FPGA Sudoku Player

Final Project
ECPE 174: Advanced Digital Design

Emily Estrada, Michael Kmak, Christopher Webster
2020-12-09

Project Overview and Objectives

The main goal of this project was to use the DE2-115 development board to host a playable game of Sudoku. Games would be predefined in the memory system, and the system should be able to solve the game as a way to check whether the player's solution is correct. The system would be interactive, able to both take player input and display the current game board. These original goals proved to be out of reach in some places and achievable in others. The project goals were stepped back in a few stages before arriving at the current solution. These changes will be described in the appropriate sections of the report.

This design can be clearly separated into three distinct portions: player input, game logic, and display control. Because of this clear divide, this report will discuss the three modules in their own sections when appropriate. In addition, the design allowed each module to be assigned to a group member. Chris wrote the game logic unit, Emily designed the input system, and Michael programmed the display controller.

Solution

Design Description

The interactivity would be accomplished by integrating two peripherals to the system. The DE2 has USB and VGA ports onboard, so a USB keyboard and VGA monitor were chosen. The keyboard would be able to move a cursor on-screen and input a number at the selected cell, and have keys to check the board and request a new board. The onboard game logic module would then handle the input and update the board accordingly, as well as perform the other related tasks. The display controller would read the game board from the logic module and translate it to a framebuffer. This would then be displayed on a VGA monitor. A single top-level module would be used to tie these three subsystems together, as illustrated in the system diagram below in Figure 1.

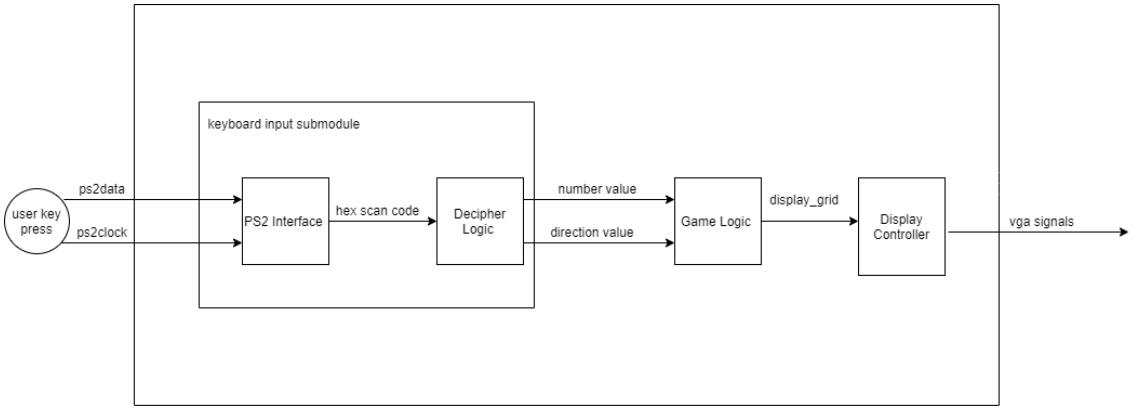


Figure 1. System Diagram

Keyboard Input

The keyboard submodule consists of two modules: the PS/2 interface and the key scan code decipher logic. The system was designed to use an external keyboard that was to be connected though a PS/2 port on the FPGA board. Once a key is pressed the data would be sent through the PS/2 port, manipulated through the PS/2 module, and then this manipulated data would then be sent to the decipher logic to determine which key was being pressed. Once this key was determined it would then be sent through to the game logic submodule.

The PS/2 port receives two main inputs, one is the data that is being sent and the other is the clock. Within the PS2 interface submodule, the data and the clock both had to be synchronized in order to work properly with the system. Once these signals were synchronized, the PS2 data then had to be stripped in order to send the hexadecimal scan code to the decipher logic. In order to do this the data was shifted into registers and the last 8 bits of the PS2 data were located in these registers and these 8 bits of data were the keyscan code of the key that was

pressed. Once the keyscan code data is within these shift registers, they are then combined to make the final scan code output that is to then be sent to the decipher logic.

Once the decipher logic receives this 8 bit scan code, it then goes through each accepted key press case. These cases consist of 1,2,3,4,5,6,7,8,9,w,a,s, and d. For the cases that are numbered the decipher logic would return their four bit number value, and when pressing a number this would mean that the user would be on their desired square and thus the decipher logic would also return a position value of zero. The w,a,s,d keys cover the navigation cases for the keyboard, and thus are designed to return position values other than zero and number values of 4'b1111 so that they will not register in the game logic of having a valid number. The position values are 1,2,3, and 4 for keys w,d,s, and a respectively. Each case has the respective key hexadecimal value, if the scancode matches this hexadecimal value it produces the respective value and position output which is then sent through to the game logic submodule as inputs.

