A Theatre Lighting Board

Maura Cordial and Irene Zhang

Introductory Digital Systems Laboratory

December 13, 2006

Abstract

The goal of this project was to design and implement a lighting board controller. The lighting board can be used to control eight dimmers, each of which can power up to two theater lights. Each dimmer is assigned to a channel that can be programmed at different intensities(0%–100%) for each of the 127 cues. Each cue has an up time and down time, up to 255 seconds, indicating how long it takes to bring the channels up to their respective intensities for the cue and back down at the end. There are also some optional parameters for each cue: wait, follow and link. Wait indicates a period of time to wait before bringing up the new cue after the go signal. Follow indicates a time period to keep the cue up before automatically continuing to the next cue. Link indicates the next cue to go into. Link can be either sequential or used to create loops and jump between cues. The user will program cues using a standard computer keyboard and screen. The screen will display detailed information for a single cue and basic information for the previous and next cue. The board will operate in two modes: blind and live. In live mode, changes to a cue will affect actual channel intensities in real time. In blind mode, the changes will not be visually seen until that cue is in live mode.

Contents

1 Introduction **4**

2 Overview **4**
2.1 Board Specifications . . . 4
2.2 Screen . . . 4
2.3 Keyboard . . . 5

3 Design **5**
3.1 System Overview . . . 5
3.2 Screen Display by Maura . . . 5
3.2.1 XVGA Module (xvga.v) . . . 5
3.2.2 Blob Module (blobv.v and blobh.v) . . . 7
3.2.3 Character String Display Module (cstringdisp.v and cstringdisp2.v) . . . 7
3.2.4 Static Sprites Module (static_sp.v) . . . 7
3.2.5 Binary to String Converter (bin_string.v) . . . 7
3.2.6 Color tracking Modules (color_tracking.v and color_tracking2.v) . . . 9
3.2.7 Dynamic Sprites Module (dynamic_sp.v) . . . 10
3.3 Keyboard Handler by Maura . . . 11
3.3.1 PS2 to ascii Module (ps2_ascii_input.v) . . . 12
3.3.2 Keyboard Interpreter Module (keyboard_interp.v) . . . 12
3.4 Processor by Irene (processor.v) . . . 16
3.4.1 Instruction Buffer . . . 16
3.4.2 Control Logic (ctrl_logic.v) . . . 16
3.4.3 Arithmetic Logic Unit (alu.v) . . . 18
3.4.4 Registers . . . 18
3.4.5 Cue Memory . . . 19
3.4.6 Macros ROM . . . 19
3.5 DMX Controller by Irene (dmx.v) . . . 20
3.5.1 Channel Twiddler Module (channeltwiddler.v) . . . 21
3.5.2 DMX output (dmxcontroller.v) . . . 21

4 Testing **23**
4.1 Screen Display Testing . . . 23
4.2 Keyboard Handler Testing . . . 24
4.3 Processor Testing . . . 25
4.3.1 Control Logic Testing . . . 25
4.3.2 ALU testing . . . 25
4.3.3 Processor Integration Testing . . . 25
4.4 DMX controller Testing . . . 26
4.5 Integration Testing . . . 26

5 Conclusion **29**
5.1 Keyboard Interpreter . . . 29
5.2 Screen . . . 29
5.3 Processor . . . 29
5.4 DMX . . . 29
5.5 Overall Reflections . . . 30

6 Appendix 31
6.1 Theatre Lighting Term Definitions 31
6.2 Instruction Set 32
6.2.1 Instruction Formats 32
6.2.2 Opcodes 32
6.3 Screen Display Verilog Files 34
6.3.1 Binary to String Converter 34
6.3.2 Color Tracking 37
6.3.3 String Display 39
6.3.4 String Display with Color Tracking 40
6.3.5 Dynamic Sprites 42
6.3.6 Static Sprites 49
6.3.7 XVGA 52
6.4 Keyboard Handler Verilog Files 53
6.4.1 Keyboard Debounce 53
6.4.2 PS2 to ASCII 55
6.4.3 Keyboard Interpreter 57
6.4.4 Keyboard Interpreter Test Bench 63
6.5 Processor Verilog Files 67
6.5.1 Processor Module 67
6.5.2 Control Logic 70
6.5.3 ALU 71
6.5.4 Right Shift 72
6.6 DMX Controller Verilog Files 73
6.6.1 Channel Twiddler 73
6.6.2 DMX output 75
6.6.3 Divider 77
6.6.4 DMX controller 78
6.6.5 Timer 80
6.7 Labkit Verilog Files 81
6.7.1 Debounce 81
6.7.2 Follow TImer 82
6.7.3 Labkit 83

List of Figures

1 Module Diagram 6
2 Example Screen Display 8
3 Example of Color Tracking 9
4 Example Error Message Display 10
5 Blind Mode 11
6 Live Mode 11
7 Keyboard Interpreter State Diagram 13
8 Processor Block Diagram 17
9 Basic Cue Data 19
10 Extended Cue Data 19
11 Channel Data 20
12 Load Cue Timing Diagram 21
13 DMX-512 timing diagram 22
14 Recording a cue 24
15 ADDC instruction 26

16 ADD Instruction 26
17 A DMX-512 packet 28

List of Tables

1 Color Values 10
2 Key Assignments 12
3 Error Messages 14
4 Reserved Registers in the Processor 19
5 DMX-512 packet specifications 23
6 Control Logic Test Cases 25
7 ALU Test Cases 26
8 Lighting Board Test Cases 27
9 More Lighting Board Test Cases 28
10 Instruction Set 33
11 Opcode Table 34

1 Introduction

This document details the design of a lighting board console. Lighting boards are used mainly in theatre and entertainment settings to program the lighting for an event. A lighting board allows the user to manipulate channel intensities to produce different looks and effects with the lighting instruments. These looks can then be made into a cue and saved in the lighting board's memory. The lighting design for a theatre production is composed of multiple preprogrammed cues that are run during the show from the board's memory. During a show, the lights transition between cue to cue either manually or automatically on a timer. Most cues are run manually by pressing the go button, which tells the lighting board to go to the next cue in the sequence. If the cue transition happens automatically, there is no need to press the go button to transition into the next cue.

2 Overview

The lighting board described in this document is similar to ones used in theatres around the world. The board can control eight channels on two dimmer boxes. These channels control the intensity of lights that are be plugged into the dimmer boxes. Each channel can control up to two lights. The dimmers are controlled by the labkit through the use of DMX-512 protocol. DMX-512 is the industry standard for communication between dimmers and the lighting board. Our lighting board stores up to 127 cues. Each cue has its own special parameters such as an up/down time, follow, link, and wait. Our lighting board utilizes a computer screen and keyboard for programming cues instead of the standard lighting board interface. The keyboard has special key bindings that implement standard lighting board buttons. The special key binding allow us to implement all of the functionalities of a real lighting board without the specialized interface on real lighting boards.

2.1 Board Specifications

The lighting board is used both to design cues and to program a show. The board controls 8 channels, which can be set to any percentage value between 0% and 100%. Any channel that is changed becomes captured until the user saves and releases the channel. That way the lighting technician does not have to worry about losing channel values before they are saved. An intensity value for each of the 8 channels can be stored for a cue along with uptime, downtime, wait time, follow time and link. The downtime and uptime dictate the period over which the lights gradually change. This gradual change can be seen on stage as well as on the screen. Wait time delays the uptime of the cue that is being loaded. Follow automatically loads the next cue after the given period of time. Link indicates the number of the following cue. The default numerically sequential but cues can also be linked out of order and in loops. The board can operate in blind mode, where the user can play with different cues but not alter the look on stage. The board stores the cue that was current when entering blind mode, so that it can return there when going back into live mode.

2.2 Screen

The user will use the screen to view the different parts of the cue as they are programming. The screen will display the current cue number and the intensity levels of the 8 channels in that cue. The channels values will be color coded to indicate whether the channel was used in the previous cue, at either the same or a different level. Also, the channel value will display a different color if the channel has been captured. The screen will show all of the other parameters for the current cue: uptime, downtime, wait time, link and follow time. The screen will contain the previous and next sequential cue numbers and the basic data, uptime, downtime and wait time, for those cues. At the top the screen shows the current mode, either live or blind. Since many parameters are set by a sequence of keys, the screen will show prompts to help the user and error messages if an incorrect key has been hit. All of these functionalities are implemented by the Screen Display.

2.3 Keyboard

The keyboard uses 28 keys to replicate the buttons on a lighting board. The goal is the make the functionality of the keyboard as similar to a commercial lighting board as possible. Many buttons are directly mapped to a selected button on the keyboard such as 'go','channel','at'. All the key mappings are listed in the Keyboard Handler section, which describes the implementation of the keyboard. In general to program a value, keys are needed to disambiguate the numerical values. This makes some keys invalid once certain keys are hit. This functionality is implemented with a large finite state machine. At the end of the sequence an instruction is issued to the Processor, which performs the operations needed to save or set the appropriate parameters.

3 Design

This section discusses the design of the lighting board. The board consists of 4 major subsystems: Screen Display, Keyboard Handler, Processor and DMX Controller. The System Overview will describe how the four subsystems integrate together and then each subsystem will be discussed in more detail individually.

3.1 System Overview

We are using a shared memory, processor-based implementation for the lighting board controller. Most of the functionalities will be handled as instructions in the processor. The modules for the lighting board controller fall into four subsystems: Keyboard Handler, Screen Display, Processor and DMX controller. All the modules in each subsystem are shown in Figure 1 except for the Processor, which is detailed in another diagram in the Processor section.

Every time a user hits a key, the Keyboard Handler decides what action needs to be performed. On most key strokes the Keyboard Handler will send an instructions to the Processor. The Processor uses the instruction to coordinate the rest of the response to the user input. The Processor also contains the cue memory, so it will store any paramters that have been entered. The Screen Display will show the new value and the user can continue programming the board.

3.2 Screen Display by Maura

The screen is composed of three subsystems: the static sprite module, the dynamic sprite module, and the XVGA module. The XVGA module takes the pixel data from the dynamic and static sprite modules and displays this data onto the screen. The screen is composed of fifty-two sprites, thirty-two which change depending on the state the lighting board is operating in and the current cue. The other twenty static sprites are headings describing the dynamic data.

3.2.1 XVGA Module (xvga.v)

The purpose of this module is to display the sprite pixels to the screen. This module is a slight modification of the XVGA module that is available online via the fall 2005 website. This module has been modified for a screen resolution of 800 x 600 pixels. By reducing the resolution of the screen, the module was able to be clocked off of a 40mhz clock, instead of the 65mhz clock. The slower clock allowed the module more time to complete all of the necessary logic to display the sprites. The extra time was needed due to the color tracking logic of the sprites and to overcome the propagation delay of the combinational logic being used to display the sprites. The reduced screen resolution caused the values of the horizontal and vertical syncs and blanks to change from the original version of this module so that the screen would actually display the proper pixels. The correct values were calculated and inserted into the module for it to be fully functional in the 800x600 pixel resolution.

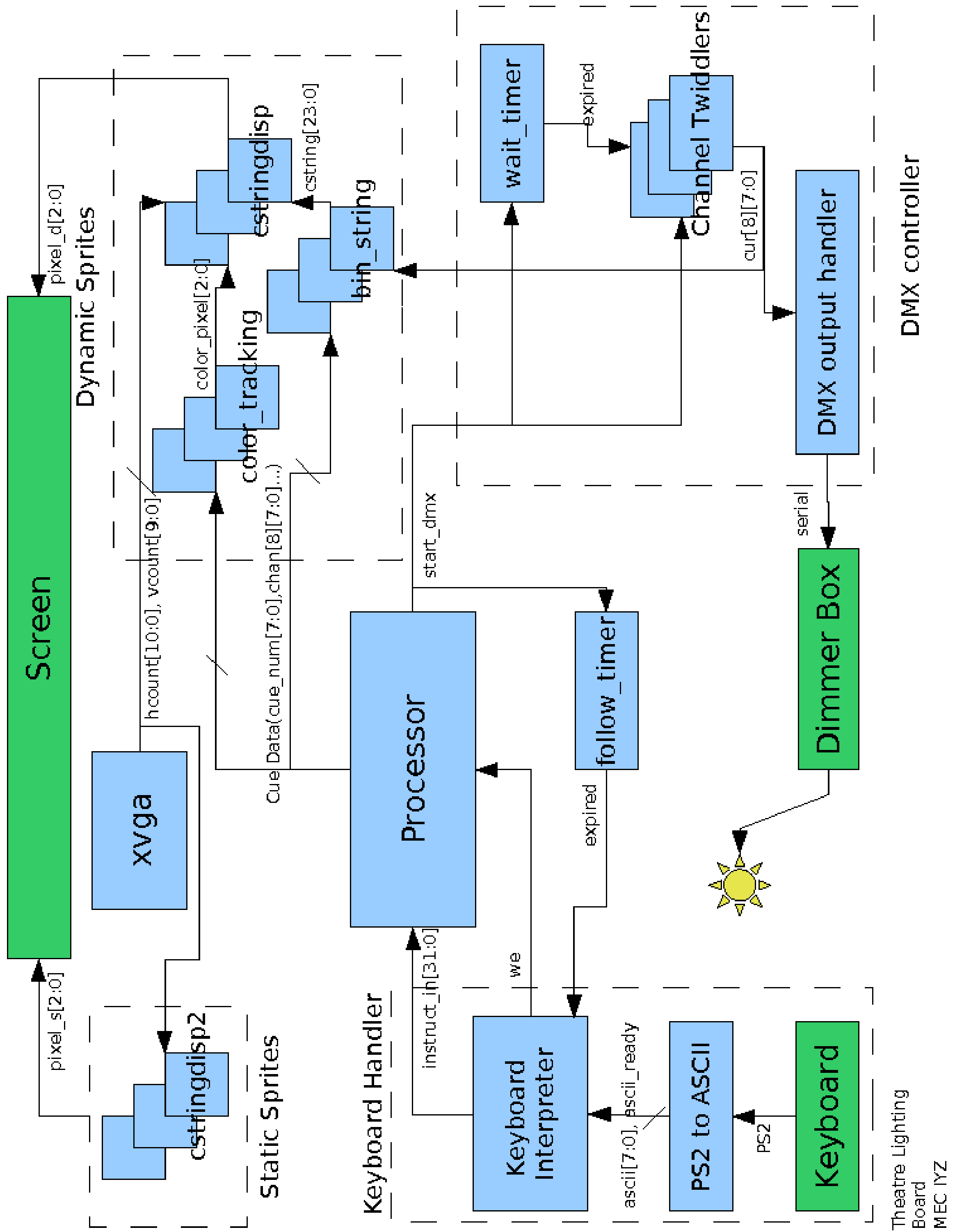

Figure 1: Module Diagram

3.2.2 Blob Module (blobv.v and blobh.v)

This module is used to divide the screen into different visual areas by creating lines. This module is based off of the blob module from Lab 4. The original module is modified into two modules, one to create a horizontal line across the length of the screen and another to create a vertical line down the bottom half of the screen. The modified blob modules are simplified and use the hcount and vcount position to determine the color of the pixel at each hcount and vcount position. The width and height parameters were removed from the module since the module was used to create a line across the screen and not a large blob.

3.2.3 Character String Display Module (cstringdisp.v and cstringdisp2.v)

The character string display module is used to display a character string on the screen through the use of the font rom module. This module is based off of the version of the character string display module found on the fall 2005 class website. This project uses two versions of this module. The first version is used exactly as it was originally written on the website, while the second version modifies the module to account for different font colors.

The basic functionality of the module is that it takes each character in the character string and looks up that character in the font rom module. Each ascii character is 8 bits and 12 bytes. The pixels for each character are assigned either a one or zero depending on whether or not the character is in the right hcount and vcount area. If the position is correct, then it is assigned a one, otherwise it is blank (zero). This process is done for each character in the string until all characters have been assigned a pixel value. This pixel data is then outputted to the appropriate screen module to be displayed on the screen.

The modification of this module, version 2, operates in the same way as the original module, but the color of the pixel can be altered depending on the value of the color_pixel variable that is inputted into the module. This color_pixel value facilitates color tracking of the channel intensity sprites as well as highlighting the sprite that being altered. The color of the character string is changed by altering the cpixel assignment. The default cpixel value is 7 for each character which will display the character white. By altering the cpixel value, the string is able to change colors depending on the value of color_pixel. The value of cpixel can now be either, 2, 3, 4, 5, 6, 7, which corresponds to green, blue, red, magenta, yellow, and white. Each color has a special meaning, which will be discussed in the color tracking module.

3.2.4 Static Sprites Module (static_sp.v)

This module creates the headings for all the static sprites on the screen. The headings do not change in our project and are therefore separate from the dynamic sprites. As seen from the screen shot in Figure 2, all of the headings are neatly arranged in the screen and allows the user an easy way to read off the appropriate data from the screen.

The static sprite module displays 22 sprites. Out of theses 22, 20 of the sprites are character strings. These character strings are headings that are assigned a static value in the module. These sprites correspond to the Mode, Channel label at the top of the screen, the channel headings, and the headings for the two boxes in the bottom half of the screen. The module calls upon the character string display and the blob module to obtain the static pixel data the module needs to output to the XVGA module.

3.2.5 Binary to String Converter (bin_string.v)

The purpose of this module was to convert an 8-bit binary number into a string of three characters. Each character is looked up separately and then all three single characters are concatenated together before outputting the character string to the dynamic sprites module.

Each character string is broken down into a hundreds place, tens place, and an ones place to represent the corresponding digit. The character for each digit placement is obtained through a series of if-statements. To obtain the character that corresponds to the correct value of the binary number, the module subtracts the value 100 from the binary number and then tests to see if the resulting value is greater than 100, 0, or neither. If the value is greater than 100, the module knows that the hundreds character digit must be a 2,

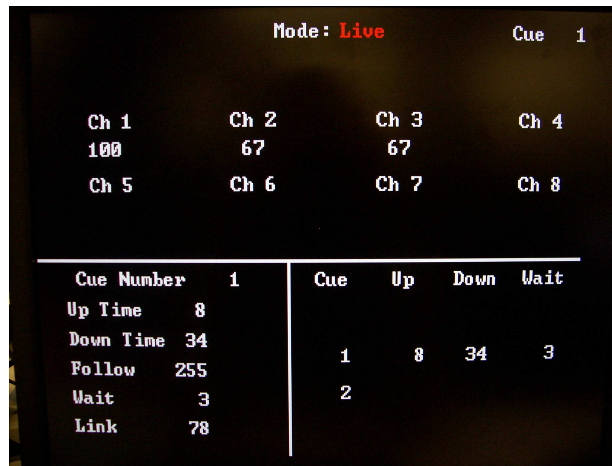

Figure 2: Example Screen Display

and then correspondingly if the resultant was greater than 0, the hundreds character digit was a one, or the number was less than 100 and therefore the hundreds character will be empty. The resultant value is then stored in the signed register, temp_num. This number will be used to calculate the tens character digit. The tests to obtain the hundreds character digit are limited to only two if-statements since the upper bound of any number that will be displayed will not be any greater than 256. There are error tests in the keyboard interpreter module to catch any data that is an invalid entry.

