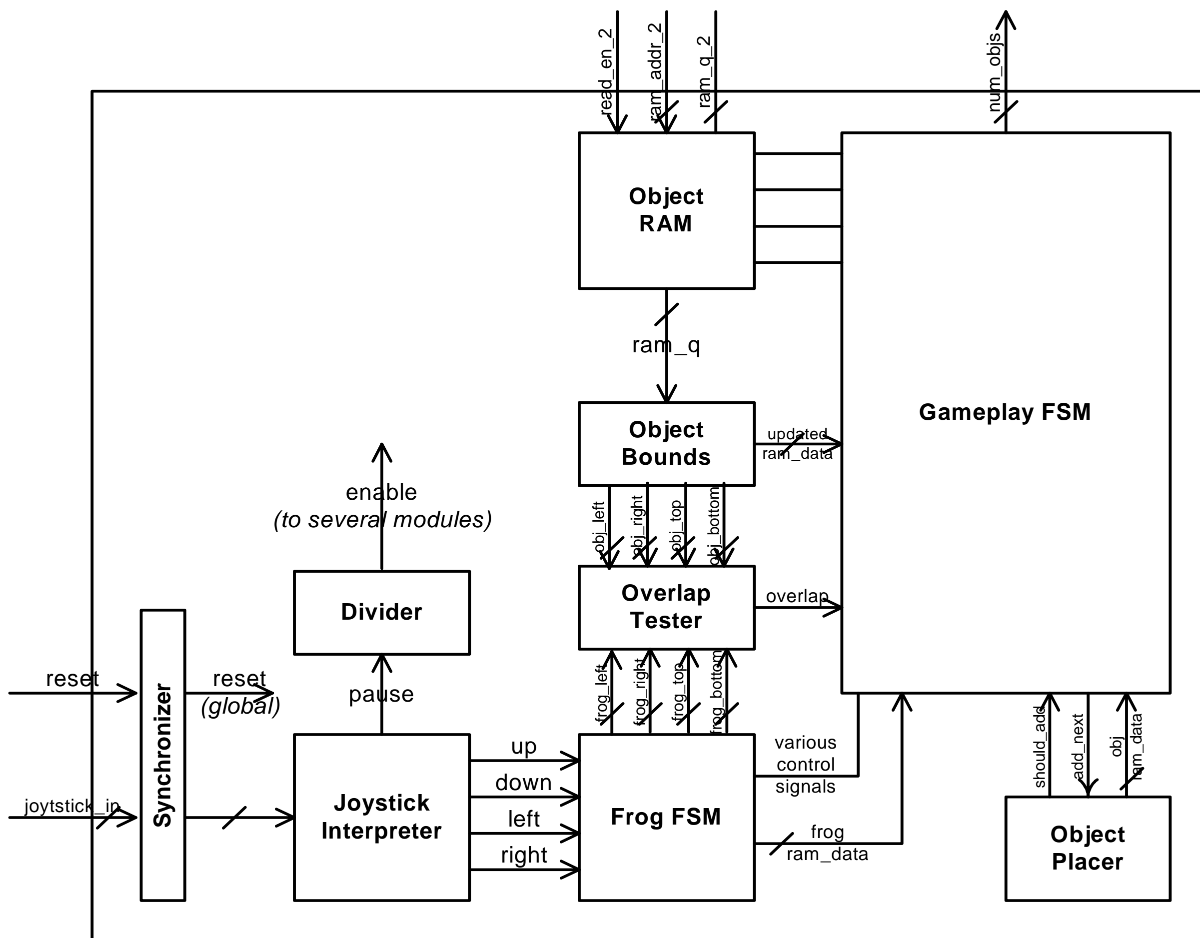

Figure 2: Block Diagram for the Gameplay Unit

2.2 Joystick Interpreter

The joystick interpreter is responsible for taking in the five-bit synchronized input from the Atari joystick and produce five usable signals, `pause`, `up`, `down`, `left`, and `right`, the first of which is used by the divider to pause the game and the latter four of which are used by the frog FSM to control the frog's movement. The Atari joystick has five output pins, corresponding to each of the four cardinal directions and one for the button. These pins are forced low when the corresponding direction or button is pressed. Before being used by the gameplay unit, each of these signals are passed through two inverters on the lab kit to ensure that each has a valid voltage level before being used by the FPGA.