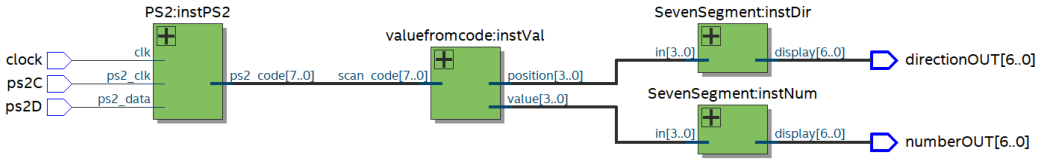


Figure 2. Design of Keyboard Submodule

Game Logic

The game logic submodule consists of three different components. These components are the board selector, board control, and board update. Using these modules together, the user is able to play a game of sudoku with a preloaded game.

The first module used in the game logic section is the board selector module. This module is used to load in a sudoku game board. The user currently can select between one or two preloaded sudoku games. This module is also used to reset the game board. If a user notices lots of mistakes in their game, by flipping a toggle switch the game board can easily be reset to its initial starter values. The next module used in this system is the board control module. This module reads in the keyboard inputs, and converts these inputs into positional coordinates that the rest of the game logic submodule can understand. For example, if 'w' is input on the keyboard, then the current row position is incremented by one on the board. If 's' is input on the keyboard, then the current row position will be decremented by one. The next module used in this system is the board update module. This module is used to update the current game board with the input keyboard value. This module will verify that the user does not enter a number over the original given value. If the user is entering a number in a valid position, then this user number will be input into the game grid at the current board coordinates. Using these modules together allows for a basic implementation of sudoku game logic.

Display Controller

As mentioned, the system went through some downscaling as the project was completed, and the display controller is no exception to that rule. However, the final module was quite close to the plans, and most of these were fairly minor compromises. The display controller module works as a series of steps, beginning from the game board and ending at the external monitor. This flow can be seen in the module diagram below.

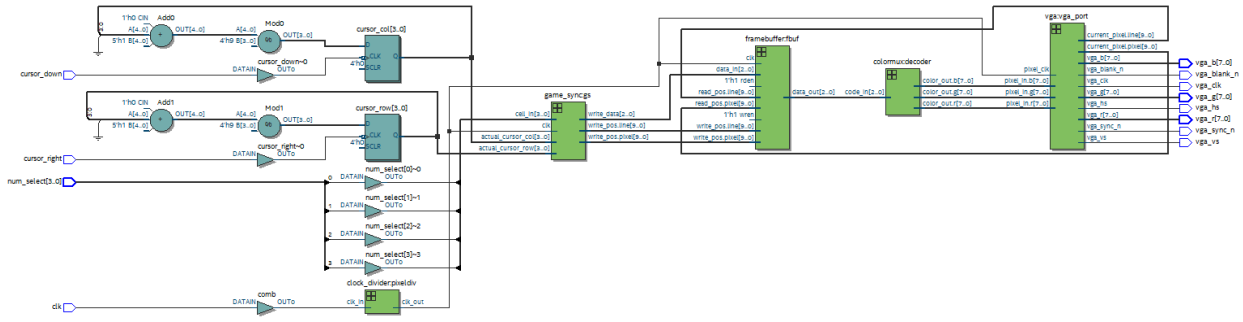


Figure 3. display_controller system diagram

Signals begin at the game_sync module. This module reads the board and writes the proper digit to the right location in the framebuffer. The font_provider module stores the digits (0-9) in 40x40 bitmaps in ROM. If the pixel is a 1, then it is written to the framebuffer as a black pixel. If it's 0, then it becomes a background-colored pixel. game_sync acts as an FSM, which assures that the module is only writing one thing at a time (either the cursor or a digit) if it's not scanning. The framebuffer is implemented as a 640x840 array of 3-bit signals in memory. The vga module generates the signal that is sent through the VGA port to the screen. It requests the pixel it needs from the framebuffer. The 3-bit code for that pixel is exchanged for a 24-bit color at the colormux module before being sent to the vga module.

In the opposite order, design began with the vga module, which generates the signal that is sent through the VGA port to the monitor. The first small compromise is here: a 60Hz display requires a pixel clock running at precisely 25.125MHz. The DE2's clock runs at 50MHz, so the pixel clock had to be 25MHz, providing approximately 59Hz at the screen. The module works by incrementing two timers, one for the pixel and one for the line. Depending on the position of these two timers, the module outputs the proper signals. VGA protocol defines that each segment be formatted as follows in figure 4, which is from the DE2's User Manual.

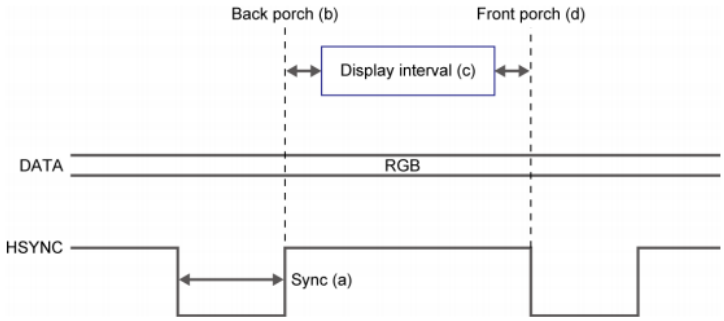


Figure 4. one cycle of a horizontal VGA signal

While the HSYNC or VSYNC signals are low, the RGB signals should output logic 0. During the front and back porch, RGB should also stay low. This is true for the output, however the RGB DAC on the board has an input signal called VGA_BLANK_N that will set the RGB output to logic 0. So rather than changing the color to black, this signal is set during the porches and sync. This pattern is repeated once for each line of pixels on the HSYNC signal, and again for every frame on the VSYNC signal, and the length of each segment (measured in pixel clock cycles) tells the monitor what resolution and refresh rate to use. For standard VGA 640x480 @60Hz, the timings are as follows, where 1 pixel is 25.125 MHz and 1 line is 800 pixels:

	Sync	**B. Porch**	**Display**	**F. Porch**	**Full Cycle**
Horizontal (pixels)	96	48	640	16	800
Vertical (lines)	2	33	480	10	525

Table 1. timing parameters for 640x480 60Hz VGA mode, from [10]

Pixels are retrieved from a framebuffer. This is where some major compromises were made to the design. The initial plan was to have two framebuffers that would swap between frames, one for reading and one for writing. However, this would have used far more memory than the board has. After a few backwards steps, the final display controller utilizes one 3-bit framebuffer implemented with simple dual-port memory. The vga module reads, and the game sync module writes. The issue with having only one framebuffer is that there's the possibility of screen tearing. Screen tearing is when a frame is only partially done being drawn when it is sent to the screen, resulting in half of an old frame and half of a new frame being displayed. This is a much less important problem than it would be in say, a DVD player or modern video game, since only one cell of the board will ever be changing at the same time. The pace of sudoku is slow, and only very little of the screen will need to change at once, so it arguably makes more sense to save the memory and use a single framebuffer, especially because one torn frame won't look too bad at 60Hz. The other compromise is in the 3-bit storage method. Even a single framebuffer holding the full 24 bits of color is far too much memory. Instead, the buffer holds a 3-bit color

code, and when the pixel is read by the vga module, it uses that code to retrieve a full 24-bit color from a palette, located in the colormux module.

The last thing to be written is the first module that a signal will go through, the game_sync module. This module reads the game board and writes it to the framebuffer. The compromise made here is likely the most significant when it comes to speed. Ideally, the module would scan the game state and compare it to a local copy, only writing changes to the framebuffer. Compromises had to be made and instead, the module constantly reads from the game board and constantly rewrites the framebuffer. The module contains a 5-state FSM. In the idle state, it's just scanning the game board. The two signal states are intermediary states to activate the drawing modules, which then change the state to their respective draw states. If a module tries to signal a change while the other is drawing, then it waits until the other is finished. Drawing the cursor takes priority over new digits. Both draw modules contain submodules that translate local positions (0,0 to 40,40/48,48) to real positions (0,0 to 640,480), and submodules to provide the correct pixel for either the cursor or the selected digit. Digits and cursors are stored in ROM, and initialized with MIF files.

Similarly, the framebuffer is initialized with a very large MIF file. These were generated with a snippet of C code. Since the background of the game will never need to be redrawn, it is simply initialized in the framebuffer's MIF. The fonts were first created as PNG files, converted to byte arrays by an online tool [6], then converted to a MIF via a snippet of C. The framebuffer's MIF was generated with a very similar C snippet.

Testing Procedures

Due to the complex nature of the design, a full-system testbench was determined to be out of reason to write. In addition, none of the three members had both a VGA monitor and a PS/2 keyboard, so the full system could not be human-tested. Accordingly, our testing took a more transitive approach at the higher levels. If each of the three modules work individually, and the two peripherals can both communicate properly with the game logic, then it is reasonable to assume that the system as a whole should work.

Keyboard Input

In order to verify the functionality of the keyboard component of the system, a testbench was created to verify the keyboard submodule and afterwards the FPGA was programmed for hardware verification.

The testbench created used specific test cases in which all possible keys that would contain values would be tested along with some keys that would not work if pressed. Randomness in this testbench was not necessary since the primary goal of verification was to test the specific keys that would be used from the keyboard. The specific keys that were being tested were the numbers located at the top of the keyboard (1-9) as well as the w,a,s, and d keys that would be used to navigate through the keyboard.

Once the testbench confirmed proper decoding of the key's hexadecimal values, the submodule was then programmed onto the FPGA board for final functionality checks. The number and position values were programmed to be visible on the seven segment display, which allowed for easy verification of keys. These functionality checks consisted of testing the keys 1-9, the w,a,s,d, and other keys whose scan codes should result in no display on the seven segments.

Game Logic

In order to test the game logic submodule, test benches were created for each individual module to test basic functionality, and to test potential corner case issues. In order to test the board selector module, a self checking test bench was utilized. The purpose of this test bench was to verify that based on different inputs, different game boards would be loaded in. The first set of tests in this test bench are used to verify that game board one is loaded using the right inputs of board selection and board enable variables. This test verifies that game board one is loaded in by checking the values in the current board to see if they match game board one's values. The next set of tests verify that board two is successfully loaded. Using the correct input values to load in board two, different positions in board two are tested to prove that board two has been loaded.

The next module tested is the board control module. This module also uses a self checking testbench to test its functionality and corner cases. The purpose of this module is that given a keyboard input, update the current 'x' and 'y' positional variables for the game board.