The tens character digit is obtained in a manner similar to that of the hundreds character digit. There are a series of ten if-statements that are evaluated only when the value of temp_num has changed. The first if-statement subtracts 90 from temp_num and if it is greater than 0, the tens character digit must be a "9", otherwise the next if-statement evaluates. This statement subtracts 80 from temp_num and compares that value to 0, if true then the tens character digit is an "8", otherwise the next if-statement evaluates. This series of comparisons continues until the statement is true. The last if-statement is a bit different from the first nine in this block. The last if-statement is evaluated if the character digit for the tens place is less than 10, which would make the character digit a 0. The last if-statement tests the value of the hundreds character digit. If the character digit was " " (so the number was less than 100, and now less than 10), the tens character digit will also be a blank character digit. This prevents a number from being displayed as 001 (or another variation), instead it will be displayed as a 1. The first two character digits are empty. If the hundreds character digit is not blank, then the tens character will be a 0. Similar to the hundreds digit, the resultant value from the subtraction is stored in another register, temp_num2, to calculate the final character in the string, the ones character digit.

The ones character digit is obtained in a manner very similar to the tens digit. It also is a series of ten if-statements that are evaluated only when temp_num2 changes. The if-statements for the ones character digit are designed so that when you subtract a number if temp_num2 is equal to zero, the ones character digit is assigned to the value that you subtracted. This means that when 6 was subtracted from temp_num2 and it was equal to 0, the ones character digit is assigned "6". The ones character digit also has the same zero testing as the tens character digit, though the test in the ones block tests the character of the tens character digit.

After each block has been evaluated, the three character digits are then combined together in a 3 character string and outputted to the dynamic display module. This module is called for each of the dynamic sprites to accurately display the binary number as a character string on the screen.

3.2.6 Color tracking Modules (color_tracking.v and color_tracking2.v)

The purpose of the color tracking modules is to change the font of the character display depending on the state of the sprite and the current past history of the data for the sprite. The color of the character strings is used as a way for the user to quickly tell a few key characteristics about the previous and current cue. The color tracking modularity is broken down into two color tracking modules, one for channel intensity data and the other for the extended data of the current cue.

The color tracking module used for tracking the channel intensity data is more complicated than the extended data module. This module uses the current state, channel, release_flag, the current value of the channel, and the previous value of the channel. The color of the character string is determined after a series of tests concerning this data are evaluated. This module is evaluated eight times, once for each channel. This is important to note when testing to see if a channel intensity is currently being selected, the value of the parameter channel_test is compared to the channel value passed into this module. This makes sure that the channel that you are editing is the only channel that gets highlighted, otherwise it is possible for other channels to be selected if their intensity value is the same.

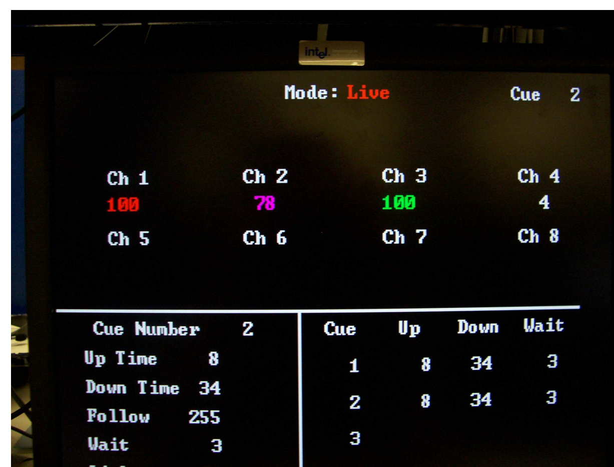

Figure 3: Example of Color Tracking

The screen capture above illustrates each color option that the lighting board can handle. To determine the color of the data multiple tests are done upon each set of data. The first test is to see if a channel is captured. If the channel is captured, then it will be colored green. If the channel is not captured, the next test evaluates if the channel state is equal to channel test and if you are in the AT state of the keyboard interpreter. This means that you are currently editing the value of the channel and the color of the character string should be yellow. If this test is false, then the current value of the channel is compared to the previous value of the channel. If the two values are the same, then the character display is assigned the color red. Otherwise if the two values are different and previous channel is not equal to zero, the value of the character string will be magenta. If the value of the previous channel data was zero, then the character string will be assigned the color white. This color data is stored in the color_pixel register and outputted to the character string display module which actually implements the color assignment to the character string.

The color tracking module for the extended data only deals with the current data that you are editing. It does not track the history of the data, like the other module does. This variation of the module turns the character string yellow if the user is currently editing the data, otherwise the character string is white. The color is assigned based off of the comparision between the state and the state test value. The state value tells the module which state the user is currently in from the keyboard module. If this state corresponds to one of the states of the extended data, then the user is currently editing the data for that corresponding extended data value; otherwise the color_pixel is set to a value that corresponds to white.

Table 1: Color Values

Color Code	Color	Meaning
0	Red	Same level in previous cue and Live Mode
1	Magenta	Different level in previous cue
2	Yellow	Currently Editing
3	White	Default
4	Blue	Error Messages and Blind Mode
5	Green	Captured

3.2.7 Dynamic Sprites Module (dynamic_sp.v)

The purpose of the dynamic sprites module is to obtain and display all of the data that will be changing through out the project and displaying the current value onto the screen. This module is able to achieve this through the character string display module, the binary to string module, both color tracking modules, and logic included inside the module itself. The dynamic sprites module calls each of its subsystem modules multiple times depending on the characteristics of each sprite.

The dynamic sprites module has to complete a three-input mux for each of the channels to determine which value the module needs to calculate and then display to the screen. This mux first tests to see if you are currently editing the value of the channel, and if so you display the param value for that channel, otherwise it tests to see if you are loading a cue. If you are loading a cue, the current value of the channel will be assigned to the temporary register value for that channel. If the dmx module is currently not loading a cue, the value displayed will be the value that is stored in the channel register. The outcome of the mux is stored in a temporary register for that channel and then that temporary register undergoes the binary to string conversion and then the color tracking module.

The dynamic sprites module undertakes a similar process for the extended data. The value of the temporary register for the extended data is assigned from the outcome of a two-input mux. If the user is editing the value of the extended data, then the dynamic sprites module will use the value of param for all of its display calculations, otherwise it uses the value stored in the register for the appropriate extended data for the current cue.

The next set of logic that the dynamic sprite module must do is to assign the error messages and to display the mode that the board is operating under. An example of an error message is seen in Figure 4.

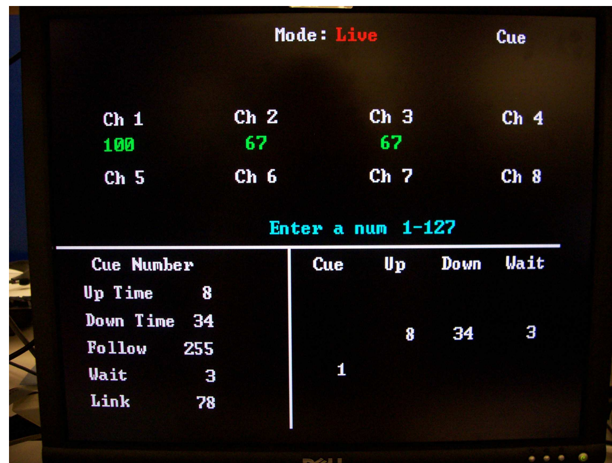

Figure 4: Example Error Message Display

The error messages are assigned through a case statement. The error messages that should be displayed

are sent to the dynamic sprites module in a 3-bit flag, so the dynamic sprites module looks up the correct messages to displays and then sends the correct character strings to the character string display module to output them to the screen. The mode of the board is determined by a 1-bit flag. If the flag is a zero, the lighting board is operating in live mode and the "Live" character string and color_pixel are displayed to the screen, otherwise the character string is assigned to "Blind" and the color_pixel for that character string. The screen captures below illustrates how the screen looks in each mode.

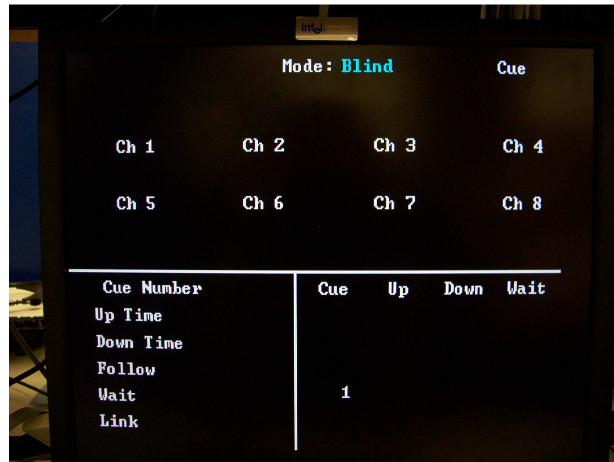

Figure 5: Blind Mode

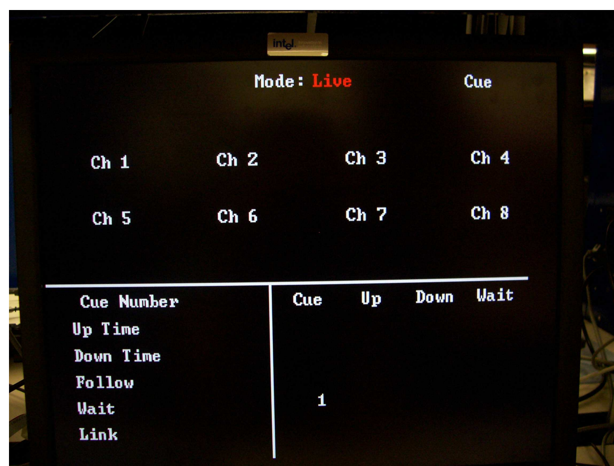

Figure 6: Live Mode

3.3 Keyboard Handler by Maura

The Keyboard handing subsystem is composed of two modules: the ps2 to ascii input module and the keyboard interpreter module. The ps2 to ascii module was obtained from the fall 2005 class website, while the keyboard interpreter module is unique to this project. The purpose of this subsystem is to allow the user to input data that will correspond to actions in the lighting board. This input is converted into an ascii value and interpreted by the keyboard interpreter module. This module allows the user to program the lighting board and execute those cues.

3.3.1 PS2 to ascii Module (ps2_ascii_input.v)

This module was obtained from the fall 2005 class website. There were no modifications to the file. This module is used to output the ascii value that corresponds to a keystroke. This ascii value is then used in the keyboard interpreter to complete the proper function. The key bindings have changed for the purposes of this project. A list of the purpose of each key is listed in the appendix.

3.3.2 Keyboard Interpreter Module (keyboard_interp.v)

The keyboard module translates the ascii input from the user into meaningful instructions to the processor. The keyboard interpreter is a large finite state machine that properly handles the functionality of the lighting board. This module is integral in issuing the proper instructions to the processor for the lighting board to function in the manner in which the user intended by their keystroke pattern.

As seen in the finite state diagram of the keyboard interpreter module (see Figure 7), the module contains ten states. You enter each of the ten states by entering in a specific ascii value which corresponds to the key binding of several of the keys used on a normal keyboard. The meaning of the key has changed for this project, but the ascii value for each key has not. The ascii value of the key is inputted into the interpreter from the ps2 to ascii input module, along with an ascii ready signal. This signal is only high when a key has been entered. A list of the ascii values along with their normal assignment and lighting board assignment is included in the table below.

Table 2: Key Assignments

ASCII (in hex)	Key	Button Name
41	A	At
42	B	Blind
43	C	Cue
44	D	Downtime
45	E	Clear Entry
46	F	Follow
47	G	Go
48	H	Channel
4A	J	Release
4C	L	Link
4F	O	Full Intensity (100%)
51	Q	Clears the cue
52	R	Record
53	S	Live Mode
55	U	Uptime
5A	Z	Reset
0D	Enter	Enter
08	Backspace	Previous Cue
30-39	0-9	0-9

Before discussing the individual states of the keyboard interpreter, a few key elements used throughout the finite state machine must first be examined. The main element is the functional parameter register, register 16. This register holds all numerical (value) data that the user inputs, which must then be sent to the processor at the correct stage in the finite state machine. The value of the functional parameter register is updated whenever the keys 0-9 are inputted by the user. This value gets stored in the temporary value register which will be used in a few of the instructions sent to the processor. The temporary value register gets updated as stated before whenever a number is entered by the user. The correct value gets stored in the temporary value register by multiplying the value of parameter by 10 and adding the number that was

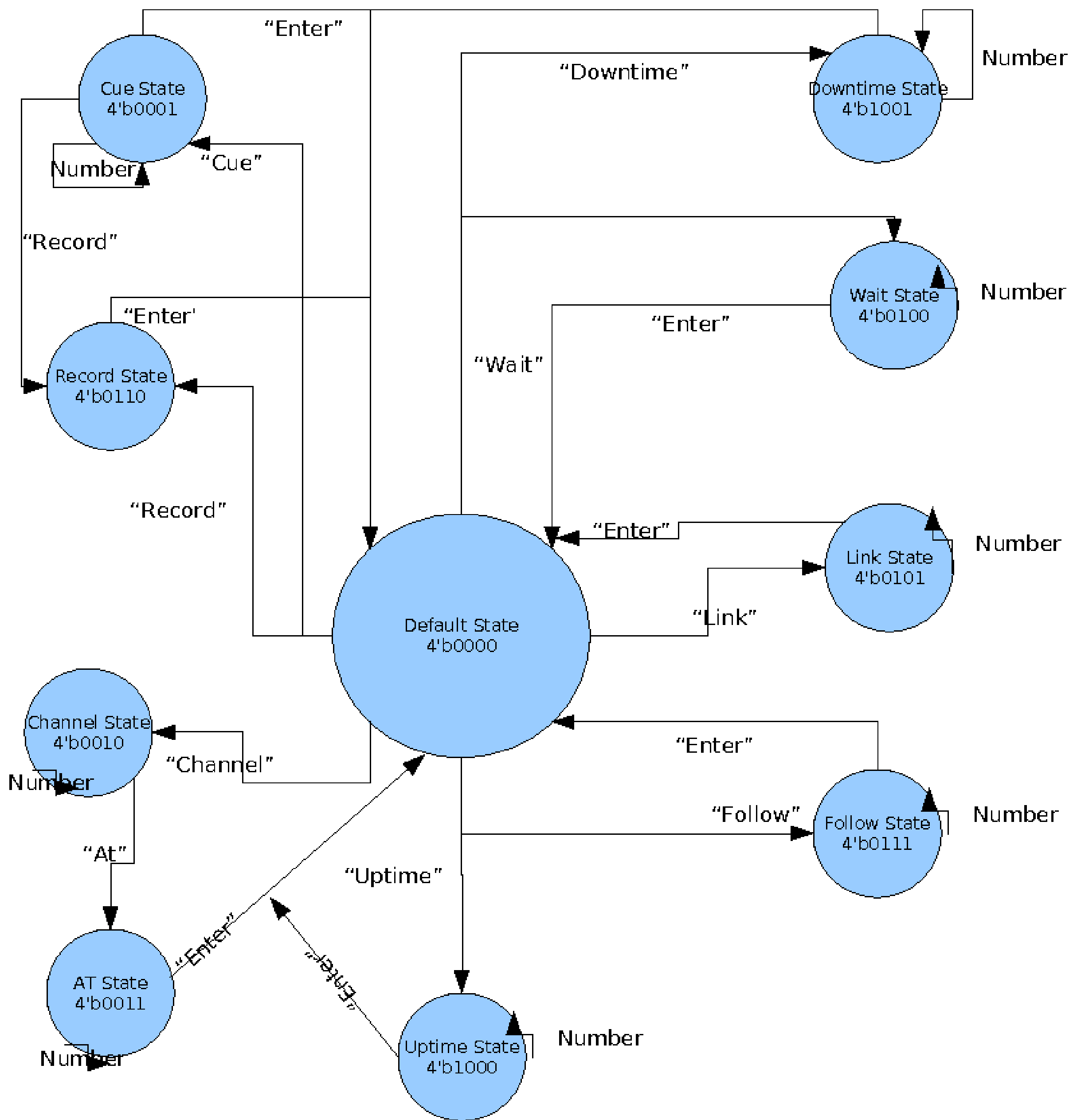

Figure 7: Keyboard Interpreter State Diagram

just entered. The functional parameter register will store this temporary value in its register, so that when a third keystroke is entered, the functional parameter already has the previous two keystrokes accounted for and the proper value can be outputted to the instruction buffer.

The keyboard module communicates to the processor through an instruction set and macros. The different between an instruction and a macro is simply that a macro is a series of instructions that need to happen at the same time. The keyboard module can only output one instruction at a time, so macros are necessary for certain functions such as loading a cue, recording a cue, releasing captured channels, setting the intensity of a channel, setting the up, down, wait, follow time of a cue, setting the link of a cue, aborting the current operation, transitioning into blind mode, and transitioning into live mode. All of these macros are explained in depth in the macro section. Each macro and instruction is sent to the processor only when write enable is high. This ensures that an instruction is not accidentally sent. The keyboard interpreter module makes sure that only the correct instructions are sent based on the input from the user at the correct point in the finite state machine. Each of the assigned keys sends the corresponding instruction to the instruction buffer.

In addition to sending out an instruction on most keystrokes, an error message flag is sent to the dynamic sprites module to display helpful messages to the user at certain states. The messages and the corresponding flags are listed in Table 3. These error messages alert the user if they have input data that is not in the valid range for the type of functionality they are trying to implement. The error message is sent out after a check on the functional parameter is done at each state if that state depends on the value of parameter for a valid instruction to be sent to the instruction buffer.