The joystick interpreter, therefore, needs only to invert each of the five signals to generate the desired signals. While the joystick module is not very complex, it was included in the design nevertheless so that the function of processing the user input is decoupled from other modules. That is, if we chose to switch to a difference joystick, only the joystick interpreter would need to be change; all other modules will not by affected at all.

2.3 Frog FSM

The frog FSM maintains the frog's position and state, in terms of whether the frog is jumping, dying, or standing idle. It uses the four direction inputs from the joystick interpreter to move the frog accordingly. It also takes in three inputs from the gameplay FSM: `collided`, `restart`, and `velocity`. The first of these is a single bit which signifies whether the frog has collided with one or more of the other moving objects. If the frog is on the road, a high value for `collided` signifies that the frog has hit and car and should die. Likewise, if the frog is jumping over the river, a low value for `collided` means that the frog has not landed on a log or turtle, and has fallen in the water. The `restart` signal is used to reset the frog to its initial position on the bottom of the screen, for example, after completing a level. The third input, `velocity`, is the velocity of the object which has collided with the frog. If the frog is standing on a log or turtle, therefore, the frog FSM uses this velocity to update the frog's position, since the frog is riding the given log or turtle.

The frog FSM outputs three signals to the gameplay FSM: `died`, `finished`, and `frog_ram_data`. The first of these signifies that the frog has died, and the gameplay should update the remaining number of lives accordingly. The signal `finished` signifies that frog has safely reached the top of the screen, and the game should continue with the next level. Lastly, the `frog_ram_data` is the data for this current frog state that should be stored in the ram to be used by the video unit.

The frog FSM also outputs four signals to the overlap tester module, which correspond to the four corners of the frog. These four values are used by the overlap tester, along with the corresponding four values from an arbitrary object, to determine if the frog and the object collide with one another.

A simplified version of the state transition diagram for the frog FSM is shown in figure 3. The frog FSM only transitions state when the enable signal is high, so that the frog's position and state and only updated once every 1/15 of a second. Signals such as `should_jump_up` are determined not only by the input from the joystick, but also from the frog's current position, so the frog is not allowed to jump beyond the edges of the screen. While the state transition diagram does not shown this, the frog remains in each of the jump states for several cycles, so that a single jump takes around a quarter of a second. Likewise, there are in reality several dying states, each corresponding to a new picture being drawn on the screen, so the animation is smooth. The signal `died` is only asserted on the last cycle of the frog dying.

2.4 Overlap Tester

The overlap tester is a small piece of combinational logic that takes in four inputs, `frog_left`, `frog_right`, `frog_top`, and `frog_bottom`, from the frog FSM, corresponding to the edges of the frog, and four inputs—`obj_left`, `obj_right`, `obj_top`, and `obj_bottom`—from the object bounds module, corresponding similarly to the edges of the object currently being updated in the ram. It outputs one bit to the gamplay FSM, `overlap`, which is assert if and only if the boundaries of the frog overlap with the boundaries of the current object.

2.5 Object Bounds

The object bounds module takes in the 32-bit data current being read from the object RAM. Based on the object type and location of this data, it chooses appropriate object bounds (left, right, up and down) and passes the four signals to the overlap tester module. It also outputs a 32-bit signal, `updater_ram_data`, which is the ram data modified to correspond to the objects velocity, as well as its type—in the case of turtles, for example, the object type changes gradually so that turtles appear to bob up and down in the water. As long as the updated object is still within the screen bounds, the gameplay FSM will use this 32-bit data to update the RAM on the following clock cycle.

2.6 Object Placer

The object placer module is responsible for placing all of the objects on the screen initially, for example, after a reset or at the beginning of a new level. The object placer also places new objects on the screen during levels so that there are always cars and logs coming from the left and right sides of the screen.