This module was tested by simulating different inputs that the keyboard would have, and verifying what the positional coordinates would be after this input. Throughout this testbench simulated inputs of ‘w’, ‘a’, ‘s’, and ‘d’ were simulated to show that the positional coordinates do update their value on these inputs, and that they stay within their boundary range of less than 9 and greater than 0.

The final test bench created for this submodule is for the board update module. This testbench is another self-checking test bench to test functionality and edge cases. This test bench works by verifying that given positional coordinates on the game grid and a user value, the final game board should then display this updated value. The first set of tests verify that this functionality works. The other functionality that this module has is that if the original game board gives you a cell value, the user cannot overwrite this value. The next set of tests verify this functionality by assigning the original board a set of values, and making sure that in these tests new values are not overwriting the original values. These three testbenches are used together to verify the functionality of the game logic submodule.

Display Controller

Display controller verification was done as work progressed. The testbenches here simply allowed the system to run for a period of time and recorded the resulting waveforms. While it's certainly possible to make more robust, self-checking testbenches, from the very beginning the project was such that these kinds of testbenches would take a lot of time to write for very little gain. It seemed far more efficient to look at the results or the VGA display. The first testbench was written for the vga module itself, which drives the display signals. A self-checking testbench could be used to verify that each signal is being driven correctly, but it was much simpler to judge the system by eye. Such a complicated testbench would be prone to its own bugs and failures, and its implementation would basically mirror the system itself. Having a testbench verify several signals over 400,000 clock cycles is a bit much, so instead the code was just programmed to the board. If the monitor turned the right color(s), then the system worked. If not, then the waveform would be analyzed by hand and the problem determined.

After verifying that the signals were correct, the framebuffer was programmed. From here onwards, most of the testing was done via the external monitor. Looking at a waveform to determine what a screen would look like just doesn't work. Most issues could be resolved by checking the board. For example, the first MIF file generated for the framebuffer did not result in vertical lines of color as expected. Upon closer inspection, it seemed like the first 20 pixels were being cut off, and the last few lines were never initialized. The source of this was that the framebuffer was accidentally defined as 660 by 480 pixels. MIF generation was tested by just trying it and fixing until it worked. Again, trying to verify the massive mif file is an ordeal in itself and it's far more efficient to just load it and look at it.

For the game sync module, testbenches were more useful. The game sync module was implemented as a state machine. A testbench was used to watch it work, without randomization

or verification. To verify that the states were changing as expected, a testbench was perfect. However, to verify that the proper checking or drawing was happening under the hood it's again far easier to see the patterns in the signals by eye than it is to program verification. Likely because of the weird implementation with 4 submodules, Quartus did not recognize game_sync as containing an FSM, and thus a state diagram will not be provided.

Overall, the display controller was far easier to test and verify by eye than it was to use a testbench. With so many signals that need so many clock cycles to operate, a quick glance to determine if the pattern works as expected seems more effective than spending hours writing incredibly robust testbenches. As a final subsystem test, I programmed the controller to take input from switches 0-3 and display that number on every square, as well as use two of the onboard buttons to move the cursor.

Results

Keyboard

The simulation of the results for the keyboard decipher logic are shown in the figure below. This demonstrates the ability for the submodule to accurately handle the hexadecimal scan codes that it receives from the shift registers of the PS/2 interface.

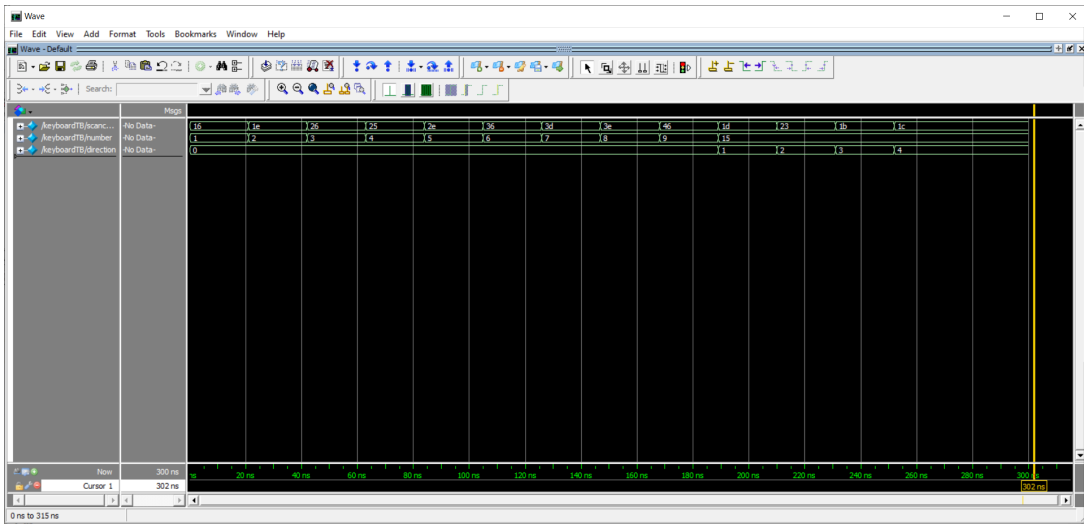


Figure 5. Simulation Results of Keyboard Submodule

As mentioned previously in the Keyboard Testing Procedure, once there was verification that the submodule was working as intended from the simulation results, the keyboard submodule was then programmed onto the FPGA for hardware verification. After successful compilation in Quartus,as shown in the figure below, the resulting sub module used a total of 92 logic elements, 37 registers, and 17 pins. The testing procedure for the keyboard was then followed, returning successful functional results on the seven segment display for the all programmed keys (1,2,3,4,5,6,7,8,9,w,a,s,d).