Table 3: Error Messages

Error Code	**Message**	**Use**
0	Enter a num 1-127	Displayed for cue selection and when entering a cue to link to
1	Enter a num 1-8	Displayed for channel selection
2	Enter a num 0-100	Displayed if an incorrect channel intensity has been entered
3	Enter a num 0-255	Displayed for entering uptime, downtime and wait time
4	Enter a num 1-255	Displayed for entering follow time
5	Enter to confirm	Displayed after record has been hit
6	Enter intensity	Displayed after 'at' has been hit

The lighting board has a special functionality that will allow the lighting cues to automatically follow each other if the user so desires. When this happens, the follower timer sends the signal, expired, high for a clock cycle to alert the keyboard module that the Go macro needs to be issued. Once this instruction has been sent, the module goes back into the default state.

As stated before, the keyboard interpreter is composed of ten states. The first state is also the default state. In this state it is possible to access eight of the other nine states. The transition to another state happens simply on the appropriate keystroke. To enter into the Cue State (4'b0001) you would enter the key that corresponds to 'Cue', which is the letter C. The user can enter the other seven states (Channel, Record, Up time, Down time, Wait, Follow, Link) in a similar fashion. The keys needed to be inputted by the user to enter each state is included in Figure 7. When the state machine transitions into one of these eight states, the appropriate error message is displayed to remind the user the type of valid data that each state can take as input.

In addition to being able to transition into the previously named eight states, the default state also allows the user the functionality of seven other keys. These keys are reset, blind mode, live mode, release, clearing the current cue, loading a previous cue, and loading a cue through the use of the go button. These seven buttons output an instruction whenever they are entered by the user. The only buttons that are usable in the other states are the reset button and the go functionality is used in the Cue State. If the user accidentally presses a key that is not usable in the default state (a key that doesn't correspond to the eight transitioning states or the seven special key inputs) the finite state machine will default to itself.

The cue state is entered whenever the cue button is pressed from the default state. In the cue state, the user is allowed to enter in a cue number 1-127 and then edit that cue through certain characteristics. The

functionalities of this state is that you can enter the Record state if you press the record button, you can load a cue to the stage and screen by pressing enter, you can select a specific cue if you type in the cue number and hten you can load the cue in show mode if you press go. There is a valid entry check when you enter in a cue number to make sure that it is between 1 and 127. If the number the user entered does not fit inside this range, an error message will be displayed, the functional parameter register will be cleared, and the user will be promoted to enter in a valid number. If a key is entered that is not assigned a function in this state, the state defaults back to the default state.

The next state to be examined is the channel state. This state allows the user to select one of the eight channels at a time. If the user tries to select an invalid channel number, an error message appears, the functional parameter register is cleared, and the user is prompted to input a valid channel number. The channel state will then logically transition to the At state whenever the At key is entered, which will allow the user to set an intensity level for that channel. The other keystroke that is valid in this state is the reset key.

In the At state the user enters in an intensity level for the channel they chose in the channel state. The user is allowed to enter in a value up to 100%. If the value of the functional parameter is greater than 100, an error message is sent and the user is prompted to enter in a valid intensity level. The other functionalities of this state are that you can use the full button to automatically set the intensity of the channel to 100%, instead of entering in 100 with three keystrokes. The channel intensity is highlighted yellow while the channel is being edited and as soon as the user presses enter to confirm the intensity level the Channel Intensity should turn green to illustrate that the channel is captured. As noted above, captured channels can be released when Release is entered. The user is only allowed to enter this state through selecting a channel number. This is the one state that the user is not allowed to enter through the default state since it only is used to assign a value to a specific channel.

After channel intensities have been recorded, these changes should be saved in a cue if the user so desires. There are two ways to record a cue. If the user wants to record the current cue, the user can press record enter to save the current cue. If the user desires to save the current channel intensities in a cue that is different from the current cue number, the user has to press Cue, number (between 1-127), record, and enter to save the cue to that cue number. If the user enters a number that is out of the valid range the screen will display and error message and the user is prompted to enter in a valid number. If a key is entered that is not enter or reset, the module resets the registers and defaults to the record state.

The user is allowed to link a cue to another cue that is not sequentially the next cue in the sequence. The user is allowed to link any cue with any other valid cue number. The link functionality is defaulted to automatically link to the next sequential cue, unless it is assigned to link to a different cue. The link state checks the data saved in the function parameter to check the validity of the data inputted by the user. If the number entered is not between 1-127 an error message is displayed and the user is prompted to enter in a valid number. Once a valid number has been entered the state moves to default state; otherwise the state keeps asking the user for valid input data. The user can choose to abort any state at any time by pressing reset.

The rest of the states in the keyboard interpreter module allow the user to set the timing parameters of cues. Each cue has a wait, follow, up, and down time that is individually associated to each cue. The user is allowed to set the value of each of the timers and save those changes for each cue. Each state checks to make sure that the number entered in is in the valid range for the state the user is operating in. The valid data ranges for up time, down time, and wait are 0-255 seconds and the follow time valid data range is 1-255. The user is not allowed to set a 0 follow time, since 0 is used as a design test to make sure that the cue is not set to auto-follow to the next cue. Each state has a reset, enter, and a clear entry key function. Any other key functionality will cause the state to reset back to itself.

Each state in the keyboard module contributes to the overall functionality of the lighting board. This module allows the user to manipulate the board in the manner in which is most useful to their purposes. The keyboard module is the module that bridges the user to the functionality of the board.

3.4 Processor by Irene (processor.v)

The Processor coordinates interactions between the other three modules in the lighting board. The registers in the processor also serve to store information about the current state of the lighting board, such as the current cue number, the current channel intensities and other data. This data is placed in reserved registers, which are then wired out to the Screen Display and DMX controller. On specific keystrokes, the Keyboard Handler places an instruction into the Instruction Buffer, so that the processor can coordinate the appropriate response to the user input. Some of these instructions may call a macro, a prewritten series of instructions, that handle the more complicated operations such as loading a cue. In addition to coordinating interactions between the other modules, the processor is also responsible for retrieving and storing cues because the processor is the only module with access to the Cue Memory. The processor module itself also contains 2 other modules, the Control Logic module and the Arithmetic Logic Unit, which implement some specific parts of the processor.

The processor used for the lighting board is a very general, unpipelined 32-bit processor. The instruction set and architecture are based on the 6.004 β. The instruction set is documented in §6.2 of the Appendix. The processor can perform all arithmetic operation except multiply and divide, comparisons and branches. Multiply and divide were removed to increase the speed of the processor and because they are not used for any lighting board operations. The processor was clocked at 20Mhz, half of the screen refresh rate. The processor could not be clocked at the screen refresh rate of 40Mhz because that did not allow enough time for the logic to finish before the data was written to memory and to register. The speed of the processor is not especially important for the lighting board because the processor still runs much faster than a user would notice.

3.4.1 Instruction Buffer

The Instruction Buffer is a 32-bit x 64-bit FIFO buffer. When the Keyboard Handler receives a keystroke that requires the processor to handle, the keyboard interpreter module sends the instruction to the instruction buffer and issues a write enable. If the processor is not working on a macro and the instruction buffer is not empty, the processor will issue a read enable to get the instruction. Since instructions are only issued by key strokes, there is little chance of the instruction buffer filling up because the processor run much faster than a person can type.

3.4.2 Control Logic (ctrl_logic.v)

The Control Logic module is responsible for setting the control signals given an instruction opcode, the mode of the processor, and z, the signal indicating whether the value coming out of the first register file port is 0. The module is entirely combinational logic. The signals are mostly for the processor, but also include the start and update signal for the DMX controller. The signals are:

- mpcsel – address selection for the macros ROM, previous+1 by default, it changes to the macro or branch address for the macro and branch instructions
- bre – instruction buffer read enable
- bsel – selects between a register value or the sign extended constant for the second argument in arithmetic operations
- ra2sel – selects the register number for the second instruction, usually bits 15-11 of the instruction, but changed to bits 25-21 for a store instruction
- wdsel – selects the data going into the register file, usually the data coming out of the ALU, except for a load instruction, the data is from the Cue Memory
- alufn – a 4-bit operator selector for the ALU

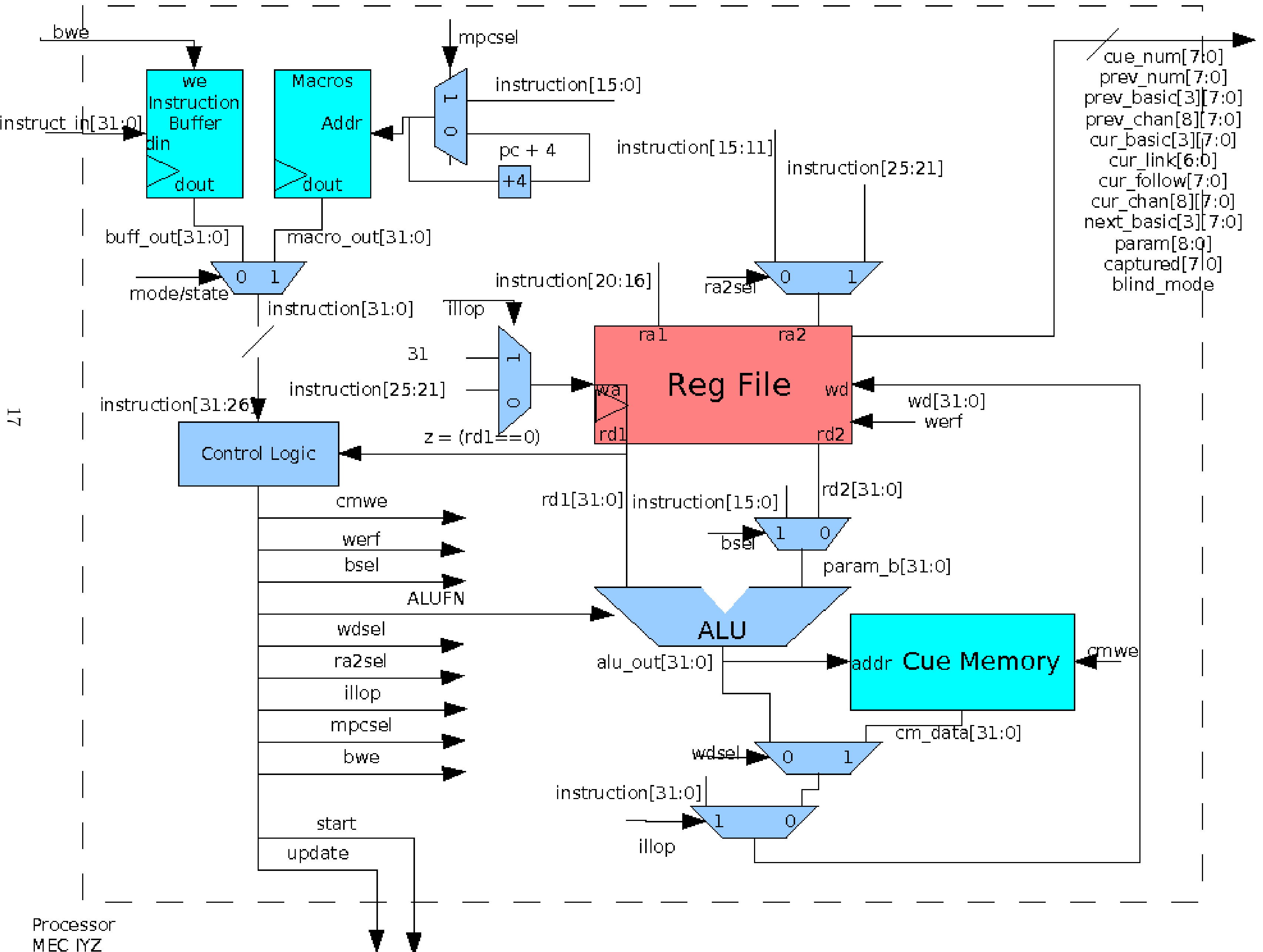

Figure 8: Processor Block Diagram

- werf – write enable register file, writes data to a register on the rising edge of the clock
- cmwe – cue memory write enable, writes data to cue memory on the next rising edge of the clock
- start – starts the loading of a cue in the DMX controller. Also starts the follow and wait timers
- update – updates the channel intensity on stage to reflect the registers.

The two write enable signals are only set when the module detects that there is a valid instruction, otherwise the signals are always 0 and the processor is idling. The valid instruction flag comes from the instruction buffer when not executing a macro, otherwise it is always 1. This is so that the processor does not execute 1 instruction from the instruction buffer over and over.

3.4.3 Arithmetic Logic Unit (alu.v)

The ALU performs arithmetic operations on two operands. The ALU also receives a 4-bit signal that selects the operation to be performed. The operations are:

- addition
- bit-wise and
- bit-wise or
- left shift
- right shift
- signed right shift
- subtraction
- bit-wise xor
- equal to
- less than
- less than equal to

The right shift is done by a module taken from the β written by Prof. Terman. The right shift has to be done carefully to ensure that the highest bit or 0 is used for the shift depending on whether a signed or unsigned right shift is needed. The output of the module is the result of the operation.

3.4.4 Registers

There are 32 32-bit registers in the processor. In addition to being used as temporary storage for calculations, the registers of the processor were also used to store information to be shared with the other module. For example, certain registers are directly wired to the Screen Display. When instructions are issued changing the values in those registers, the screen automatically reflects the new values. The table below lists all of the reserved registers that are used. Registers 1-19 are all used by the Screen Display for displaying a current value or doing color tracking.

Register 16, the first macro parameter, is sent to the screen for display when the user is inputting a new value, so that each digit is shown as the user is typing it in. The current cue channel values, up time, down time and wait time, are used by the DMX controller to load cues and update what is on stage.

Table 4: Reserved Registers in the Processor

Use	Register Num.
Zero register	0
Previous/Current cue address	1
Previous cue basic data	2
Previous cue ch1-ch4	3
Previous cue ch5-ch8	4
Current cue basic data	5
Current cue extended data	6
Next cue basic data	7
Current cue ch1-ch8	8-15
Macro parameter 1	16
Macro parameter 2	17
Captured channels	18
blind mode	19
blind mode cue address	20
illop instruction	31

3.4.5 Cue Memory

The Cue memory is a 32-bit x 512-bit BRAM. The data for each cue takes up 4 lines. The higher 7 bits of the address are also the cue number, while the lower 2 bits address the lines of data for each cue. The first line is basic data about the cue such as up and down time and wait time. The second line holds extended data such as follow and link.

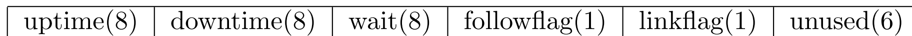

Figure 9: Basic Cue Data

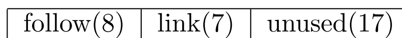

Figure 10: Extended Cue Data

The last 2 lines hold the channel intensities. The channel intensities range from 0-100, so only 7 bits are needed for each channel. The 4 channels are evenly spaced over 32-bits.

All values are defaulted to 0 and a link or follow of 0 is ignored. By default cue 0 is a special cue that is never changed. It is the first cue loaded when the lighting board is turned on and is used as a scratch cue.

The cue memory is clock on the falling edge of the clock as opposed to the rising, so that there is time for the data to come out of the memory before being written to register.

3.4.6 Macros ROM

The Macros ROM is a 32-bit x 256-bit ROM. It contains 244 instructions, for 13 macros. The macros are:

- Load cue – loads the cue in register 16 to screen. Issues the start signal to the DMX controller if the macro was called from the Go macro. Otherwise issues the update signal to the DMX controller if in live mode.
- Record – stores a cue to memory. Packs up all of the channels and saves them along with the basic and extended data. If register 16 is 0, record to the current cue address, otherwise record to the cue indicated in register 16.

Channel 1(7)	unused(1)	Channel 2(7)	unused(1)	Channel 3(7)	unused(1)	Channel 4(7)	unused(1)

Channel 5(7)	unused(1)	Channel 6(7)	unused(1)	Channel 7(7)	unused(1)	Channel 8(7)	unused(1)

Figure 11: Channel Data

- Release – Reverts all current channel intensity level back to what is stored in the cue memory. Sets all captured channel flags (register 18)to 0.
- Set channel – sets the channel in register 17 to the value in register 16. Updates the channel intensity on stage if in live mode. Set the captured channel flag for that channel. Does not store the channel value to memory.
- Set up – sets the uptime for the current cue to the value in register 16. Stores the value to cue memory if the current cue is not cue 0.
- Set down – sets the downtime for the current cue to the value in register 16. Stores the value to cue memory if the current cue is not cue 0.
- Set wait – sets the wait time for the current cue to the value in register 16. Stores the value to cue memory if the current cue is not cue 0.
- Set link – links the current cue to the cue in register 16. Stores the linked cue if the current cue is not cue 0.
- Set follow – sets the follow time for the current cue to the value in register 16. Stores the value to cue memory if the current cue is not cue 0.
- abort – clear out the two macro paramter registers, registers 16 and 17.
- Go – sets a flag to indicate a Go macro call and calls the load cue macro. If register 16 is 0, the linked cue number is placed into the parameter register (register 16) for the load cue macro. If the linked cue is 0, the next sequential cue is put into register 16.
- Blind – stores the current cue in the blind mode cue register (register 20). Sets the blind mode flag to 1.
- Live – loads the cue number in the blind mode cue register by placing it into register 16 and calling load cue.

Each macro does some number checking, but the bulk of the error checking is done in the Keyboard Handler, so that the appropriate error messages can be sent without bothering the processor.

3.5 DMX Controller by Irene (dmx.v)

The DMX controller sends data out to the dimmer boxes to bring the cues to stage. The controller receives 8 channel values from the registers in the processor as well as the uptime, downtime and wait time. The controller uses these values to set the intensity of the lights on stage. The DMX output module outputs the single bit that sends the serial data to the dimmer boxes to set light intensities.

The DMX controller contains a Channel Twiddler for each channel to manage the intensity of each channel. Update signals are just passed on to each channel twiddler module to handle. Start signals indicate that the controller needs to begin loading a cue. The start signal also starts the wait timer. The values coming from the registers on a start signal are treated as the new cue to be loaded. Each channel that is zero in the new cue needs to be brought down for the new cue. The period inputted into the channel twiddler is set as the downtime and the start signal is issued right away. The channels that are non-zero need to be brought up in the uptime after waiting for the wait time. The period for those channels is set to the uptime and the start signal is the set to the expired signal coming out of the wait timer.