The object placer signifies to the gameplay FSM that it is ready to place a new object by asserting `should_add`. When the gameplay FSM asserts `add_next`, the RAM data for the new object are passed back to the gameplay FSM as `placer_ram_data`.

Internally, the object placer maintains dozens of small modules, each of which is responsible for determining when to place one type of new object, perhaps the left block of a log on a given row. These small modules maintain two registers, most_recent_x and spacing, which

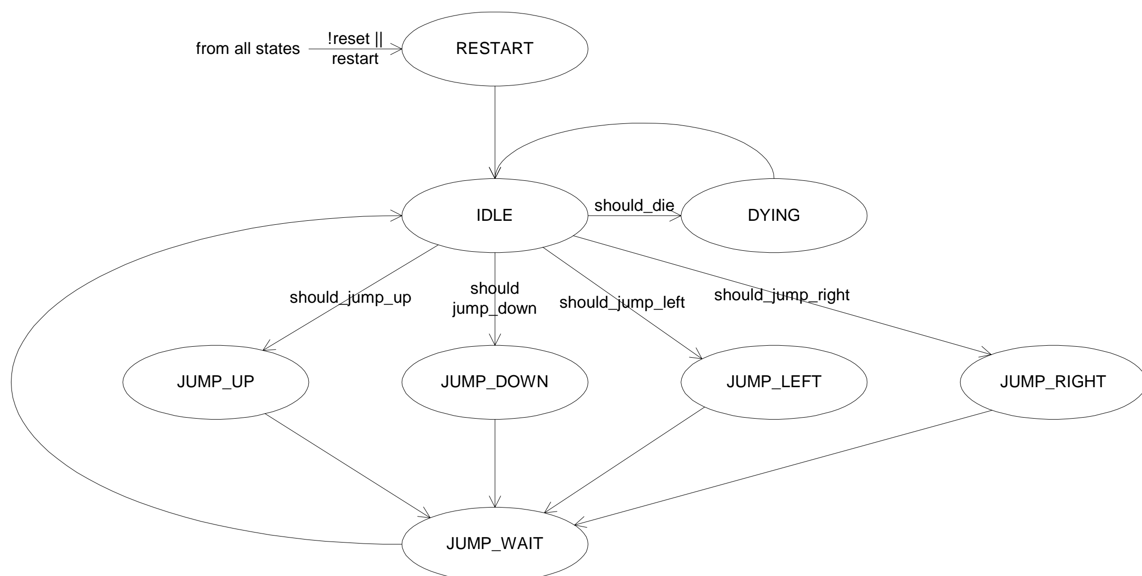

Figure 3: State transition diagram for the frog FSM

hold, respectively, the most recent x position of the last object placed by the module, and the desired horizontal spacing between objects placed by the module. On enable, these modules update `most_recent_x` based on the object velocity, and determine whether the next object to be added, given the most recent position and the spacing, would be on the screen. If so, it signals that it is ready to add an object. When signaled to do so, it latches the RAM data for the new object.

The large object placer module is responsible for maintaining each of these smaller modules, and routing signals accordingly so the larger module signals `should_add` when any of the small modules signal, and when `add_next` is asserted, the object placer module makes sure the correct smaller module is signaled.

2.7 Object RAM

The object RAM maintains, for each object on the screen, the object type, its x location, its y location, and its velocity, which are stored in 6, 10, 10, and 6 bits, respectively. The object RAM has one write port, used by the gameplay FSM to update and add new objects, and two read ports. One of these read ports is used by the gameplay FSM; the other is used by the video unit to read the objects stored by the gameplay unit.

For the read ports, new data is latched to the output when read enable is high. Similarly, for the write port, new data is latch into the RAM when write enable is high.