Flow Status	Successful - Wed Dec 09 19:25:22 2020
Quartus Prime Version	18.0.0 Build 614 04/24/2018 SJ Lite Edition
Revision Name	test1
Top-level Entity Name	toplevel
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	92 / 114,480 (< 1 %)
Total registers	37
Total pins	17 / 529 (3 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 6. Compilation Results of Keyboard Submodule

Game Logic

In order to test that the game logic worked as expected, test benches were designed for each used module. The figure below shows the test bench of the game board selector module. This module verifies that different game boards can easily be loaded into the sudoku player, along with boards being able to be reset.

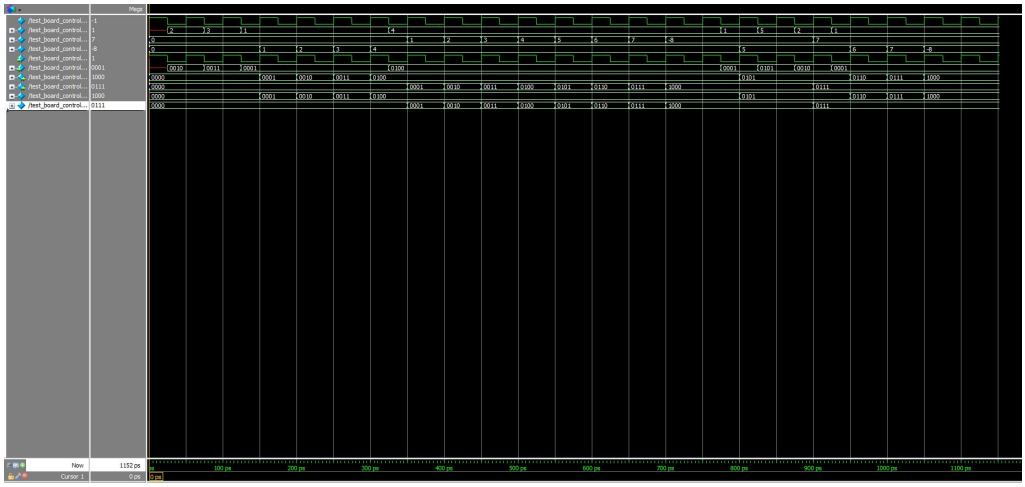


Figure 7. Simulation of Board Selector Module

The next figure is verification of the control module. This module is used to update the game board positional coordinates based off of the user input. Verification of this functionality is shown below as the positional coordinates increment and decrement, and do not go past their boundaries of less than nine and greater than zero.

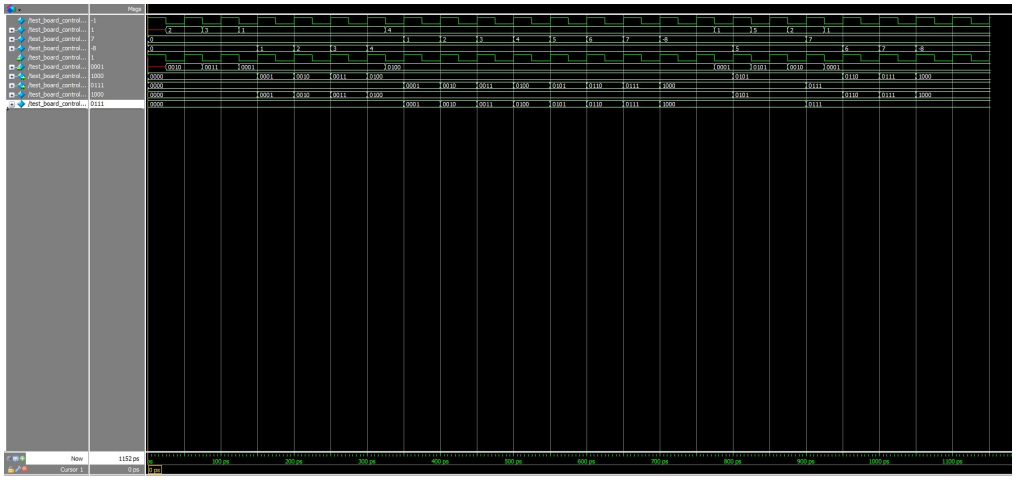


Figure 8. Simulation of Board Control

The last simulation used in this submodule to verify the game logics functionality is the board update module. This simulation verifies that a user can successfully enter and save their input values into each square, along with not allowing the user to write values to initial squares. This simulation is shown in the figure below.