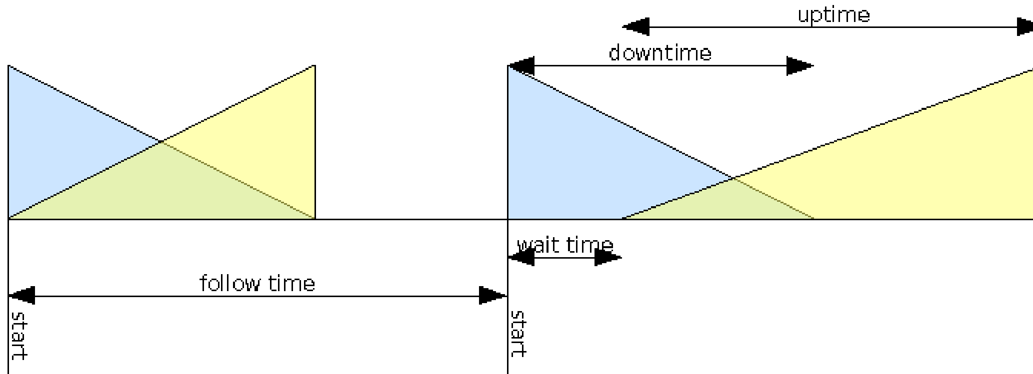

Figure 12: Load Cue Timing Diagram

3.5.1 Channel Twiddler Module (channeltwiddler.v)

The channel twiddler manages the intensity level of one channel. The module receives the intensity level as a 0-100 percentage from a register and converts to a 0-255 intensity level to send to the DMX output module. This conversion is approximated compactly by summing each 1 in the binary number weighted by its position times 2.55. In other words,

$$dmx_value = (percentage_value[6])?163 : 0 + (percentage_value[5])?82 : 0 + \cdots \quad (1)$$

On an update signal from the processor, the module stores the new value that is coming in from the value in the register and begins sending that to the DMX output module. On a start signal from the DMX controller, the module gradually changes the channel intensity from the current intensity level to the new level over the period supplied by the DMX controller. The gradual change is achieved by calculating the period between changes of 1 in the intensity level and using a timer to alert the module when it is time to increase or decrease the intensity level. The clock frequency is 20 mhz for this module so the equation to calculate the period of the timer was,

$$count_to = \frac{2025}{|current_value - new_value|} \cdot 10000 \cdot period \quad (2)$$

The module keeps a loading flag to keep track of when a gradual increase in intensity level is taking place. When the flag is high and expired is issued, the intensity level is increased or decreased by 1 depending on whether the end intensity is higher or lower than the beginning intensity level.

3.5.2 DMX output (dmxcontroller.v)

The DMX output module takes intensity values in from the Channel Twiddler module for each channel. Using these values, the module outputs the correct serial data at 250 khz to a user pin on the labkit. This user pin is then connected to a chip that converts the output from labkit to standard RS-485 differential output. The output from the chip is wired to a 5-pin XLR connecter that plugs into the first dimmer box. The second dimmer box is daisy-chained with a XLR cable to the first box. A switch is flipped on the second dimmer box indicating the addressing of the channels on that box starts at 4 instead of 0.

The serial output from the DMX output module follows the DMX protocol as detailed in the DMX-512 standard. Data is sent in packets consisting of slots. Each packet consists of a mark before break, break, mark after break and up to 513 slots. Each slot contains the intensity value of a channel, while the first slot in each packet is reserved for the start code. The null start code for dimmer class data is just 8 zeros. The timing diagram from the DMX-512 standard is included below.

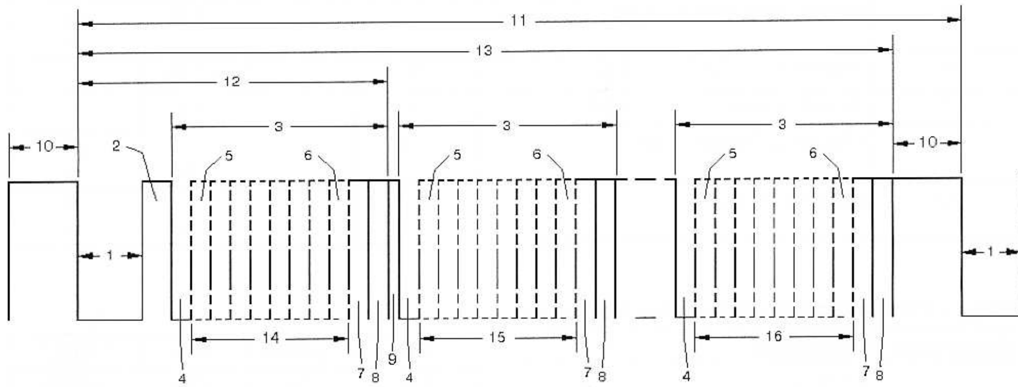

Figure Key
1 - “SPACE” for BREAK
2 - “MARK” After BREAK (MBA)
3 - Slot Time
4 - START Time
5 - LEAST SIGNIFICANT Data Bit
6 - MOST SIGNIFICANT Data Bit
7 - STOP Bit
8 - STOP Bit
9 - “MARK” Time Between Slots
10 - “MARK” Before BREAK (MBB)
11 - BREAK to BREAK Time
12 - RESET Sequence (BREAK, MAB, START Code)
13 - DMX512 Packet
14 - START CCODE (Slot 0 Data)
15 - SLOT 1 DATA
16 - SLOT nnn DATA (Maximum 512)

Figure 13: DMX-512 timing diagram

Since this lighting board only controls 8 channels, the DMX output module sends packets with only 9 slots, the null start code and 8 channel intensities. The minimum break-to-break time of 1204 μs,mandated by the DMX-512 standard, is met by padding the marks before and after each packet break and the marks after each slot. The table below contains the timing specifications for the packets sent by this lighting board.

Table 5: DMX-512 packet specifications

Description	value	# of bits	period
Mark before Break	1	52	208μs
Break	0	44	176μs
Mark after Break	1	52	208μs
Start bit	0	1	4μs
Channel Intensities	LSB to MSB	8	32μs
Mark after Data	1	11	44μs
Slot	–	20	80μs
Packet	–	332	1328μs

4 Testing

This section describes our testing strategy for the board. In general the module were individually tested in simulation, then integrated and tested as subsystems on the labkit. The final step was integration testing which verified that the lighting board met all the specifications listed in our proposal.

4.1 Screen Display Testing

The static sprites module was fairly easy to debug since I could clearly see what was wrong on the screen. A lot of the errors that I ran into when I first created the module were problems with the instantiation of character string display and not inputting the current parameters into each instantiation of the module. This was the first module that I built, so when I first displayed all of the headings to the screen I did not run into any screen glitches that came along later in the process as I add more combinational logic to the screen display. At this point in the process, the screen had a resolution of 1024x768 and all of the vertical and horizontal positioning was based off of this coordinate system.

After the dynamical sprites module was added, the screen became very glitchy and we tried a few methods to get rid of the glitches. One of the methods was to save the static data into a ROM. When the labkit was first reset, the static data would have been stored glitch free into the memory block and then after the first clock cycle, the data would only be read from the memory block. We thought that this would reduce the amount of sprites that were ored together for the screen display. This did not cut down on any time at all and we aborted this method to try and find a better way to get the screen to stop glitching.

The next thing that we tried was to reduce the resolution of the screen. This helped a lot, but it didnt fix all of our problems. We changed the resolution to 800x600 pixels so that the screen would be clocked on off of a 40mhz clock. There was just simply too many sprites and logic to be displayed in one clock cycle without pipelining the screen. So, we began to pipeline the screen. The screen is pipelined through 3 stages, which helped reduce the glitching except for a few parts in the screen. The parts that were still not perfect after pipelining was caused by illogical logic to obtain the error messages and the screen mode display. Once this logic was cleaned up, the screen display was perfect.

We did not try the screen at a resolution of 800x600 because that would shift the placement of all of the sprites yet again and we were not positive if the pipelining would have fixed the resolution problem at the higher resolution.

In the modules that made up the dynamic sprite module, the modules that I had errors was color tracking. The problem with this module was that it would select all of the sprites on the screen that had the same

value that the parameter was currently equal to. This problem was caused by not correctly isolating the data that the user was editing by checking the channel and state to a set value for each instantiation of the color tracking module. Once this logic was modified so that it tested both the test state and the state being modified in the keyboard interpreter state, the parameter would only highlight the sprite that the user was currently editing.

The binary to string module was at first a huge case statement where it would look up the binary number (0-255) and would then assign the character string to be the appropriate string representation of that binary number. This was highly inefficient, so the module underwent a radical change. The new design broke the binary number into three character digits and used three always blocks to find the value of the binary number. This was a much more elegant way of finding the character string since it involved only about 25 if statements in comparison to 256 case statements.

4.2 Keyboard Handler Testing

The keyboard handler was fairly easy to debug through the use of a test bench. After the module was created, I created a test bench for each state of the keyboard interpreter. There were a lot of errors at first, which were mainly caused by typos and not paying close enough attention to the code. Most of these bugs surfaced during the test bench, but a few were not apparent until the processor and screen modules were integrated with the keyboard interpreter. The test bench was useful in making sure that the right instruction was being sent at the correct time and that the key inputs were taking the finite state machine to the right state. Once the test bench confirmed that the keyboard handler was transiting states properly, we connected the keyboard interpreter to the processor and to the screen display. This allowed us to see if the macros and instructions were actually manipulating the data correctly and that the dynamic sprites were displaying the proper data. When we integrated all three modules together, we realized that the macro coe file was incorrect and we started debugging the macros until they were doing the correct thing for each state. During this stage, we also caught a few state transition errors that caused the keyboard module to call the wrong instruction at a particular state. This caused us to double check all of the instructions and caused the keyboard interpreter module to become more streamlined. In Figure 14 it can be seen that the keyboard interpreter module correctly transitions from state to state and issues the instruction at the proper moment.

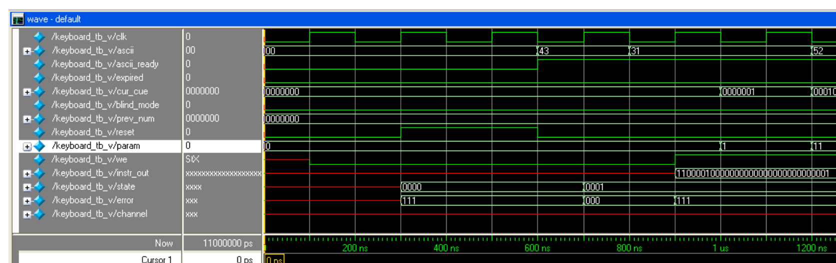

Figure 14: Recording a cue

The initial version of the keyboard module was over 1500 lines and not very efficient. This was due to the fact that there was a case statement for each number 0-9 for every state, which added a considerable amount of coded lines (about 800 lines). This logic became simplified in a single if statement that checked the hex input since all of the numbers were sequentially numbered from 8'h30 to 8'h39. This allowed for a lot of unnecessary logic to be removed and lessened the propagation delay of the module. This also made it

easier to see all of the keys used in each state on the screen at the same time and made it easier to double check the logic and instructions.

4.3 Processor Testing

The Processor could not be tested in ModelSim because of the 3 memory modules. The Control Logic and ALU were each individually tested using ModelSim, but the integration testing was done entirely on the labkit.

4.3.1 Control Logic Testing

The Control Logic Module was tested in ModelSim to verify the signals were correct for the given opcodes, state, and valid and invalid instructions. The state indicates whether the processor is executing a macro or is waiting for instructions from the instruction buffer. Instr_ready was always high, so that the processor would not idle. The table below lists the correct signals for all instructions. Remember that z is the signal indicating whether the value out of the first port of the register file is 0 or not. Not all instructions were

Table 6: Control Logic Test Cases

Instruction	Opcode	mpcsel	bsel	ra2sel	wdsel	alufn	werf	cmwe	start	update
Buffer Mode Tests										
OP	10****	–	0	0	0	****	1	0	0	0
OPC	11****	–	1	0	0	****	1	0	0	0
MACRO	001000	1	0	0	0	–	0	0	0	0
Macro Mode Tests										
OP	10****	0	0	0	0	****	1	0	0	0
OPC	11****	0	1	0	0	****	1	0	0	0
RET	001001	0	0	0	0	1001	0	0	0	0
BEQ	001010	z	0	0	0	1010	0	0	0	0
BNE	001011	z	0	0	0	1011	0	0	0	0
LOAD	001000	0	1	0	1	0000	1	0	0	0
STORE	001000	0	1	1	0	0000	0	1	0	0
START	010000	0	0	0	0	0000	0	0	1	0
UPDATE	01001	0	0	0	0	0000	0	0	0	1

tested in ModelSim, but at least one instruction from each category below was tested in simulation. Some instructions such as macro or branch are only used in either macro mode or buffer mode and were only testing in those modes. Others are used in both and were tested in both modes. Much of the testing was left to the processor and integration testing due to the simplicity of the module.

4.3.2 ALU testing

All ALU operations were just tested with two operands using ModelSim. More extensive testing was not necessary or possible because the module is basically a case statement. Also, the ALU is thoroughly tested by the processor testing in addition to the integration testing. The tests are listed below.

4.3.3 Processor Integration Testing

The processor was tested on the labkit by placing the test instructions into labkit.v file. Different instructions could be selected using the switches on the labkit. One of the button on the labkit was used to place an instruction into the instructions buffer every time it was pressed. The control signals and some of the registers were hooked up to the logic analyzer to check the tests. Some test macros were also written and

Table 7: ALU Test Cases

Op	**alufn**	**a**	**b**	**Output**
addition	0000	20	10	30
bitwise and	1000	31	20	20
bitwise or	1001	20	10	30
left shift	1100	4	2	16
right shift	1101	-31	30	3
signed right shift	1110	-31	30	-2147483648
subtraction	0001	20	10	10
bitwise xor	1010	20	10	30
equal	0100	20	30	0
less than	0101	20	20	0
less than equal to	0110	20	20	1

placed into the ROM for testing of the macros. Figures 15 and 16 show an ADDC and an ADD instruction on the logic analyzer.

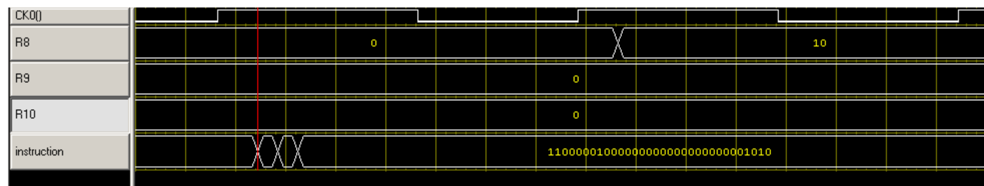

Figure 15: ADDC instruction

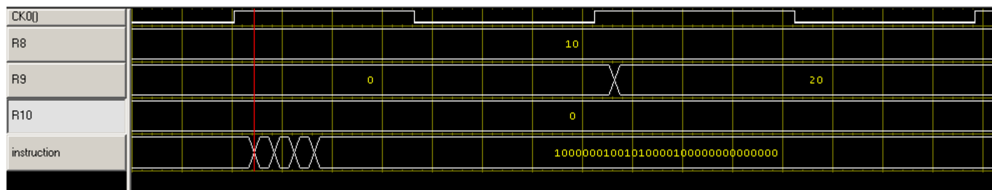

Figure 16: ADD Instruction

4.4 DMX controller Testing

There was no way to usefully test the DMX controller in ModelSim, subsequently the only testing performed on the controller was done using the labkit. Both the logic analyzer and the oscilloscope were used. Labkit testing was done by hooking up the 8 switches on the labkit to a channel and using the switches to control the channel intensity. Then some channels were wired to constants as reference.

Figure 17 shows a DMX-512 packet on the logic analyzer. The long periods of high/low/high is the mark before break, the break and the mark after break. The faster changes after those are the slots being sent.

4.5 Integration Testing

Integration testing was done with the labkit and both dimmer boxes. The screen and lights were checked for the correct behavior for each of the valid key sequences. The valid key sequences are included below.

Key	**Result**
Test Case: Record current cue to cue #	
Cue (C)	Message: Enter a num 1-127
number (1-127)	number appears in upper right corner of screen
Record (R)	Message: Enter to Confirm
Enter	Cue is recorded to indicated number if originally in the scratch cue, goes to recorded cue
Test Case: Record current cue	
Record (R)	Message: Enter to confirm if in scratch cue, Message: Enter a num 1-127
Enter	Cue is saved
Test Case: Alter the intensity of a Channel	
Channel (H)	Message: Enter a num 1-8
number(1-8)	no visual
At (a)	message: Enter intensity
number(1-100)	number appears below the channel selected
Enter	channel intensity is changed and channel is captured
Test Case: Set the intensity of a Channel at 100%	
Channel (H)	Message: Enter a num 1-8
number(1-8)	no visual
At (a)	Message: Enter intensity
Full (O)	100 appears below the channel selected
Enter	channel intensity is changed and channel is captured
Test Case: To release the captured channel intensities	
Release (J)	All captured channels are released and are no longer green
Test Case: To clear the current entry	
Clear Entry (E)	The entry that you were editing is cleared
Test Case: To reset back to the default state	
Reset (Z)	registers are cleared and sent to default state
Test Case: To set the uptime of a cue	
Uptime (U)	Message: Enter a num 0-255
number(0-255)	number appears in the uptime slot in the bottom boxes
Enter	uptime is saved for the current cue
Test Case: To set the downtime of a cue	
Downtime (D)	Message: Enter a num 0-255
number(0-255) Enter	number appears in the downtime slot in the bottom boxes downtime is saved for the current cue
Test Case: To set the wait time of a cue	
Wait Time (W) number(0-255) Enter	Message: Enter a num 0-255 number appears in the wait time slot in the bottom boxes wait time is saved for the current cue
Test Case: To set the follow time of a cue	
Follow time(F) number(1-255) Enter	Message: Enter a num 1-255 number appears in the follow time slot in the bottom boxes follow time is saved for the current cue
Test Case: To link a cue to another cue	
Link (L) number(1-127) Enter	Message: Enter a num 1-127 number appears in the link slot in the bottom boxes link is saved for the current cue

Table 8: Lighting Board Test Cases

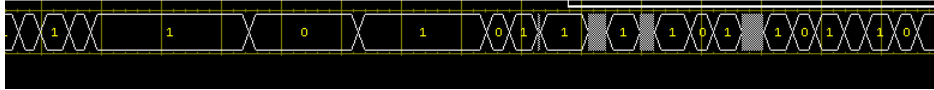

Figure 17: A DMX-512 packet

Key	**Result**
Test Case: To change the mode to live	
Live(S)	Live is displayed by the Mode header at the top of the screen restore last cue in live mode
Test Case: To change the mode to blind	
Live(B)	Blind is displayed by the Mode header at the top of the screen
Test Case: To clear the cue from the stage	
Clear Q (Q)	The cue is cleared from the screen and stage and cue 0 is loaded
Test Case: To load the previous cue onto stage	
Backspace Enter	previous cue number appears in the top right corner of the screen cue is loaded to the stage
Test Case: To load a cue immediately to stage	
Cue (C) number(1-127) Enter	Message: Enter a number 1-127 cue number is displayed at the top right of the screen the cue is loaded to the stage with an uptime of 0
Test Case: To load a cue to stage	
Cue (C) number(1-127) Go (G)	Message: Enter a number 1-127 cue number is displayed at the top right of the screen the cue is loaded to the stage with the correct timing
Test Case: To load a cue to stage	
Go (G)	the cue is loaded to the stage with the correct timing

Table 9: More Lighting Board Test Cases

5 Conclusion

5.1 Keyboard Interpreter

The keyboard interpreter module the user to effectively use the lighting board console to the fullest ability of the board. The keyboard was the gateway to making sure that the lighting board would be able to be used. This module did not have that many huge design decisions that would have made it difficult to implement our project. A lot of the original problems that were run into dealt with simple logical errors when the finite state machine for the whole module was implemented. The part of this subsystem that took the most time was making sure that the correct instruction was being sent at the right clock cycle. It was very important to make sure that the instruction was being sent correctly otherwise the lighting board would not function in the manner that was expected. Creating the states for each instance of the special key assignment was the easiest part of this module. This meant making sure that all of the necessary steps that needed to happen whenever a key was pressed. This was a lot of booking, but it was very important to make sure that all information was being handled properly inside the module, so that when it was combined with the other modules they would work together smoothly to create a lighting board.