2.8 Gameplay FSM

While the gameplay FSM controls the flow of the game, most of the bulk of the work of updating and moving objects is done by other modules. The gameplay FSM is, however, chiefly

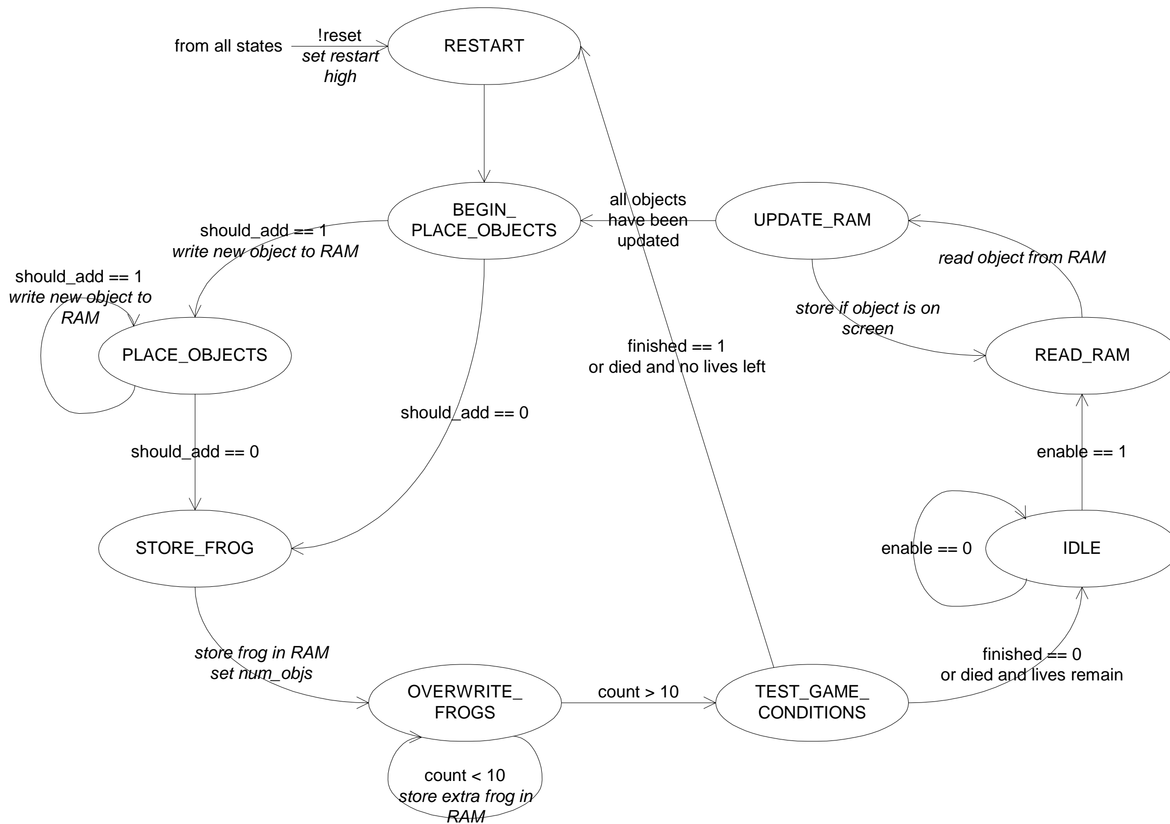

Figure 4: State transition diagram for the gample FSM

responsible for writing and reading to and from RAM, and supplying signals that control the other modules.

The state transition diagram for the gameplay FSM is shown in figure 4. The basic flow of the FSM is as follows. Starting in the IDLE state, the FSM waits for the enable signal before begin to the first stage in the game cycle, updating the objects stored in the RAM. One by one, the FSM reads an object from the RAM, takes the new data from the object bounds module, and updates the RAM. If the updated data is off the screen, nothing is written to the ram, and the write address is not incremented.

Once all objects have been updated, the gameplay FSM then checks to see if the object placer is ready to add new objects. The state BEGIN_PLACE_OBJECTS is needed because of the once clock cycle delay between `add_next` being asserted and `placer_ram_data` becoming valid. The FSM keeps adding objects from the placer so long as `should_add` is high.

Next, the FSM stores the frog data from the frog FSM in the RAM, and sets `num_objs` corresponding to the number of objects which are in the RAM. Then, an additional 10 locations are written with the same frog data. This is to prevent the video unit from reading junk data if the two are not synchronized perfectedly.