Figure 9. Simulation of Board Update

Display Controller

The display controller functions quite well. It's able to perform its job of reading from the game board and displaying the results to the screen, as illustrated below

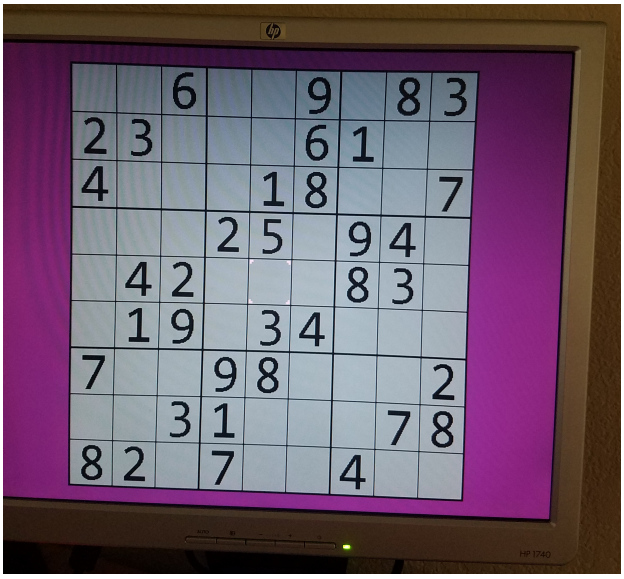


Figure 10. Functional display of a game board

One issue noted during testing is that the button clicks will occasionally trigger strange cursor movement, sometimes putting the cursor out of bounds, where it will leave a trace forever, because the background is never overwritten. I believe, however, that this is an issue with synchronizing the onboard buttons, and might not occur when reading actual keyboard input.

As discussed in the testing section, there were few testbenches for the system. The first of which tested the vga module, and its waveform is below in whole as well as zoomed in to a single line.

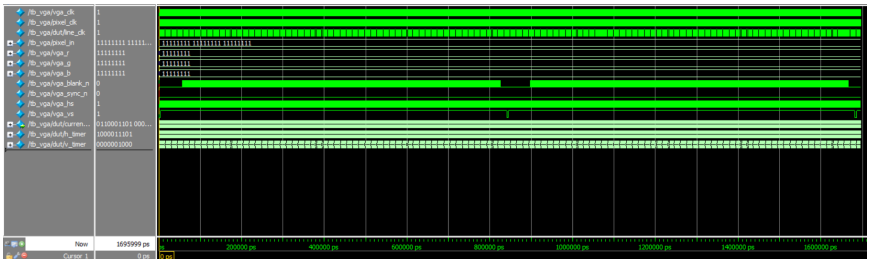


Figure 11. Simulation of 2 frames sent over VGA

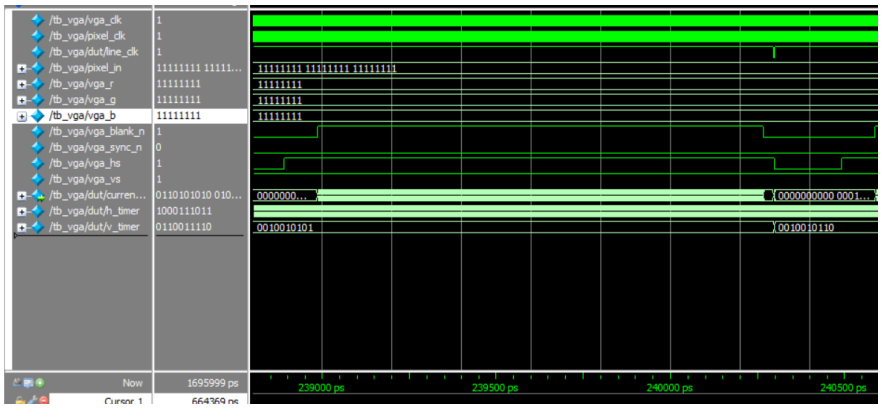


Figure 12.Simulation of one line sent over VGA

The test runs for almost 900,000 clock cycles and involves an intricate system of timers and smaller clocks. Because of this, verification was not implemented. It's far easier to check by eye that the timers are incrementing correctly and use the screen to see whether the timing is correct (because it won't display properly or at all). Below is a later testbench, used to verify that the cursor signalling and drawing state takes priority over the cell drawing, as well as watch the drawing process.