5.2 Screen

The screen was the window for the user to see what was actually being processed by the lighting board. The screen displayed the information for each cue that was relevant to the user. It took many tries to try and get the screen to function without showing any glitches. The two sprite modules together are displaying over fifty sprites and this amount of combinational logic was too much for the xvga module to calculate in a 65mhz clock cycle. To fix the glitching the module was run off of a 40mhz clock that supported a screen resolution of 800x600 that went through 3 levels of pipelining. The part of this module that took the most time was trying to remove all of the glitches from the screen display. The intriguing part of this subsystem of the project was creating a manner in which the relevant data was displayed to the screen since the channel data had the potential of three sources of data input. It was very interesting developing an algorithm that would allow the channel data to be tracked and displayed properly.

5.3 Processor

In spite of being a responsible for integrating the whole project, the processor was the easiest part. There was not a lot of design involved since the architecture is based on the β. The processor was also fairly easy to isolate and test to ensure that it was functioning properly without the other modules. One problem that came up was that since the processor was unpipelined, the data came out of the memory at the same time the register was being written, on the next clock cycle. This was solved by clocking the memory on the falling edge of the clock, so that the first half of the clock period could be used to calculate the address and the second half could be used to get the data out of the memory and to the register. This meant that we had to clock our processor a bit slower than we intended because all calculations for a load instruction had to happen in the first half of the clock cycle.

5.4 DMX

The DMX module was the unknown factor at the beginning of our project. Initially we thought we would use the RS-232 serial port on our labkit to talk to the dimmer box. This plan did not work because we would not find a low cost RS-232 to 5-pin XLR RS-485 converter. In the end, the final solution was much easier that a converter. We purchased a TI converter chip that would take the output from one of the labkit's user pins and supply the correct differential voltage for the RS-485 architecture used by the dimmer boxes. Then all that was needed was a 5-pin XLR connector for plugging into the dimmer box. That ended up being a good solution because the interface with the labkit was so simple.

The other risk in using DMX was the serial protocol because there is no way to debug the dimmer box. We just had to hope that the boxes would recognize the serial data we were sending. Initially we forgot that the first slot in each packet is a code for the following data and the dimmer box would refuse the data unless the first channel was set to 0. Thankfully, there were no other problems after the initial mistake with the serial protocol and the lights started displaying properly.

5.5 Overall Reflections

Maura I thought that this project was a great experience. I really learned a lot about how to go about implementing a real life application. It was very useful being able to break everything down into specific subsystems and then tackle each subsystem on its own. I feel like this class and especially this project has really made me realize the value in handling big tasks in smaller chunks. I was really excited that we were able to make our project fully functional and easy to use. It was really exciting for me personally to be able to create a lighting board since I work with them all the time in theatres and never really thought that I would be able to create a fully functional lighting board. I was very impressed with our ability to do so.

Irene I thought this project worked out very well. It was a lot of work but that meant that it was better when it finally worked in the end. The hardest part for me was learning how a lighting board works and then being able to replicate it. The best we could do was use a real lighting board to figure out the specifications and then take a guess at how it was doing it. I was very impressed by how many features of a real lighting board we could implement with our labkit.

6 Appendix

6.1 Theatre Lighting Term Definitions

This section summarizes some important theater lighting terms.

- blind mode – when the light board is in this mode, you can edit future cues without altering the current light intensities (i.e. the current cue)
- channel – a label for a dimmer so that the light intensity can be set; in our design the dimmer and the channel are synonymous
- clear cue – clears the current cue from the stage and resets the current cue to 0
- cue – specific combination of light intensities for one stage scene
- dimmer – the power source for stage lights that controls the intensity of up to two lights; controlled by DMX
- dimmer box – the physical box for powering theatre lights; A dimmer box contains 4 dimmers
- down time – the amount of time it takes to fade a cue down
- follow – a way to have the next cue follow the present cue after a certain period of time
- go – signal to the lighting board to transition to the next cue
- intensity – a percentage of the maximum light brightness
- link – a way to link cues non–sequentially
- live mode – when the lighting board is in this mode, any changes made to the current cue will affect the current light intensities
- release – this releases the channels that have been captured which happens after you record a cue, but don't reset the intensity of the channels to 0
- up time – the amount of time it takes to fade a cue up to the final level
- wait – the amount of time the lighting board waits to load the cue after the go signal.

6.2 Instruction Set

This section contains information about the processor architecture used for this project.

6.2.1 Instruction Formats

Arithmetic Operations without Literal

31 − 10xxxx(6) − 26	25 − rc(5) − 21	20 − ra(5) − 16	15 − rb(5) − 11	10 − unused(11) − 0

Arithmetic Operations with Literal

31 − 11xxxx(6) − 26	25 − rc(5) − 21	20 − ra(5) − 16	15 − signed literal(16) − 0

Cue Memory Access

31 − 01xxxx(6) − 26	25 − rc(5) − 21	20 − 00001 − 16	15 − signed literal(16) − 0

Macro Operations

31 − 00xxxx(6) − 26	25 − unused(18) − 8	7 − address(8) − 0

6.2.2 Opcodes

Table 10: Instruction Set

Instruction	Opcode	Description
Arithmetic Operations without Literal		
ADD	100000	rc ← ra + rb
AND	101000	rc ← ra & rb
OR	101001	rc ← ra ‖ rb
SHL	101100	rc ← ra << rb
SHR	101101	rc ← ra >> rb
SRA	101110	rc ← SEXT(ra >> rb)
SUB	100001	rc ← ra - rb
XOR	101010	rc ← ra ⊗ rb
Comparison Operations without Literal		
CMPEQ	100100	rc ← ra==rb
CMPLT	100101	rc ← ra¡rb
CMPLE	100110	rc ← ra¡=rb
Arithmetic Operations with Literal		
ADDC	110000	rc ← ra+SEXT(literal)
ANDC	111000	rc ← ra&SEXT(literal)
ORC	111001	rc ← ra‖SEXT(literal)
SHLC	111100	rc ← ra<<literal
SHRC	111101	rc ← ra>>literal
SRAC	111110	rc ← SEXT(ra>>literal)
SUBC	110001	rc ← ra-SEXT(literal)
XORC	111010	rc ← ra ⊗ SEXT(literal)
Comparison Operations with Literal		
CMPEQC	110100	rc ← ra==SEXT(literal)
CMPLTC	110101	rc ← ra¡SEXT(literal)
CMPLEC	110110	rc ← ra¡=SEXT(literal)
Cue Memory Access		
LD	011000	rc ← Mem[ra+SEXT(literal)]
ST	011001	Cue_Mem[ra+SEXT(literal)] ← rc
Macro Operations		
MACRO	001000	MPC ← address
RET	001001	Return to instruction buffer
Branch		
BEQ	001010	MPC ← address if 1
BNE	001011	MPC ← address if 0
DMX		
START	010000	Starts the load cue timer
UPDATE	010001	Updates the lights on stage

Table 11: Opcode Table

2:0 / 5:3	000	001	010	011	100	101	110	111
000								
001	MACRO	RET	BEQ	BNE				
010	START	UPDATE						
011	LD	ST						
100	ADD	SUB			CMPEQ	CMPLT	CMPLE	
101	AND	OR	XOR		SHL	SHR	SRA	
110	ADDC	SUBC			CMPEQC	CMPLTC	CMPLEC	
111	ANDC	ORC	XORC		SHLC	SHRC	SRAC	

6.3 Screen Display Verilog Files

This section contains the Verilog files used to create the screen for this project.

6.3.1 Binary to String Converter

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Engineer: Maura Cordial
//
// Create Date:    21:26:51 12/03/06
// Module Name:    bin_string
// Project Name:   Theatre Lighting Board
// Description:    This module takes in an 8 bit binary number and converts it
// into a character string.
//
////////////////////////////////////////////////////////////////////////////////

module bin_string(b_in, cstring_out);

    //imput
  input signed [7:0] b_in; //binary number input
    //it is signed because it is used in subtraction and the number should
    //stay positive.
    //output
  output [23:0]      cstring_out;

    //registers that will be used to convert binary into a character string
    reg [7:0]        hun;
  reg [7:0]        ten;
  reg [7:0]        one;
  reg signed [8:0]   temp_num;
  reg signed [8:0]   temp_num2;
  wire signed [8:0]  data_signed;

  assign        data_signed = {1'b0, b_in};
  assign        cstring_out = {hun, ten, one};

  //we only want to convert the binary input into a character string whenever the
  //binary number has changed. This module deals with signed numbers so that all
  //of the numbers will be positive once we assigned data_signed to always have
  //its first number be a zero.

  always @(data_signed)
    begin
      //this will collect the hundreds character for the string based on comparing
      //the value of the binary input after subtrating 100. This determines whether
      //the hundreds character is a 2, 1, or nothing. After 100 is subracted this value
      //is stored in a temporary register. If the value is less than 100, the temporary
      //value equals itself.
        if((data_signed -100) >= 100)
      begin
        hun = "2";
        temp_num = data_signed - 200;
      end
      else
      if((data_signed -100) >= 0)
        begin
          hun = "1";
          temp_num = data_signed - 100;
        end
      else
        begin
          hun = "_";
          temp_num = data_signed;
        end
    end

  //This always block gets evaluated after the value of temp_num has changed, which happens
  //after the binary number has changed. This always block determines the value of the tens
```

```
//digit and the appropiate character string. The tens value is found by subtracting a tens
//value starting from 90 and working down to 10. If after you subtract the multiple of ten
//the number is greater than zero, the tens place is assigned the appropiate character
//string and temp_num2 is assigned a new value. If the value of temp_num is less than 10
//then nothing gets displayed and temp_num is equal to temp_num2.
always @(temp_num)
  begin
    if((temp_num - 90) >= 0)
    begin
      ten = "9";
      temp_num2 = temp_num - 90;
    end
    else
    if((temp_num - 80) >= 0)
        begin
        ten = "8";
        temp_num2 = temp_num - 80;
      end
    else
      if((temp_num - 70) >= 0)
      begin
        ten = "7";
        temp_num2 = temp_num - 70;
      end
      else
      if((temp_num - 60) >= 0)
        begin
          ten = "6";
          temp_num2 = temp_num - 60;
        end
      else
        if((temp_num - 50) >= 0)
        begin
          ten = "5";
          temp_num2 = temp_num -50;
        end
        else
        if((temp_num - 40) >= 0)
          begin
            ten = "4";
            temp_num2 = temp_num - 40;
          end
        else
          if((temp_num - 30) >= 0)
          begin
            ten = "3";
            temp_num2 = temp_num - 30;
          end
          else
          if((temp_num - 20) >= 0)
            begin
              ten = "2";
              temp_num2 = temp_num - 20;
            end
          else
            if((temp_num - 10) >= 0)
            begin
              ten = "1";
              temp_num2 = temp_num - 10;
            end
            else
            if(hun == "_")
              begin
                ten = "_";
                temp_num2 = temp_num;
              end
            else
              begin
                ten = "0";
                temp_num2 = temp_num;
              end
  end

//This always blocks figures out the one digit. It is moduled very similiarly to
//the tens block above, but it subtracts 9 through 0 and if the result is equal to
//zero, then the ones digit gets assigned to that corresponding characer string.
always@ (temp_num2)
  begin
    if((temp_num2 - 9) == 0)
    one = "9";
    else
    if((temp_num2 - 8) == 0)
        one = "8";
    else
      if((temp_num2 - 7) == 0)
      one = "7";
      else
      if((temp_num2 - 6) == 0)
        one = "6";
      else
        if((temp_num2 - 5) == 0)
        one = "5";
        else
        if((temp_num2 - 4) == 0)
          one = "4";
        else
          if((temp_num2 - 3) == 0)
          one = "3";
          else
          if((temp_num2 - 2) == 0)
            one = "2";
          else
```

```
                if((temp_num2 - 1) == 0)
                one = "1";
                else
                if(ten == "_")
                  one = "_";
                else
                  one = "0";
      end

endmodule
```

6.3.2 Color Tracking

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Engineer: Maura Cordial
//
// Create Date:    16:37:37 11/29/06
// Module Name:    color_tracking
// Project Name:   Lighting Board
// Description:    The purpose of this module is to change the color of the dynamic
//sprites on the screen. The color that the sprite is depends on a few conditions.
//The manner in which is changes font is by passing a pixel_color value to the
//char_string_display, which alters the color of the font, based on the conditions
//for each sprite. The first condition is that if you are currently editting a sprite,
//that sprite will refect the real time changes that you are editing. The font color
//will be yellow. If you have 'captured' channels on stage, then the font color for
//everything that is captured will be green.
//As captured channels are released, the font color will then depend on the past
//value. If the value of the channel is the same as it was in the previous cue, then
//the color of the font will be red. If the channel is set at a different intensity level
//then the font color will be magenta. If the channel was not used in the previous cue,
//then the font color is white.
//
//The Color_Tracking2 module the module only keeps track of the current data that you are
//editing. It would show the data that you are currently editing would be yellow, otherwise
//the data would be displayed as white.
//
////////////////////////////////////////////////////////////////////////////////

module color_tracking(clock, d_ch, prev_ch, pixel_color_ch,
            channel, release_flag, state);

  //parameters
  parameter CHANNEL_TEST = 3'b000; //default channel

  //inputs
  input       clock; //20mhz clock
  input [7:0] d_ch, prev_ch;
  input       release_flag;
  input [2:0] channel;
  input [3:0] state;

  //output
  output reg [2:0] pixel_color_ch;
  //this outputs the appropiate color choice for each channel
  //this output will be used in the character display module
  //to pick the right font color.


  //This always block will test the following cases
  //1. If the channel is captured, and if that is false
  //2. If the channel that you are editing is the same channel
  //    that you are in the state for; and if that is false
  //3. If the prev_ch data and the current channel data are equal
  //    while, the prev_ch was not equal to 0, if this is false then
  //4. If the previous channel and the current channel and not equal
  //      but the current channel value is not zero, otherwise
  //5. The default color of the channel is white.

  always@(posedge clock)
    begin
      if(release_flag) //if the channels are captured
      pixel_color_ch <= 3'b101; //color = green
      else
      if ((state == 4'b0011) && (channel == CHANNEL_TEST))
              //if you are editing a channel currently
        pixel_color_ch <= 3'b010; //color = yellow
      else
              if((d_ch == prev_ch) && (prev_ch))
                //prev channel and cur channel value are the same
              pixel_color_ch <= 3'b000; //color = red
          else
        if((prev_ch) && (d_ch!=0))
                  //If there was a previous channel data and the
                  //cur channel data was not 0
          pixel_color_ch <= 3'b001; //color = magenta
        else
          pixel_color_ch <= 3'b011; //color = white, default
    end

endmodule

/////////////////////////////////////////////////
//This module is similiar to the module above, the difference
//is that this module deals with the data for the timing
//parts of the lighting board. This module changes the color
//of the sprite when you are currently editing. It turns it
//yellow. It checks to make sure that the state that you are
//editing is also correct so that not all of the data
//that has the same value lights up yellow.

module color_tracking2(clock, data, state, pixel_color_d);

  parameter STATE_TEST = 4'b0000;

  input       clock; // 20mhz clock
  input [7:0] data;
  input [3:0] state;
  output reg [2:0] pixel_color_d;

  always@(posedge clock)
```

```
    begin
      if(state == STATE_TEST) //so you are editing the sprite
      pixel_color_d <= 3'b010; //color = yellow
      else
      pixel_color_d <= 3'b011; //color = white

    end
endmodule
```