Lastly, the FSM checks if any significant game conditions need to be handled, such as the frog dying or the frog finishing a level. Once these have been handled, the FSM returns either to an IDLE state, or, if the frog has run out of lives or a new level was reached, to RESTART. Notice that, from the RESTART state, the updating the RAM objects part of the game cycle is skipped, because there are no valid objects in the RAM at that point.

3 Video Unit

The video unit is responsible for displaying the game data to a video screen. The video format we chose is video graphics array (VGA), and we used a 640x480 bit resolution operating at 60Hz. The VGA output signal is generated by the ADV7125 codec, which is built into the new labkit. The video signal can output to any standard (LCD or other) computer monitor with the blue VGA pin cable. A block diagram of the video unit can be seen in Figure 5.

The video unit uses a major/minor FSM structure. The major FSM is the video FSM. The video FSM has three minor FSMs. The Video Out RAM Controller controls the video output signals, including the horizontal and vertical syncing and blanking, as well as the ZBT SRAM control and addressing for video output. The Write Blank FSM is responsible for writing a background screen to the ZBT SRAMs. The background is everything that doesn't move on the Frogger screen, including the road, river and grass. It should be mentioned that 12 bits of data are written to the SRAM, with bits[3:0] representing red, bits [7:4] representing green, and bits[11:8] blue. When data is sent to the video codec it is split up in these sections, with the lower 4 bits of each color being arbitrarily set to zero.

Finally, the Update Objects FSM is responsible for writing the moving objects to the ZBT SRAMs. These include the frog, cars, logs, and turtles. In order to update these objects, the Update Objects FSM must send control and address signals to the Game Unit, and receive back the data stored in the game unit regarding the position of each object.

Both ZBT SRAMs are used because as one is outputting video, the other is being overwritten with the next frame. At the end of every video frame (sixty times per second) the SRAM controlling the video switches. This is accomplished with the multiplexer that selects which SRAM data is passed to the AD7125 codec, using the video_ram_sel signal controlled by the video FSM.

Another important part of the video unit is that because all three minor FSM's need to control both ZBT SRAMs at some point (either for addressing the video output, or writing data),