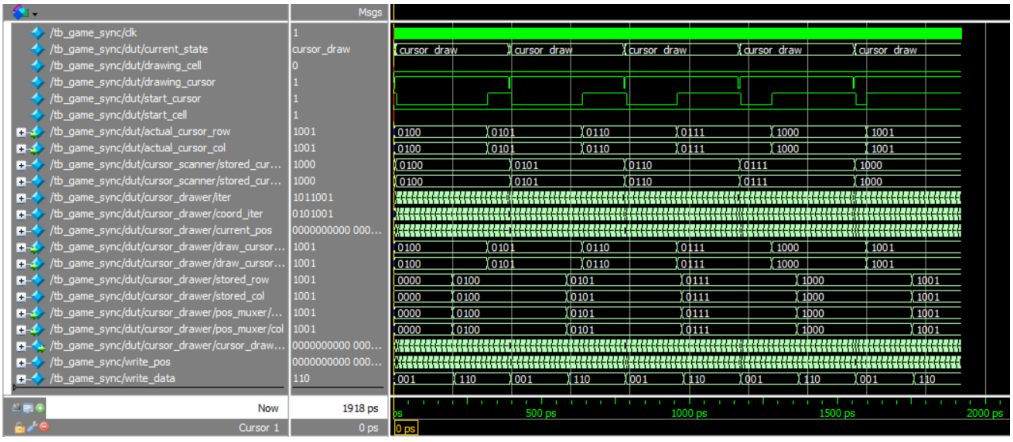


Figure 13. Testbench for the game_sync FSM

Full System

As mentioned previously, the design was not able to be fully assembled for a hardware verification of the system. However, the full system was successfully compiled in Quartus, and displayed the expected logic system based on RTL viewer results. Reference the Analysis Section for further details on the full system design results.

Analysis

The final system design required a total of 2,900 logic elements, 863 registers, 36 pins, and 23% of the system memory. This system was not optimized in terms of space. A great deal of logic elements were used in the game logic sub module and this would be the sub module focused on for improving the overall space of the design. Since this design had a more complex system than any of the group members had experienced previously, a proper timing analyzer was not developed. However based on previous knowledge in improving timings of systems, this system would have a significant delay based on the total logic elements and the complexity of the design. The one upside to timing in terms of the game logic, is that all of the logic elements are stacked based on the RTL viewer so the game logic system was optimized in terms of flattening the logic structures which will help in overall timing.

Future work for the keyboard module would be the incorporation of a delete button, this also ties into game logic and how it would handle deleting data from one of the squares, and the implementation of extraneous keys that hold the same values for the keys already present.

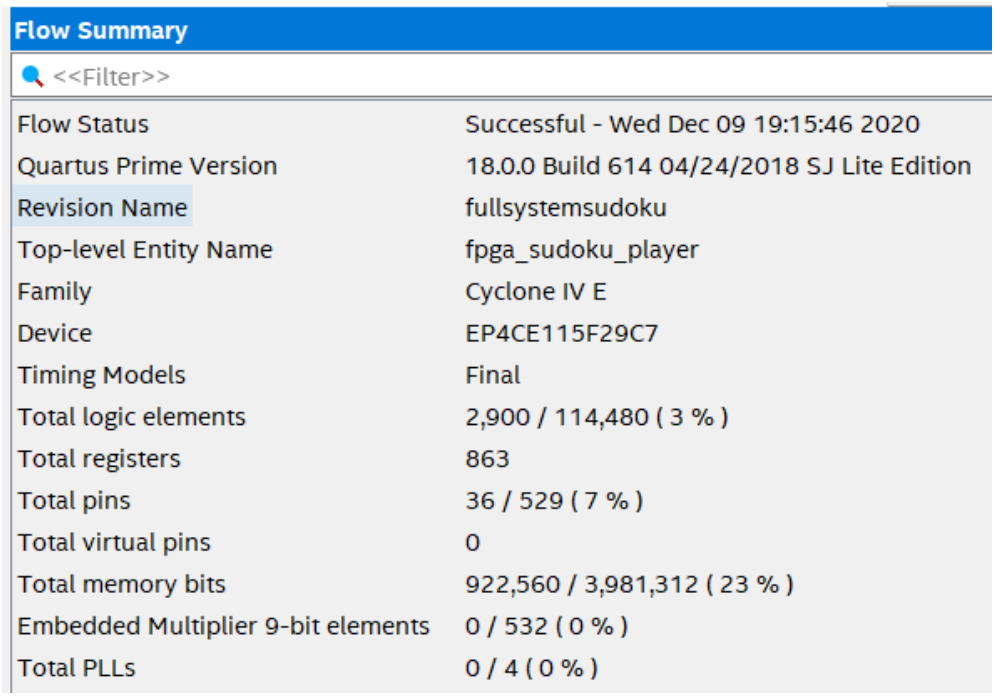
An item that was not thought of but was brought to the groups' attention during the system demonstration was the user's ability to delete a value from a square. Currently the system operates so that if the user would want a different number in the square they would select the desired key and it would overwrite their previous choice. However, the implementation of the delete key would not be overly complicated. This would require the inclusion of the delete (backspace) key hexadecimal case and a corresponding number and position which would theoretically the value would be 4'b1111 so that the game logic processes the number as a blank and the position would be 4'b0000 so that the game logic processes it as the current square.

Another luxury item that could be added to the keyboard module would be the implementation of the special navigation keys (found on an extended keyboard) that are strictly the up, down, left, and right arrows, as well as the numbers found on a keypad. This would be more complex since these keys would require special break characters to be instantiated, however would be possible to include with more key scan code research.