6.3.3 String Display

```
//
// File:    cstringdisp2.v
// Date:    24-Oct-05
// Author: I. Chuang, C. Terman
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// INPUTS:
//
//    vclock        - video pixel clock
//    hcount        - horizontal (x) location of current pixel
//    vcount        - vertical (y) location of current pixel
//    cstring       - character string to display (8 bit ASCII for each char)
//    cx,cy         - pixel location (upper left corner) to display string at
//
// OUTPUT:
//
//    pixel         - video pixel value to display at current location
//
// PARAMETERS:
//
//    NCHAR         - number of characters in string to display
//    NCHAR_BITS    - number of bits to specify NCHAR
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.
//
// Each character is 8x12, but pixels are doubled horizontally and vertically
// so fonts are magnified 2x.  On an XGA screen (1024x768) you can fit
// 64 x 32 such characters.
//
// Needs font_rom.v and font_rom.ngo
//
// For different fonts, you can change font_rom.  For different string
// display colors, change the assignment to cpixel.


////////////////////////////////////////////////////////////////////////////
//
// video character string display
//
////////////////////////////////////////////////////////////////////////////

module char_string_display2 (vclock,hcount,vcount,pixel,cstring,cx,cy);

  parameter NCHAR = 8;  // number of 8-bit characters in cstring
  parameter NCHAR_BITS = 3; // number of bits in NCHAR

  input      vclock; // 40MHz clock
  input [10:0] hcount;  // horizontal index of current pixel (0..799)
  input [9:0]  vcount; // vertical index of current pixel (0..599)
  output [2:0] pixel; // char display's pixel
  input [NCHAR*8-1:0] cstring;  // character string to display
  input [10:0]      cx;
  input [9:0]       cy;

  // 1 line x 8 character display (8 x 12 pixel-sized characters)

  wire [10:0]       hoff = hcount-1-cx;
  wire [9:0]        voff = vcount-cy;
  wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // < NCHAR
  wire [2:0]          h = hoff[3:1];              // 0 .. 7
  wire [3:0]          v = voff[4:1];        // 0 .. 11

  // look up character to display (from character string)
  reg [7:0]           char;
  integer           n;
  always @(*)
      for (n=0 ; n<8 ; n = n+1 )     // 8 bits per character (ASCII)
    char[n] <= cstring[column*8+n];

  // look up raster row from font rom
  wire            reverse = char[7];
  wire [10:0]         font_addr = char[6:0]*12 + v;     // 12 bytes per character
  wire [7:0]          font_byte;
  font_rom f(font_addr,vclock,font_byte);

  // generate character pixel if we're in the right h,v area
  wire [2:0]          cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0;
  wire            dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16)
                  & (vcount < cy + 24));
  wire [2:0]          pixel = dispflag ? cpixel : 0;

endmodule
```

6.3.4 String Display with Color Tracking

```
//
// File:    cstringdisp.v
// Date:    24-Oct-05
// Author: I. Chuang, C. Terman with modifcations by Maura Cordial
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// INPUTS:
//
//    vclock        - video pixel clock
//    hcount        - horizontal (x) location of current pixel
//    vcount        - vertical (y) location of current pixel
//    cstring       - character string to display (8 bit ASCII for each char)
//    cx,cy         - pixel location (upper left corner) to display string at
//
// OUTPUT:
//
//    pixel         - video pixel value to display at current location
//
// PARAMETERS:
//
//    NCHAR         - number of characters in string to display
//    NCHAR_BITS    - number of bits to specify NCHAR
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.
//
// Each character is 8x12, but pixels are doubled horizontally and vertically
// so fonts are magnified 2x.  On an XGA screen (1024x768) you can fit
// 64 x 32 such characters.
//
// Needs font_rom.v and font_rom.ngo
//
// For different fonts, you can change font_rom.  For different string
// display colors, change the assignment to cpixel.
//
//
// Description: This module will display a character string. This module has
// been modified to take in a color_pixel input which will look up the correct
// color of the character string. Each color has a specific meaning that is
// important in interpreting the sreen.


////////////////////////////////////////////////////////////////////////////
//
// video character string display
//
////////////////////////////////////////////////////////////////////////////

module char_string_display(vclock,hcount,vcount,pixel,cstring,cx,cy,color_pixel);

  //parameters
  parameter NCHAR = 8;  // number of 8-bit characters in cstring
  parameter NCHAR_BITS = 3; // number of bits in NCHAR

  //inputs
  input      vclock; // 40MHz clock
  input [10:0] hcount;  // horizontal index of current pixel (0..799)
  input [9:0]  vcount; // vertical index of current pixel (0..599)
  input [NCHAR*8-1:0] cstring;  // character string to display
  input [10:0]     cx;
  input [9:0]      cy;
  input [2:0]      color_pixel; //color choice

  //outputs
  output [2:0]     pixel;  // char display's pixel
  // 1 line x 8 character display (8 x 12 pixel-sized characters)

  wire [10:0]      hoff = hcount-1-cx;
  wire [9:0]       voff = vcount-cy;
  wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // < NCHAR
  wire [2:0]         h = hoff[3:1];          // 0 .. 7
  wire [3:0]         v = voff[4:1];       // 0 .. 11

  // look up character to display (from character string)
  reg [7:0]          char;
  integer          n;

  reg [2:0]          cpixel;
  wire             reverse = char[7];
  wire [10:0]        font_addr = char[6:0]*12 + v;     // 12 bytes per character
  wire [7:0]         font_byte;

    //this always blocks selects the cpixel that will be used for this character string.
    //it basically is a color selector for the string.
  always@(color_pixel)
    begin
      case(color_pixel)
        3'b101: cpixel = (font_byte[7 - h] ^ reverse) ? 2 : 0; // release - green
        3'b100: cpixel = (font_byte[7 - h] ^ reverse) ? 3 : 0; // error and blind - blue
        3'b011: cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0; // default- white
        3'b010: cpixel = (font_byte[7 - h] ^ reverse) ? 6 : 0;  //param (current item selected) - yellow
        3'b001: cpixel = (font_byte[7 - h] ^ reverse) ? 5 : 0;  //channel is in a prev cue, but at a different level - magenta
        3'b000: cpixel = (font_byte[7 - h] ^ reverse) ? 4 : 0;  //channel is in a prev cue, but at the same level - red
        default: cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0;
      endcase
    end

  always @ *
```

```
    for (n=0 ; n<8 ; n = n+1 )     // 8 bits per character (ASCII)
  char[n] <= cstring[column*8+n];

 // look up raster row from font rom
  font_rom f(font_addr,vclock,font_byte);

 // generate character pixel if we're in the right h,v area
 wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16)
        & (vcount < cy + 24));
 wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule
```

6.3.5 Dynamic Sprites

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Engineer: Maura Cordial
// Create Date:    22:12:11 11/09/06
// Module Name:    dynamic_sp
// Project Name:   Theatre Lighting Board
// Description:
//
//
////////////////////////////////////////////////////////////////////////////////
module dynamic_sp(vclock,hcount,vcount,hsync,vsync,blank,
          d_hsync,d_vsync,d_blank,pixel_d, cur_cue_num, prev_num,
          prev_wait, cur_wait,
          ch1, ch2, ch3, ch4, ch5, ch6, ch7, ch8,
          prev_chan1,prev_chan2, prev_chan3, prev_chan4,
          prev_chan5, prev_chan6, prev_chan7, prev_chan8,
          prev_uptime, prev_downtime, cur_uptime, cur_downtime,
          cur_link, cur_follow, next_uptime, next_downtime, next_wait,
          live_mode, state, channel, param, error, captured_flags,
          loading_flag, loading_value0, loading_value1, loading_value2,
          loading_value3, loading_value4, loading_value5, loading_value6,
          loading_value7);

  //INPUT
  input vclock; // 40MHz clock
  input [10:0] hcount;  // horizontal index of current pixel (0..799)
  input [9:0]  vcount; // vertical index of current pixel (0..599)
  input      hsync;   // XVGA horizontal sync signal (active low)
  input      vsync;   // XVGA vertical sync signal (active low)
  input      blank;   // XVGA blanking (1 means output black pixel)

  input [6:0]   cur_cue_num, prev_num;
  input [7:0]   ch1, ch2, ch3, ch4, ch5, ch6, ch7, ch8;
  input [7:0]   prev_chan1, prev_chan2, prev_chan3, prev_chan4;
  input [7:0]   prev_chan5, prev_chan6, prev_chan7, prev_chan8;
  input [7:0]   prev_uptime; //uptime of the previous cue
  input [7:0]   prev_downtime; //downtime of the previous cue
  input [7:0]   prev_wait; //wait time of the previous cue
  input [7:0]   cur_uptime; //uptime of the current cue
  input [7:0]   cur_downtime; //downtime of the current cue
  input [6:0]   cur_link; //link of the current cue
  input [7:0]   cur_follow; //follow time of the current cue
  input [7:0]   cur_wait;  //wait time of the current cue
  input [7:0]   next_wait; //wait time of the next cue
  input [7:0]   next_uptime; //uptime of the next cue
  input [7:0]   next_downtime; //downtime of the next cue
  input         live_mode;  //mode
  input [2:0]   channel; //channel state
  input [3:0]   state; //state (from the keyboard)
  input [7:0]   param;
  input [2:0]   error; //error message flag
  input [7:0]   captured_flags; //channel captured flags

  //Inputs for Real time tracking for the Channel Intensities
  input [7:0]   loading_flag;
  input [7:0]   loading_value0;
  input [7:0]   loading_value1;
  input [7:0]   loading_value2;
  input [7:0]   loading_value3;
  input [7:0]   loading_value4;
  input [7:0]   loading_value5;
  input [7:0]   loading_value6;
  input [7:0]   loading_value7;

    //Outputs
    output      d_hsync;         // dynamic sprites horizontal sync
    output      d_vsync;         // dynamic sprites vertical sync
    output      d_blank;         // dynamic sprites blanking
    output [2:0] pixel_d;    // dynamic sprites pixel

  //Wires
  //used for the real value of the channel as a cue is loading
  wire [7:0]    loading_value_ch[7:0];

  wire [6:0]    next_num; //used to calculate the number of the next cue

  //Wires for the Channel Strings and their pixels
  wire [23:0]   cstring_ch1;
  wire [2:0]    cdpixel_ch1;
  wire [23:0]   cstring_ch2;
  wire [2:0]    cdpixel_ch2;
  wire [23:0]   cstring_ch3;
  wire [2:0]    cdpixel_ch3;
  wire [23:0]   cstring_ch4;
  wire [2:0]    cdpixel_ch4;
  wire [23:0]   cstring_ch5;
  wire [2:0]    cdpixel_ch5;
  wire [23:0]   cstring_ch6;
  wire [2:0]    cdpixel_ch6;
  wire [23:0]   cstring_ch7;
  wire [2:0]    cdpixel_ch7;
  wire [23:0]   cstring_ch8;
  wire [2:0]    cdpixel_ch8;

  //Wires for the strings of the next, previous
  //and current cues (link, follow, up, down, wait) and
  //their pixels
  wire [23:0]   cstring_prev_uptime;
  wire [2:0]    cdpixel_prev_uptime;
```

```
wire [23:0]   cstring_prev_downtime;
wire [2:0]    cdpixel_prev_downtime;
wire [23:0]   cstring_prev_wait;
wire [2:0]    cdpixel_prev_wait;
wire [23:0]   cstring_prev_num;
wire [2:0]    cdpixel_prev_num;
wire [23:0]   cstring_next_uptime;
wire [2:0]    cdpixel_next_uptime;
wire [23:0]   cstring_next_downtime;
wire [2:0]    cdpixel_next_downtime;
wire [23:0]   cstring_next_wait;
wire [2:0]    cdpixel_next_wait;
wire [23:0]   cstring_next_num;
wire [2:0]    cdpixel_next_num;

//Wires for the Current Cue
wire [23:0]   cstring_cur_uptime;
wire [2:0]    cdpixel_cur_uptime;
wire [2:0]    cdpixel_cur_uptime2;
wire [23:0]   cstring_cur_downtime;
wire [2:0]    cdpixel_cur_downtime;
wire [2:0]    cdpixel_cur_downtime2;
wire [23:0]   cstring_cur_link;
wire [23:0]   cstring_link_f2;
wire [2:0]    cdpixel_cur_link;
wire [23:0]   cstring_cur_follow;
wire [2:0]    cdpixel_cur_follow;
wire [23:0]   cstring_cur_f2;
wire [23:0]   cstring_cur_wait;
wire [2:0]    cdpixel_cur_wait;
wire [2:0]    cdpixel_cur_wait2;
wire [23:0]   cstring_cur_cue_num;
wire [23:0]   cstring_cur_cue_num_top;
wire [2:0]    cdpixel_cur_cue_num;
wire [2:0]    cdpixel_cur_cue_num2;
wire [2:0]    cdpixel_cur_cue_num3;

//Wires for the pixel_color for each Channel to be used
//in color tracking of each channel
wire [2:0]    pixel_color_ch1;
wire [2:0]    pixel_color_ch2;
wire [2:0]    pixel_color_ch3;
wire [2:0]    pixel_color_ch4;
wire [2:0]    pixel_color_ch5;
wire [2:0]    pixel_color_ch6;
wire [2:0]    pixel_color_ch7;
wire [2:0]    pixel_color_ch8;

//Wires/Regs for the pixel_color of the data of the current cue
//and for the operation mode and cue number
  reg [2:0]     pixel_color_mode;
  wire [2:0]    pixel_color_num;
  wire [2:0]    pixel_color_wait;
  wire [2:0]    pixel_color_link;
  wire [2:0]    pixel_color_follow;
  wire [2:0]    pixel_color_up;
  wire [2:0]    pixel_color_down;
  reg [39:0]    cstring_mode;
  wire [2:0]    cdpixel_mode;

  //Wires used to calculate the current value of the channel after a
  //series of comparisons
  wire [7:0]    d_ch1;
  wire [7:0]    d_ch2;
  wire [7:0]    d_ch3;
  wire [7:0]    d_ch4;
  wire [7:0]    d_ch5;
  wire [7:0]    d_ch6;
  wire [7:0]    d_ch7;
  wire [7:0]    d_ch8;

  //Wires used to calculate the current value of the parameters of the
  //current cue after testing to see if you are currently editing that
  //data.
  wire [7:0]    d_cur_uptime;
  wire [7:0]    d_cur_downtime;
  wire [7:0]    d_cur_link;
  wire [7:0]    d_cur_follow;
  wire [7:0]    d_cur_wait;
  wire [7:0]    d_cue_num;

//Wires/Regs used for the error message displays
reg [39:0]    cstring_error2;
wire [2:0]    cdpixel_error2;
reg [127:0]   cstring_error;
wire [2:0]    cdpixel_error;

/////////////////////////////////////////////////////////////
//Assignments
assign      d_hsync = hsync;
assign      d_vsync = vsync;
assign      d_blank = blank;

assign      next_num = cur_cue_num + 1;
reg [2:0]     temp [1:0];
reg [2:0]     temp1 [7:0];
reg [2:0]     temp0 [29:0];

/////////////////////////////////////////////////////////////

//This section is to check and see if the channel has been changed from the saved value
//This part assigns the value of the channel if it is being currently loaded in a cue. This
```

```
//allows the screen to be continously updated along with what the lights show on stage.
assign      loading_value_ch[0] = ((loading_value0[7]) ? 50:0) + ((loading_value0[6]) ? 25:0) +
                    ((loading_value0[5]) ? 13:0) + ((loading_value0[4]) ? 6:0) +
                    ((loading_value0[3]) ? 3:0) + ((loading_value0[2]) ? 2:0) +
                    ((loading_value0[1]) ? 1:0);
assign      loading_value_ch[1] = ((loading_value1[7]) ? 50:0) + ((loading_value1[6]) ? 25:0) +
                    ((loading_value1[5]) ? 13:0) + ((loading_value1[4]) ? 6:0) +
                    ((loading_value1[3]) ? 3:0) + ((loading_value1[2]) ? 2:0) +
                    ((loading_value1[1]) ? 1:0);
assign      loading_value_ch[2] = ((loading_value2[7]) ? 50:0) + ((loading_value2[6]) ? 25:0) +
                    ((loading_value2[5]) ? 13:0) + ((loading_value2[4]) ? 6:0) +
                    ((loading_value2[3]) ? 3:0) + ((loading_value2[2]) ? 2:0) +
                    ((loading_value2[1]) ? 1:0);
assign      loading_value_ch[3] = ((loading_value3[7]) ? 50:0) + ((loading_value3[6]) ? 25:0) +
                    ((loading_value3[5]) ? 13:0) + ((loading_value3[4]) ? 6:0) +
                    ((loading_value3[3]) ? 3:0) + ((loading_value3[2]) ? 2:0) +
                    ((loading_value3[1]) ? 1:0);
assign      loading_value_ch[4] = ((loading_value4[7]) ? 50:0) + ((loading_value4[6]) ? 25:0) +
                    ((loading_value4[5]) ? 13:0) + ((loading_value4[4]) ? 6:0) +
                    ((loading_value4[3]) ? 3:0) + ((loading_value4[2]) ? 2:0) +
                    ((loading_value4[1]) ? 1:0);
assign      loading_value_ch[5] = ((loading_value5[7]) ? 50:0) + ((loading_value5[6]) ? 25:0) +
                    ((loading_value5[5]) ? 13:0) + ((loading_value5[4]) ? 6:0) +
                    ((loading_value5[3]) ? 3:0) + ((loading_value5[2]) ? 2:0) +
                    ((loading_value5[1]) ? 1:0);
assign      loading_value_ch[6] = ((loading_value6[7]) ? 50:0) + ((loading_value6[6]) ? 25:0) +
                    ((loading_value6[5]) ? 13:0) + ((loading_value6[4]) ? 6:0) +
                    ((loading_value6[3]) ? 3:0) + ((loading_value6[2]) ? 2:0) +
                    ((loading_value6[1]) ? 1:0);
assign      loading_value_ch[7] = ((loading_value7[7]) ? 50:0) + ((loading_value7[6]) ? 25:0) +
                    ((loading_value7[5]) ? 13:0) + ((loading_value7[4]) ? 6:0) +
                    ((loading_value7[3]) ? 3:0) + ((loading_value7[2]) ? 2:0) +
                    ((loading_value7[1]) ? 1:0);

////////////////////////////////////////////////////

//displays the channel value
//The value of each channel depends on if the channel is currently being edited, which the first mux
//tests. If the channel is not being edited, the second mux tests to see if the channel is currently being
//loaded to the screen. If the channel is not being loaded, the value used is the value stored in the
//appropiate register for each channel.