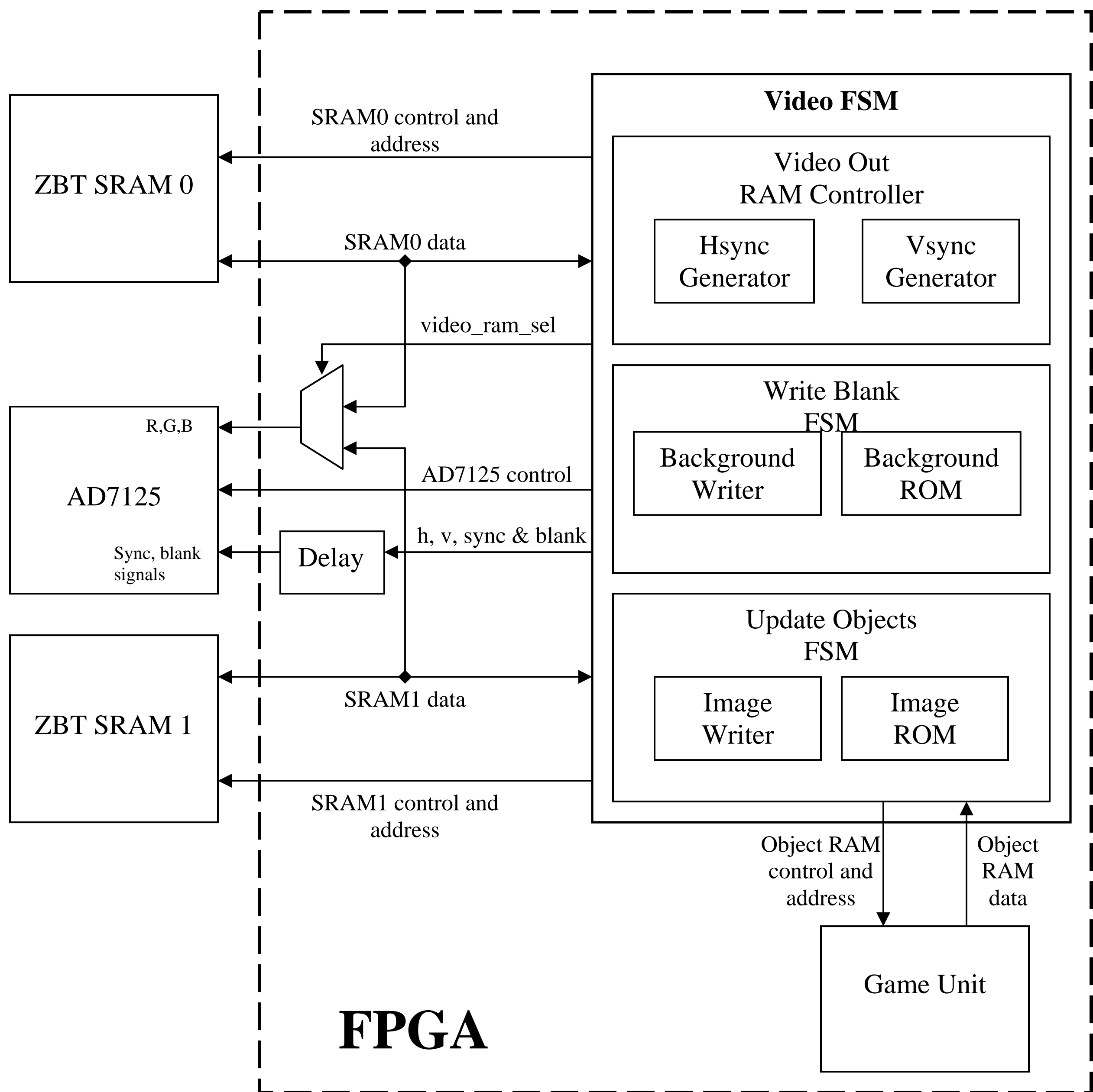

Figure 5: The Video Unit

there is some additional logic in the Video FSM that must output the correct enable and address signals at the correct time. This is discussed in the next section.

The final module in Figure 5 is the delay unit. This adds a two clock-cycle delay to the syncing and blanking signals passed to the AD7125. This is because there is a two-cycle latency for the digital color signals to be converted to analog video outputs, but the sync signals are passed directly to the monitors. In order for the syncing signals to correspond with the correct video data, this delay must be introduced.

3.1 Video FSM

The video FSM is shown in Figure 6. It uses a Mealy FSM structure with outputs on transitions. The main signals involved are shown in Table 1. On a reset the FSM goes into the RESET state until reset goes high. Then it goes into the IDLE state. It waits in the IDLE state until it receives a start signal from the Video Out RAM Controller indicating that Video output has begun. When this signal goes high, the Video FSM transitions into the VIDEO_OUT stage. It also sends a start_blank signal to the Write Blank FSM indicating that a blank screen should be written to the other SRAM, changes the video ram, and sets image_out_en low. The video output and blank screen writing are done in parallel in the VIDEO_OUT state, so that as video is being output from one of the SRAMs, every address of the other one is written the correct value for an empty background screen. This allows the full screen of background to be written during the video output so that the objects may be written in the video blanking period.

When video output is completed, the FSM transitions to the UPDATE_OBJECTS state. On this transition start_objs is set high, as is image_out_en, which is held high during the entire UPDATE_OBJECTS cycle. This results in the Update Objects FSM writing the updated images to the SRAM. When the done_objs signal from the Update Objects FSM goes high, objects have been written, and the FSM goes back to the IDLE state waiting for the next video cycle to start. This is all done in 1/60th of a second every video cycle, with the majority of the time being spent in the VIDEO_OUT state. The other minor FSMs are discussed briefly below.