For the game logic submodule, one future improvement could be a win checking module. This module would check every time a new value is entered if the user has won or not. This could be done by having nine instances of a module to count if there are numbers one through nine in a row, if there are it would return true, if not it would return false. If all nine rows return true, then the user has won. Another future work that could be done on the game logic submodule is improving the total logic elements that are used in this module. There are lots of copies of the game board that are used as intermediate values that use up lots of logic elements. This section could easily be redesigned and improved for efficiency rather than completion.

As for the display controller, there are a few things to be done in the future. First and foremost would be to change the game sync module to only write to the framebuffer when the

board changes. This would likely warrant a change in how the game board is passed to display_controller, ideally creating a local copy of the board that is constantly compared to the master board within the game logic subsystem. This would greatly reduce the writes to the framebuffer, which is probably worth the extra cost of implementing it. As well, it would be nice to differentiate between the different numbers on the board. Those that were given would be displayed in black, numbers entered by the user in gray. When the board is checked, correct guesses would be in green and incorrect in red. This would also require some change in the game logic. Beyond these major changes, there are some light bugs to squash and the handling of wren and rden in the framebuffer module. Proper practices are to disable the signals when you're not actually reading or writing, but as of now they're both permanently enabled. In general, there are likely plenty of cleanup jobs and optimizations to be done as well.



Flow Summary

<<Filter>>

Flow Status	Successful - Wed Dec 09 19:15:46 2020
Quartus Prime Version	18.0.0 Build 614 04/24/2018 SJ Lite Edition
Revision Name	fullsystemsudoku
Top-level Entity Name	fpga_sudoku_player
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	2,900 / 114,480 (3 %)
Total registers	863
Total pins	36 / 529 (7 %)
Total virtual pins	0
Total memory bits	922,560 / 3,981,312 (23 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 14. Compilation Summary for Final System

Conclusion

Although the final design was not able to be verified functionally due to lack of external hardware amongst group members, each group member was able to verify their own separate subcomponents. As earlier, if we could prove that the three submodules all work independently, and that the keyboard and display can each connect to the game logic properly, then that is sufficient proof that the entire system will work. Emily developed her knowledge on the inner workings of the PS/2 port and how data is passed through it both through a keyboard and mouse. She was able to have the user pressed keys register correctly through her decipher logic. Chris better learned how 2-d logic arrays work in system verilog. Chris also improved his knowledge of how to work with large variables that are used in multiple functions and how intermediate variables need to be used in these instances. Mike researched and learned quite a bit on the VGA standard, as well as framebuffers and low-level video output. There were some interesting hurdles presented by trying to implement graphics in a hardware synthesis language, since many of the originally planned techniques were much more reasonable in software if not impossible for hardware synthesis. Using an 8-color palette is akin to early video games using very similar methods to cram their content into the tiny storage on a cartridge and get it to run on the console hardware.

For this project, Emily, Chris, and Mike all worked on developing their own subsystems and test benches for these subsystems. These designs were worked on independently for the most part. Discussions between Chris and Emily occurred with how the keyboard inputs would be stored so the game logic could work with these inputs. Then likewise, discussions between Mike and Chris occurred with how the display grid would be output so Mike could use this data in his subsystem. For the presentation, everybody helped equally create the slide show, along with everybody helping equally writing this document.

References

[1] Archiveddocs, “Key Scan Codes,” *Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-6.0/aa299374(v=vs.60)?redirectedfrom=MSDN. [Accessed: 27-Dec-2020].

[2] E. Manj, “Verilog-PS2-LCD-Interface,” *GitHub*. [Online]. Available: https://github.com/EliasManj/Verilog-PS2-LCD-Interface. [Accessed: 25-Nov-2020].

[3] “EEVblog Electronics Community Forum,” *Help with PS2 keyboard interface?* . [Online]. Available: https://www.eevblog.com/forum/fpga/help-with-ps2-keyboard-interface/. [Accessed: 27-Nov-2020].

[4] Embedded Thoughts, “FPGA Keyboard Interface,” *Embedded Thoughts*, 16-Nov-2016. [Online]. Available: https://embeddedthoughts.com/2016/07/05/fpga-keyboard-interface/. [Accessed: 23-Nov-2020].

[5] G. Callen , “verilog,” *GitHub*. [Online]. Available: https://github.com/DendriteDigital/verilog. [Accessed: 24-Nov-2020].

[6] *image2cpp*. [Online]. Available: https://javl.github.io/image2cpp/. [Accessed: 10-Dec-2020].

[7] Keyboard scan codes. [Online]. Available: https://www.freepascal.org/docs-html/current/rtl/keyboard/kbdscancode.html. [Accessed: 25-Nov-2020].

[8] Keyboard scancodes: Keyboard scancodes. [Online]. Available: https://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html. [Accessed: 23-Nov-2020].

[9] S. Larson, *PS/2 Keyboard to ASCII Converter (VHDL) - Logic - eewiki*, 01-Aug-2018. [Online]. Available: https://www.digikey.com/eewiki/pages/viewpage.action?pageId=28279002. [Accessed: 24-Nov-2020].

[10] “VGA Signal 640 x 480 @ 60 Hz Industry standard timing,” *TinyVGA.com*. [Online]. Available: http://tinyvga.com/vga-timing/640x480@60Hz. [Accessed: 10-Dec-2020].