assign      d_ch1 = ((state == 4'b0011) && (channel == 3'b000)) ? param :
            (loading_flag[0]) ? loading_value_ch[0] : ch1;
assign      d_ch2 = ((state == 4'b0011) && (channel == 3'b001)) ? param :
            (loading_flag[1]) ? loading_value_ch[1] : ch2;
assign      d_ch3 = ((state == 4'b0011) && (channel == 3'b010)) ? param :
            (loading_flag[2]) ? loading_value_ch[2] : ch3;
assign      d_ch4 = ((state == 4'b0011) && (channel == 3'b011)) ? param :
            (loading_flag[3]) ? loading_value_ch[3] : ch4;
assign      d_ch5 = ((state == 4'b0011) && (channel == 3'b100)) ? param :
            (loading_flag[4]) ? loading_value_ch[4] : ch5;
assign      d_ch6 = ((state == 4'b0011) && (channel == 3'b101)) ? param :
            (loading_flag[5]) ? loading_value_ch[5] : ch6;
assign      d_ch7 = ((state == 4'b0011) && (channel == 3'b110)) ? param :
            (loading_flag[6]) ? loading_value_ch[6] : ch7;
assign      d_ch8 = ((state == 4'b0011) && (channel == 3'b111)) ? param :
            (loading_flag[7]) ? loading_value_ch[7] : ch8;

/////////////////////////////////////////////////////
//The extended data for the current cue is handled in a manner similiar to the data of the channel,
//though the extended data is only checked to see if it is currently being edited. If it is currently
//being edited, then that value is displayed in the param register, otherwise the screen displays the value
//that is stored in the extended data register

assign      d_cur_uptime = (state == 4'b1000) ? param : cur_uptime;
assign      d_cur_downtime = (state == 4'b1001) ? param : cur_downtime;
assign      d_cur_link = (state == 4'b0101) ? param : {1'b0,cur_link};
assign      d_cur_follow = (state == 4'b0111) ? param : cur_follow;
assign      d_cur_wait = (state == 4'b0100) ? param : cur_wait;
assign      d_cue_num = (state == 4'b0001) ? param : {1'b0,cur_cue_num};

//this is where everything will be outputed back to the screen
//if follow is equal to
  //this mux only allows cur and link to display a value on the screen if the data is not 0
assign      cstring_cur_f2 = (d_cur_follow) ? cstring_cur_follow : "   ";
assign      cstring_link_f2 = (d_cur_link) ? cstring_cur_link : "   ";


  //this is the pixel output for the dynanic sprites module
assign      pixel_d = temp[0] | temp[1];

/////////////////////////////////////////////////////////////////////////////////
//This always block takes in the error message from the keyboard module and displays the appropiate
//error message to the screen.

always@(error)
    begin
        case(error)
      3'b000: begin
                cstring_error2 = "1-127";
        cstring_error = "Enter a num     "; end
            3'b001: begin
        cstring_error2 = "1-8  ";
        cstring_error = "Enter a num     "; end
            3'b010: begin
        cstring_error2 = "0-100";
        cstring_error = "Enter a num     "; end
          3'b011: begin
        cstring_error2 = "0-255";
        cstring_error = "Enter a num     "; end
          3'b100: begin
```

```
          cstring_error2 = "1-255";
          cstring_error = "Enter_a_num_____"; end
            3'b101: begin
          cstring_error = "Enter_to_confirm";
          cstring_error2 = "_____"; end
              3'b110: begin
          cstring_error =  "Enter_intensity_";
          cstring_error2 = "_____"; end
              3'b111: begin
          cstring_error = "________________";
          cstring_error2 = "_____"; end
              default: begin
          cstring_error = "________________";
          cstring_error2 = "_____"; end
        endcase
  end

////////////////////////////////////////////////////////////////
//This block allows the correct module with the correct color to
//be displayed depending on the live_flag that is input into the
//module from the processor.

always @(live_mode)
  begin
    if(~live_mode)
    begin
      cstring_mode = "Live_";
      pixel_color_mode = 3'b000; //red
    end
    else
    begin
      cstring_mode = "Blind";
      pixel_color_mode = 3'b100; //blue
    end
  end

/////////////////////////////////////////////
//Pipelining Screen
/////
//This always block pipelines the screen so that it reduces the
//amount of glitches that are preset by giving the module more time
//to complete the necessary logic

always @ (posedge vclock) begin
  temp0[0] <= cdpixel_cur_cue_num;
  temp0[1] <= cdpixel_mode;
  temp0[2] <= cdpixel_cur_cue_num2;
  temp0[3] <= cdpixel_cur_cue_num3;
  temp0[4] <= cdpixel_ch8;
  temp0[5] <= cdpixel_ch7;
  temp0[6] <= cdpixel_ch6;
  temp0[7] <= cdpixel_ch5;
  temp0[8] <= cdpixel_ch4;
  temp0[9] <= cdpixel_ch3;
  temp0[10] <= cdpixel_ch2;
  temp0[11] <= cdpixel_ch1;
  temp0[12] <= cdpixel_prev_uptime;
  temp0[13] <= cdpixel_prev_downtime;
  temp0[14] <= cdpixel_cur_uptime;
  temp0[15] <= cdpixel_cur_downtime;
  temp0[16] <= cdpixel_cur_link;
  temp0[17] <= cdpixel_cur_follow;
  temp0[18] <= cdpixel_next_uptime;
  temp0[19] <= cdpixel_next_downtime;
  temp0[20] <= cdpixel_cur_wait;
  temp0[21] <= cdpixel_next_wait;
  temp0[22] <= cdpixel_prev_wait;
  temp0[23] <= cdpixel_next_num;
  temp0[24] <= cdpixel_prev_num;
  temp0[25] <= cdpixel_cur_wait2;
  temp0[26] <= cdpixel_cur_uptime2;
  temp0[27] <= cdpixel_cur_downtime2;
  temp0[28] <= cdpixel_error;
  temp0[29] <= cdpixel_error2;

  temp1[0] <=  temp0[0] | temp0[1] | temp0[2] | temp0[3];
  temp1[1] <=  temp0[4] | temp0[5] | temp0[6] | temp0[7];
  temp1[2] <=  temp0[8] | temp0[9] | temp0[10] | temp0[11];
  temp1[3] <=  temp0[12] |temp0[13] | temp0[14] |temp0[15];
  temp1[4] <=  temp0[16] | temp0[17]| temp0[18] | temp0[29];
  temp1[5] <=  temp0[19] | temp0[20]| temp0[21] | temp0[22];
  temp1[6] <= temp0[23] | temp0[24]| temp0[25] | temp0[26];
  temp1[7] <= temp0[27] | temp0[28];

  temp[0] <= temp1[0] | temp1[1] | temp1[2] | temp1[3];
  temp[1] <= temp1[4] | temp1[5] | temp1[6] | temp1[7];

end

//////////////////////////////////////////////////////
//COLOR TRACKING
//////
//this block of code is used to color track the data of the channels
//and changes the color of whatever data the user is editing by turning that data yellow
//until the information is saved in the registers. If the person is editing a channel, when the user
//presses enter the channel data is turned green to represent it being captured.
//Below is a quick summary of the colors, though it is explained in detail in the actual color_tracking module
//red is the same intensity in the past and current cue
//magenta is that they both were used, but at different intensities
//white not used in past cue
//param - yellow
//Mode - live is red and blind is blue
```

```
  color_tracking ch1c(vclock, d_ch1, prev_chan1, pixel_color_ch1, channel, captured_flags[7], state);
  defparam ch1c.CHANNEL_TEST = 3'b000;
  color_tracking ch2c(vclock, d_ch2, prev_chan2, pixel_color_ch2, channel, captured_flags[6], state);
  defparam ch2c.CHANNEL_TEST = 3'b001;
  color_tracking ch3c(vclock, d_ch3, prev_chan3, pixel_color_ch3, channel, captured_flags[5], state);
  defparam ch3c.CHANNEL_TEST = 3'b010;
  color_tracking ch4c(vclock, d_ch4, prev_chan4, pixel_color_ch4, channel, captured_flags[4], state);
  defparam ch4c.CHANNEL_TEST = 3'b011;
  color_tracking ch5c(vclock, d_ch5, prev_chan5, pixel_color_ch5, channel, captured_flags[3], state);
  defparam ch5c.CHANNEL_TEST = 3'b100;
  color_tracking ch6c(vclock, d_ch6, prev_chan6, pixel_color_ch6, channel, captured_flags[2], state);
  defparam ch6c.CHANNEL_TEST = 3'b101;
  color_tracking ch7c(vclock, d_ch7, prev_chan7, pixel_color_ch7, channel, captured_flags[1], state);
  defparam ch7c.CHANNEL_TEST = 3'b110;
  color_tracking ch8c(vclock, d_ch8, prev_chan8, pixel_color_ch8, channel, captured_flags[0], state);
  defparam ch8c.CHANNEL_TEST = 3'b111;

  color_tracking2 cur_upc(vclock, d_cur_uptime, state, pixel_color_up);
  defparam cur_upc.STATE_TEST = 4'b1000;
  color_tracking2 cur_downc(vclock, d_cur_downtime, state, pixel_color_down);
  defparam cur_downc.STATE_TEST = 4'b1001;
  color_tracking2 cur_followc(vclock, d_cur_follow, state, pixel_color_follow);
  defparam cur_followc.STATE_TEST = 4'b0111;
  color_tracking2 cur_linkc(vclock, d_cur_link, state, pixel_color_link);
  defparam cur_linkc.STATE_TEST = 4'b0101;
  color_tracking2 cur_waitc(vclock, d_cur_wait, state, pixel_color_wait);
  defparam cur_waitc.STATE_TEST = 4'b0100;
  color_tracking2 cur_numc(vclock, d_cue_num, state, pixel_color_num);
  defparam cur_numc.STATE_TEST = 4'b0001;


//////////////////////////////////////////////////////
//BINARY TO STRING CONVERSION INSTANTIATIONS
//////
//This module converts an 8 bit binary number into a character string
//that is 3 characters long. This character string is displayed in the
//character string display instatiations below.

  bin_string cur_cue_num_string_top(d_cue_num, cstring_cur_cue_num_top);
  bin_string cur_cue_num_string({1'b0,cur_cue_num}, cstring_cur_cue_num);
  bin_string next_num_string({1'b0,next_num}, cstring_next_num);
  bin_string prev_num_string({1'b0,prev_num}, cstring_prev_num);

  bin_string ch1_string(d_ch1, cstring_ch1);
  bin_string ch2_string(d_ch2, cstring_ch2);
  bin_string ch3_string(d_ch3, cstring_ch3);
  bin_string ch4_string(d_ch4, cstring_ch4);
  bin_string ch5_string(d_ch5, cstring_ch5);
  bin_string ch6_string(d_ch6, cstring_ch6);
  bin_string ch7_string(d_ch7, cstring_ch7);
  bin_string ch8_string(d_ch8, cstring_ch8);

  bin_string prev_uptime_string(prev_uptime, cstring_prev_uptime);
  bin_string prev_downtime_string(prev_downtime, cstring_prev_downtime);
  bin_string prev_wait_string(prev_wait, cstring_prev_wait);

  bin_string cur_uptime_string(d_cur_uptime, cstring_cur_uptime);
  bin_string cur_downtime_string(d_cur_downtime, cstring_cur_downtime);
  bin_string cur_link_string(d_cur_link, cstring_cur_link);
  bin_string cur_follow_string(d_cur_follow, cstring_cur_follow);
  bin_string cur_wait_string(d_cur_wait, cstring_cur_wait);

  bin_string next_uptime_string(next_uptime, cstring_next_uptime);
  bin_string next_downtime_string(next_downtime, cstring_next_downtime);
  bin_string next_wait_string(next_wait, cstring_next_wait);

//////////////////////////////////////////////////////////////////////////////
//CHARACTER STRING DISPLAY INSTANTIATIONS

//////////////////////////////////////////////
//These character string display instantionations display the error sprites.
//These sprites are located between the second row of the channel intensity and
//the dividing line across the screen. The data is recieved from a case statement
//from above.

  char_string_display errordisplay(vclock,hcount,vcount,cdpixel_error,cstring_error,
                                   11'd350,10'd300,3'b100);
  defparam errordisplay.NCHAR = 16; // number of 8-bit characters in cstring
  defparam errordisplay.NCHAR_BITS = 5; // number of bits in NCHAR

  char_string_display errordisplay2(vclock,hcount,vcount,cdpixel_error2,cstring_error2,
                                    11'd550,10'd300,3'b100);
  defparam errordisplay2.NCHAR = 5; // number of 8-bit characters in cstring
  defparam errordisplay2.NCHAR_BITS = 3; // number of bits in NCHAR

////////////////////////////////////////////////
//These character string display instantiations display the mode and current cue number
//sprites at the top of the screen. There values are also assigned at the top depending
//on the blind_mode flag value.

  char_string_display cur_cue_num_display_top(vclock,hcount,vcount,cdpixel_cur_cue_num3,
                                              cstring_cur_cue_num_top,11'd750,10'd40, pixel_color_num);
  defparam cur_cue_num_display_top.NCHAR = 3; // number of 8-bit characters in cstring
  defparam cur_cue_num_display_top.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display mode_display_top(vclock,hcount,vcount,cdpixel_mode,
                                       cstring_mode,11'd450,10'd40, pixel_color_mode);
  defparam mode_display_top.NCHAR = 5;  // number of 8-bit characters in cstring
  defparam mode_display_top.NCHAR_BITS = 3; // number of bits in NCHAR

////////////////////////////////////////////////////////////
```

```
//These character string display instantiations display the sprites for the information
//for each channel intensity level. They are displayed under the appropiate channel
//header.

  char_string_display ch1_display(vclock,hcount,vcount,cdpixel_ch1,cstring_ch1,
                                  11'd100,10'd185, pixel_color_ch1);
  defparam ch1_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam ch1_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display ch2_display(vclock,hcount,vcount,cdpixel_ch2,cstring_ch2,
                                  11'd300,10'd185, pixel_color_ch2);
  defparam ch2_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam ch2_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display ch3_display(vclock,hcount,vcount,cdpixel_ch3,cstring_ch3,
                                  11'd500,10'd185, pixel_color_ch3);
  defparam ch3_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam ch3_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display ch4_display(vclock,hcount,vcount,cdpixel_ch4,cstring_ch4,
                                  11'd700,10'd185, pixel_color_ch4);
  defparam ch4_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam ch4_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display ch5_display(vclock,hcount,vcount,cdpixel_ch5,cstring_ch5,
                                  11'd100,10'd265, pixel_color_ch5);
  defparam ch5_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam ch5_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display ch6_display(vclock,hcount,vcount,cdpixel_ch6,cstring_ch6,
                                  11'd300,10'd265, pixel_color_ch6);
  defparam ch6_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam ch6_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display ch7_display(vclock,hcount,vcount,cdpixel_ch7,cstring_ch7,
                                  11'd500,10'd265, pixel_color_ch7);
  defparam ch7_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam ch7_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display ch8_display(vclock,hcount,vcount,cdpixel_ch8,cstring_ch8,
                                  11'd700,10'd265, pixel_color_ch8);
  defparam ch8_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam ch8_display.NCHAR_BITS = 2; // number of bits in NCHAR

//////////////////////////////////////////////////////////////
//These character display instantiations display the sprites for the information
//for the current cue in the bottom left box on the screen. The information that
//gets displayed is the current cue number, the up time, the down time, the wait
//time, the follow time, and if it is linked to another cue or not.

  char_string_display cur_cue_num_display(vclock,hcount,vcount,cdpixel_cur_cue_num,
                                          cstring_cur_cue_num,11'd265,10'd350, 3'b011);
  defparam cur_cue_num_display.NCHAR = 3; // number of 8-bit characters in cstring
  defparam cur_cue_num_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display cur_uptime_display(vclock,hcount,vcount,cdpixel_cur_uptime,
                                         cstring_cur_uptime,11'd215,10'd390, pixel_color_up);
  defparam cur_uptime_display.NCHAR = 3;  // number of 8-bit characters in cstring
  defparam cur_uptime_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display cur_downtime_display(vclock,hcount,vcount,cdpixel_cur_downtime,
                                           cstring_cur_downtime,11'd215,10'd430, pixel_color_down);
  defparam cur_downtime_display.NCHAR = 3;  // number of 8-bit characters in cstring
  defparam cur_downtime_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display cur_follow_display(vclock,hcount,vcount,cdpixel_cur_follow,
                                         cstring_cur_f2,11'd215,10'd470, pixel_color_follow);
  defparam cur_follow_display.NCHAR = 3;  // number of 8-bit characters in cstring
  defparam cur_follow_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display cur_wait_display(vclock,hcount,vcount,cdpixel_cur_wait,
                                       cstring_cur_wait,11'd215,10'd510, pixel_color_wait);
  defparam cur_wait_display.NCHAR = 3;  // number of 8-bit characters in cstring
  defparam cur_wait_display.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display cur_link_display(vclock,hcount,vcount,cdpixel_cur_link,
                                       cstring_link_f2,11'd215,10'd550, pixel_color_link);
  defparam cur_link_display.NCHAR = 3;  // number of 8-bit characters in cstring
  defparam cur_link_display.NCHAR_BITS = 2; // number of bits in NCHAR

//////////////////////////////////////////////////////////////
//These character display instantiations display the sprites that contain
//the information for the previous, current, and next cue in the bottom right
//box. It displays the cue number, wait, up time, and downtime for those
//three cues.

  char_string_display cur_cue_num_display_boxR(vclock,hcount,vcount,cdpixel_cur_cue_num2,
                                               cstring_cur_cue_num,11'd420,10'd450, 3'b011);
  defparam cur_cue_num_display_boxR.NCHAR = 3;  // number of 8-bit characters in cstring
  defparam cur_cue_num_display_boxR.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display prev_num_display_boxR(vclock,hcount,vcount,cdpixel_prev_num,
                                            cstring_prev_num,11'd420,10'd400, 3'b011);
  defparam prev_num_display_boxR.NCHAR = 3; // number of 8-bit characters in cstring
  defparam prev_num_display_boxR.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display next_num_display_boxR(vclock,hcount,vcount,cdpixel_next_num,
                                            cstring_next_num,11'd420,10'd500, 3'b011);
  defparam next_num_display_boxR.NCHAR = 3; // number of 8-bit characters in cstring
  defparam next_num_display_boxR.NCHAR_BITS = 2; // number of bits in NCHAR

  char_string_display next_uptime_display(vclock,hcount,vcount,cdpixel_next_uptime,
```