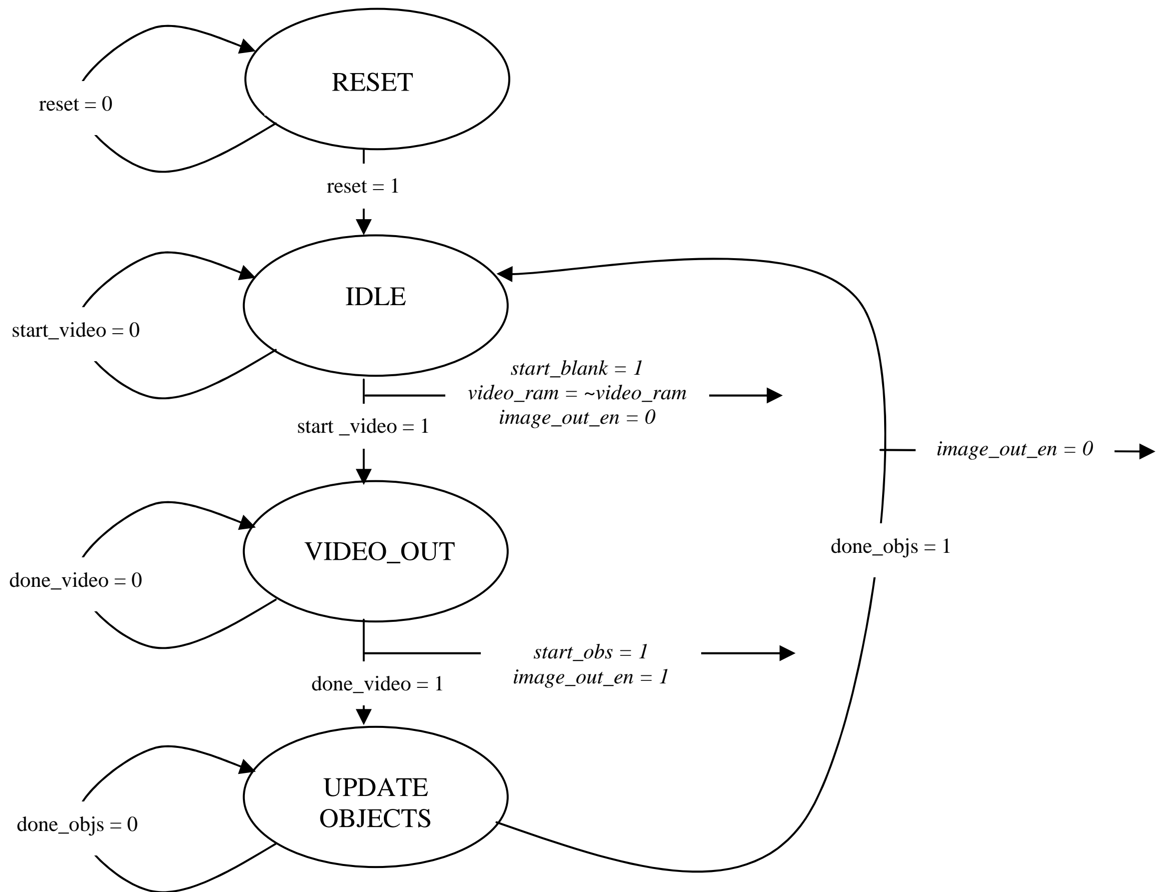

Figure 6: The Video FSM

Name	***Type***	***Source/destination***	***Description***
reset	Input	User	Causes the system to reset; disables video
start_video	Input	Video_Control_FSM	Indicates that video output is about to begin
done_video	Input	Video_Control_FSM	Indicates that video output has completed
start_objs	Output	Update_Objects_FSM	Starts writing the moving objects to the SRAMS
done_objs	Input	Update_Objects_FSM	Indicates that writing new objects has completed
start_blank	Input	Write_Blank_FSM	Starts writing a blank screen to the SRAMs
image_out_en	Control	Video_FSM	Controls whether Update Objects or Write Blank FSM controls SRAM
video_ram_sel	Control	Video_FSM	Controls which RAM outputs video

Table 1: Signals in the Video FSM

3.2 Video Out RAM Controller

The Video Out RAM Controller is basically an interpreter for the horizontal and vertical sync generators. The function of the sync generators is to control the horizontal and vertical syncing and blanking signals. They also keep internal counts of what row (for vertical) or column (for horizontal) they are on. The Video Out RAM Controller concatenates these two values to generate the SRAM address, which is output through the Video FSM to the SRAM. Each of these values is 10-bits, so the lowest-order 10-bits of the address is the horizontal location of the pixel, and bits [19:10] represent the vertical location. The Video Controller also ensures that the SRAMs are in read-mode by setting oe_b low and w_b high during the video cycle.

3.3 Write Blank FSM

The Write Blank FSM writes the background screen to the SRAM. When it receives a start signal from the Video FSM, it starts at location zero and writes 23x30 bit blocks across the screen. The color of the block to be written is determined by the Background ROM, which allows a different color to be written to every block. The Background ROM is an asynchronous ROM with a nine-bit address, representing the location (in blocks) of the image on the screen. The five lowest-order bits represent the horizontal block address, and the four most-significant bits represent the vertical address. The Background Writer module takes in the horizontal and vertical block numbers and outputs the correct color to every individual pixel in that block.

3.4 Update Objects FSM

The Update Objects FSM is similar to the Write Blank FSM. On receiving the start_objs signal from the Video FSM it begins, accessing the game kit for information on the object type and location at location zero. Then it sends this data to the Image Writer, which accesses the Image ROM for the color information about the specific image to be written and sends the color data to the SRAM with the correct addressing information. When the Image Writer is done with an image, the Update Objects FSM moves onto the next location in the game unit's ram until all of the objects have been written. The num_objs input from the game unit determines this threshold value.

4 Testing/Debugging

The testing and debugging process was long and gradual. The complexities of the game unit made it difficult to test until the video unit could display things properly on the screen, so that was our first goal.

Testing the video unit consisted of several steps. The first step Cory took was using the sync generators to create a signal the could be sent the the AD7125 codec and output a colored screen (keeping the red, green, and blue pins constant). This step was quick, and there were no problems. The second step was interfacing the video to read from a ZBT SRAM. This involved building controllers to both write and read from the SRAMS. The reason we needed to be able to write to them was that the SRAMs contained random data on startup, so they needed to be blanked to see if we could display a pixel in specific locations. When this step was complete we could combine our units for the first time, and Nate could check if we were able to control the movement of the frog (a single white pixel on a black screen) correctly.

Once this was working we worked on creating a system to display all of the objects from the game unit on the screen. This was done by displaying different colored boxes for each object type. This allowed us to test and debug issues with the game unit.

Attempting to simulate an entire working game can be tedious and often uninformative, since problems may get lost in the thousands of transitioning signals. Therefore, the smaller parts of the game unit were simulated, starting with the pieces of combinational logic such as the interpreter, the object bounds, and the overlap tester. The small modules not mentioned in the description in section 2 were also simulated. Once these were working, the more complex parts, such as the object placer and the frog FSM were tested. Once these were all tested and simulating correctly, the gameplay FSM and the RAM were tested, first without input from other modules. Gradually, more modules were connected and simulated together; however, as the circuit became more complex and more objects were present, simulations became overly large and difficult to examine, and therefore, less useful. At this point, luckily, the video unit was working, so that the gameplay unit could be tested by actually playing the game. Simulations were still used after this point, but generally only to find the source of a problem identified with the help of the video unit.

The only part of the gameplay unit that caused significant difficulties was the transition between levels. The relative timing of the objects and the frog being updated caused quite a few errors. This caused us to changed the design of the two FSMs slightly, and finally, the levels work correctly.

Burning the images to a ROM was a tedious final step. We were unable to get any of the automatic file generating scripts working, so we were forced to hand code the images onto a .coe file. Because of the building up process of the previous tests, all we had to do was replace the ROM that generated colored blocks with the one that generated images, and this worked on the first try.

5 Conclusion

The laboratory project was a success. Both the gameplay unit and the video unit functioned correctly. While programming the ROM bit by bit for each of the 1380 pixels for each of a few dozen images was tedious, the effort paid off because the graphics looked better than we had hoped. Debugging a system as large as this was often frustrating, especially when seemingly minor changes in one unit caused major errors in the other unit. However, all these problems were solved and the Frogger game worked correctly, and looked almost like the arcade version.