```
                                           cstring_next_uptime,11'd525,10'd500, 3'b011);
    defparam next_uptime_display.NCHAR = 3; // number of 8-bit characters in cstring
    defparam next_uptime_display.NCHAR_BITS = 2; // number of bits in NCHAR

    char_string_display cur_uptime_display2(vclock,hcount,vcount,cdpixel_cur_uptime2,
                                           cstring_cur_uptime,11'd525,10'd450, pixel_color_up);
    defparam cur_uptime_display2.NCHAR = 3; // number of 8-bit characters in cstring
    defparam cur_uptime_display2.NCHAR_BITS = 2; // number of bits in NCHAR

    char_string_display prev_uptime_display(vclock,hcount,vcount,cdpixel_prev_uptime,
                                           cstring_prev_uptime,11'd525,10'd400, 3'b011);
    defparam prev_uptime_display.NCHAR = 3; // number of 8-bit characters in cstring
    defparam prev_uptime_display.NCHAR_BITS = 2; // number of bits in NCHAR

    char_string_display next_downtime_display(vclock,hcount,vcount,cdpixel_next_downtime,
                                             cstring_next_downtime,11'd615,10'd500, 3'b011);
    defparam next_downtime_display.NCHAR = 3; // number of 8-bit characters in cstring
    defparam next_downtime_display.NCHAR_BITS = 2; // number of bits in NCHAR

    char_string_display cur_downtime_display2(vclock,hcount,vcount,cdpixel_cur_downtime2,
                                             cstring_cur_downtime,11'd615,10'd450, pixel_color_down);
    defparam cur_downtime_display2.NCHAR = 3; // number of 8-bit characters in cstring
    defparam cur_downtime_display2.NCHAR_BITS = 2; // number of bits in NCHAR

    char_string_display prev_downtime_display(vclock,hcount,vcount,cdpixel_prev_downtime,
                                             cstring_prev_downtime,11'd615,10'd400, 3'b011);
    defparam prev_downtime_display.NCHAR = 3; // number of 8-bit characters in cstring
    defparam prev_downtime_display.NCHAR_BITS = 2; // number of bits in NCHAR

    char_string_display prev_wait_display(vclock,hcount,vcount,cdpixel_prev_wait,
                                         cstring_prev_wait,11'd715,10'd400, 3'b011);
    defparam prev_wait_display.NCHAR = 3; // number of 8-bit characters in cstring
    defparam prev_wait_display.NCHAR_BITS = 2; // number of bits in NCHAR

    char_string_display next_wait_display(vclock,hcount,vcount,cdpixel_next_wait,
                                         cstring_next_wait,11'd715,10'd500, 3'b011);
    defparam next_wait_display.NCHAR = 3; // number of 8-bit characters in cstring
    defparam next_wait_display.NCHAR_BITS = 2; // number of bits in NCHAR

    char_string_display BRcur_wait_display2(vclock,hcount,vcount,cdpixel_cur_wait2,
                                           cstring_cur_wait,11'd715,10'd450, pixel_color_wait);
    defparam BRcur_wait_display2.NCHAR = 3; // number of 8-bit characters in cstring
    defparam BRcur_wait_display2.NCHAR_BITS = 2; // number of bits in NCHAR

endmodule
```

6.3.6 Static Sprites

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Engineer: Maura Cordial
//
// Create Date:    20:55:01 11/07/06
// Module Name:    static_sp
// Project Name:   Theatre Lighting Board
// Description:    This module displays all of the labels on the screen. The lables
// include the channel 1 through 8 headings, the mode you are operating in, cue number
// in the top right of the screen, the headings for the bottom boxes on the left and
// right.
//
////////////////////////////////////////////////////////////////////////////////
module static_sp(vclock,hcount,vcount,hsync,vsync,blank,
              static_hsync,static_vsync,static_blank,pixel_s);

   //inputs
   input vclock;  // 40MHz clock
   input [10:0] hcount; // horizontal index of current pixel (0..799)
   input [9:0]  vcount; // vertical index of current pixel (0..599)
   input hsync;   // XVGA horizontal sync signal (active low)
   input vsync;   // XVGA vertical sync signal (active low)
   input blank;   // XVGA blanking (1 means output black pixel)

   //outputs
   output static_hsync; // static sprites horizontal sync
   output static_vsync; // static sprites vertical sync
   output static_blank; // static sprites blanking
   output [2:0] pixel_s;  // static sprites pixel

    //Wires

    //the Border Boxes
  wire [2:0] blob_pixel, blob_pixel2;

    //Wires for the Headings for the Bottom Left Box
    //The Character string and the pixel for each string
    wire [79:0] cstring_BLnumcue = "Cue_Number";
    wire [2:0]  cdpixel_BLnumcue;
    wire [55:0] cstring_BLup = "Up_Time";
    wire [2:0]  cdpixel_BLup;
    wire [71:0] cstring_BLdown = "Down_Time";
    wire [2:0]  cdpixel_BLdown;
    wire [47:0] cstring_BLfollow = "Follow";
    wire [2:0]  cdpixel_BLfollow;
    wire [31:0] cstring_BLwait = "Wait";
    wire [2:0]  cdpixel_BLwait;
    wire [31:0] cstring_BLlink = "Link";
    wire [2:0]  cdpixel_BLlink;

    //Wires for the headings for the Bottom Right Box
    //The Character string and the pixel for each string
    wire [23:0] cstring_BRnumcue = "Cue";
    wire [2:0]  cdpixel_BRnumcue;
    wire [15:0] cstring_BRup = "Up";
    wire [2:0]  cdpixel_BRup;
    wire [31:0] cstring_BRdown = "Down";
    wire [2:0]  cdpixel_BRdown;
    wire [31:0] cstring_BRwait = "Wait";
    wire [2:0]  cdpixel_BRwait;

    //Wires for the Channel Headings
    //The Character string and the pixel for each string
    wire [31:0] cstring_ch1 = "Ch_1";
    wire [2:0]  cdpixel_ch1;
    wire [31:0] cstring_ch2 = "Ch_2";
    wire [2:0]  cdpixel_ch2;
    wire [31:0] cstring_ch3 = "Ch_3";
    wire [2:0]  cdpixel_ch3;
    wire [31:0] cstring_ch4 = "Ch_4";
    wire [2:0]  cdpixel_ch4;
    wire [31:0] cstring_ch5 = "Ch_5";
    wire [2:0]  cdpixel_ch5;
    wire [31:0] cstring_ch6 = "Ch_6";
    wire [2:0]  cdpixel_ch6;
    wire [31:0] cstring_ch7 = "Ch_7";
    wire [2:0]  cdpixel_ch7;
    wire [31:0] cstring_ch8 = "Ch_8";
    wire [2:0]  cdpixel_ch8;

    //Wires for the Mode and Cue Header on the Top of the Screen
    //The Character string and the pixel for each string
    wire [39:0] cstring_mode = "Mode:";
    wire [2:0]  cdpixel_mode;
    wire [23:0] cstring_cue = "Cue";
    wire [2:0]  cdpixel_cue;

    //Registers for pipelining the screen
  reg [2:0] temp[1:0];
  reg [2:0] temp1[5:0];
  reg [2:0] temp0[21:0];

    //assigments for the video sync
  assign static_hsync = hsync;
    assign static_vsync = vsync;
    assign static_blank = blank;

    //for the pixel output
    assign pixel_s = temp[1] | temp[0];
```

```
////////////////////////////////////////////////////////////
//This is to pipeline the sprites to prevent glitching
//It is a three stage pipelining system

always @ (posedge vclock) begin
    temp0[0] <= cdpixel_ch1;
    temp0[1] <= cdpixel_ch2;
    temp0[2] <= cdpixel_ch3;
    temp0[3] <= cdpixel_ch4;
    temp0[4] <= cdpixel_ch5;
    temp0[5] <= cdpixel_ch6;
    temp0[6] <= cdpixel_ch7;
    temp0[7] <= cdpixel_ch8;
    temp0[8] <= cdpixel_BLnumcue;
    temp0[9] <= cdpixel_BLup;
    temp0[10] <= cdpixel_BLdown;
    temp0[11] <= cdpixel_BLfollow;
    temp0[12] <= cdpixel_BLwait;
    temp0[13] <= cdpixel_BLlink;
    temp0[14] <= cdpixel_BRup;
    temp0[15] <= cdpixel_BRdown;
    temp0[16] <= cdpixel_BRnumcue;
    temp0[17] <= cdpixel_cue;
    temp0[18] <= cdpixel_mode;
    temp0[19] <= cdpixel_BRwait;
    temp0[20] <= blob_pixel;
    temp0[21] <= blob_pixel2;

    temp1[0] <= temp0[0] | temp0[1]| temp0[2] | temp0[3];
    temp1[1] <= temp0[4] | temp0[5]| temp0[6] | temp0[7];
    temp1[2] <= temp0[8] | temp0[9]| temp0[10] | temp0[11];
    temp1[3] <= temp0[12] | temp0[13]| temp0[14] | temp0[15];
    temp1[4] <= temp0[16] | temp0[17]| temp0[18] | temp0[19];
    temp1[5] <= temp0[20] | temp0[21];

    temp[0] <= temp1[0] | temp1[1] | temp1[2];
    temp[1] <= temp1[4] | temp1[5] | temp1[3];

end

//////////////////////////////////////////////////////////
//This section below is the character display instantion for each sprite to be displayed

   //Channel Headers
      char_string_display2 ch1(vclock,hcount,vcount,cdpixel_ch1,cstring_ch1,11'd100,10'd150);
    defparam ch1.NCHAR = 4; // number of 8-bit characters in cstring
    defparam ch1.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 ch2(vclock,hcount,vcount,cdpixel_ch2,cstring_ch2,11'd300,10'd150);
    defparam ch2.NCHAR = 4; // number of 8-bit characters in cstring
    defparam ch2.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 ch3(vclock,hcount,vcount,cdpixel_ch3,cstring_ch3,11'd500,10'd150);
    defparam ch3.NCHAR = 4; // number of 8-bit characters in cstring
    defparam ch3.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 ch4(vclock,hcount,vcount,cdpixel_ch4,cstring_ch4,11'd700,10'd150);
    defparam ch4.NCHAR = 4; // number of 8-bit characters in cstring
    defparam ch4.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 ch5(vclock,hcount,vcount,cdpixel_ch5,cstring_ch5,11'd100,10'd230);
    defparam ch5.NCHAR = 4; // number of 8-bit characters in cstring
    defparam ch5.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 ch6(vclock,hcount,vcount,cdpixel_ch6,cstring_ch6,11'd300,10'd230);
    defparam ch6.NCHAR = 4; // number of 8-bit characters in cstring
    defparam ch6.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 ch7(vclock,hcount,vcount,cdpixel_ch7,cstring_ch7,11'd500,10'd230);
    defparam ch7.NCHAR = 4; // number of 8-bit characters in cstring
    defparam ch7.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 ch8(vclock,hcount,vcount,cdpixel_ch8,cstring_ch8,11'd700,10'd230);
    defparam ch8.NCHAR = 4; // number of 8-bit characters in cstring
    defparam ch8.NCHAR_BITS = 3; // number of bits in NCHAR

//////////////////////////////////////////////////
//Border Lines Instanations
      blobv blob(hcount,vcount,blob_pixel);
    blobh blob2(hcount,vcount,blob_pixel2);

//////////////////////////////////////////////////////////
//Character String Display Instantations for the headers for the bottom left box

    char_string_display2 boxL_numcue(vclock,hcount,vcount,cdpixel_BLnumcue,cstring_BLnumcue,
                         11'd75,10'd350);
    defparam boxL_numcue.NCHAR = 10;  // number of 8-bit characters in cstring
    defparam boxL_numcue.NCHAR_BITS = 4; // number of bits in NCHAR

    char_string_display2 boxL_up(vclock,hcount,vcount,cdpixel_BLup,cstring_BLup,
                         11'd60,10'd390);
    defparam boxL_up.NCHAR = 7; // number of 8-bit characters in cstring
    defparam boxL_up.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 boxL_down(vclock,hcount,vcount,cdpixel_BLdown,cstring_BLdown,
                         11'd60,10'd430);
    defparam boxL_down.NCHAR = 9; // number of 8-bit characters in cstring
    defparam boxL_down.NCHAR_BITS = 4; // number of bits in NCHAR

    char_string_display2 boxL_follow(vclock,hcount,vcount,cdpixel_BLfollow,cstring_BLfollow,
                         11'd60,10'd470);
```

```
    defparam boxL_follow.NCHAR = 6; // number of 8-bit characters in cstring
    defparam boxL_follow.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 boxL_wait(vclock,hcount,vcount,cdpixel_BLwait,cstring_BLwait,
                                   11'd60,10'd510);
    defparam boxL_wait.NCHAR = 4; // number of 8-bit characters in cstring
    defparam boxL_wait.NCHAR_BITS = 3; // number of bits in NCHAR

    char_string_display2 boxL_link(vclock,hcount,vcount,cdpixel_BLlink,cstring_BLlink,
                                   11'd60,10'd550);
    defparam boxL_link.NCHAR = 4; // number of 8-bit characters in cstring
    defparam boxL_link.NCHAR_BITS = 3; // number of bits in NCHAR

///////////////////////////////////////////////////////////////
//Character String Display Instantations for the headers for the bottom right box

      char_string_display2 boxR_numcue(vclock,hcount,vcount,cdpixel_BRnumcue,cstring_BRnumcue,
                                       11'd415,10'd350);
      defparam boxR_numcue.NCHAR = 3; // number of 8-bit characters in cstring
      defparam boxR_numcue.NCHAR_BITS = 2; // number of bits in NCHAR

      char_string_display2 boxR_up(vclock,hcount,vcount,cdpixel_BRup,cstring_BRup,
                                   11'd523,10'd350);
      defparam boxR_up.NCHAR = 2; // number of 8-bit characters in cstring
      defparam boxR_up.NCHAR_BITS = 2; // number of bits in NCHAR

      char_string_display2 boxR_down(vclock,hcount,vcount,cdpixel_BRdown,cstring_BRdown,
                                     11'd610,10'd350);
      defparam boxR_down.NCHAR = 4; // number of 8-bit characters in cstring
      defparam boxR_down.NCHAR_BITS = 3; // number of bits in NCHAR

      char_string_display2 boxR_wait(vclock,hcount,vcount,cdpixel_BRwait,cstring_BRwait,
                                     11'd710,10'd350);
      defparam boxR_wait.NCHAR = 4; // number of 8-bit characters in cstring
      defparam boxR_wait.NCHAR_BITS = 3; // number of bits in NCHAR

/////////////////////////////////////////////////////////
//Character String Display Instantations for the headers at the top of the Screen

      char_string_display2 mode(vclock,hcount,vcount,cdpixel_mode,cstring_mode,11'd360,10'd40);
      defparam mode.NCHAR = 5;  // number of 8-bit characters in cstring
      defparam mode.NCHAR_BITS = 3; // number of bits in NCHAR

      char_string_display2 cue(vclock,hcount,vcount,cdpixel_cue,cstring_cue,11'd690,10'd40);
      defparam cue.NCHAR = 3; // number of 8-bit characters in cstring
      defparam cue.NCHAR_BITS = 2; // number of bits in NCHAR

endmodule


////////////////////////////////////////////////////////////////////////
//blob gnerate rectangle on screen
////////////////////////////////////////
//This module creates the horizontal line across the screen to divide
//between the channel headings and the boxes of information in the
//bottom half of the screen

module blobh(hcount,vcount,blob_pixel2);

   input [10:0] hcount;
   input [9:0] vcount;
   output reg [2:0] blob_pixel2;

    always @ (hcount or vcount) begin
      if ((hcount <= 383) && (hcount >= 380) && (vcount >= 339))
          blob_pixel2 = 3'b111;
      else blob_pixel2 = 0;
   end

endmodule

//////////////////////////////////////
//This module creates the vertical line across the screen to divide
//between the boxes in the lower half of the screen

module blobv(hcount,vcount,blob_pixel);
  input [10:0] hcount;
   input [9:0] vcount;
   output [2:0] blob_pixel;

   reg [2:0] blob_pixel;
      always @ (hcount or vcount) begin
      if ((vcount <= 339) && (vcount >= 336))
          blob_pixel = 3'b111;
      else blob_pixel = 0;
   end

endmodule
```

6.3.7 XVGA

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: 6.111 Fall 2005 Version of this module with edits by Maura Cordial
//
// Create Date:    21:25:23 11/07/06
// Module Name:    xvga
// Project Name:   Theatre Lighting Board
// Description: This module will generate XVGA display signals for a screen with
// a resolution of 800x600 clocked at a 40mhz clock. The changes to this module
// that I made were a result of changing the module from a 1024x768 resolution
// screen to the 800x600. To make this change I had to change the values of
// hcount, vcount, h and y sync, h and v blank, and their reset values.
//
//
////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (800 x 600 @ 40Hz)
//
////////////////////////////////////////////////////////////////////////////////

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
  input vclock;
  output [10:0] hcount;
  output [9:0]  vcount;
  output        vsync;
  output        hsync;
  output        blank;

  reg       hsync,vsync,hblank,vblank,blank;
  reg [10:0]    hcount;    // pixel number on current line
  reg [9:0]     vcount;  // line number

  // horizontal: 1088 pixels total
  // display 800 pixels per line
  wire      hsyncon,hsyncoff,hreset,hblankon;
  assign      hblankon = (hcount == 799);
  assign      hsyncon = (hcount == 831);
  assign      hsyncoff = (hcount == 959);
  assign      hreset = (hcount == 1088);

  // vertical: 619 lines total
  // display 600 lines
  wire      vsyncon,vsyncoff,vreset,vblankon;
  assign      vblankon = hreset & (vcount == 599);
  assign      vsyncon = hreset & (vcount == 600);
  assign      vsyncoff = hreset & (vcount == 604);
  assign      vreset = hreset & (vcount == 619);

  // sync and blanking
  wire      next_hblank,next_vblank;
  assign      next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign      next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
    hcount <= hreset ? 0 : hcount + 1;
    hblank <= next_hblank;
    hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

    vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
    vblank <= next_vblank;
    vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

    blank <= next_vblank | (next_hblank & ~hreset);
  end

endmodule
```

6.4 Keyboard Handler Verilog Files

This section contains the Verilog files used to handle keyboard input for this project.

6.4.1 Keyboard Debounce

```
//
// File:     ps2_kbd.v
// Date:     24-Oct-05
// Author: C. Terman / I. Chuang with modifications by Maura Cordial
//
// PS2 keyboard input for 6.111 labkit
//
//Description: This module takes in data from the keyboard and outputs
//the appropiate ascii value for that character. The module also outputs
//a ready signal when a key is hit along with the value of the key hit.
//The only real change that I made was to clock this module at 20mhz.

//////////////////////////////////////////////////////////////////////////

module ps2_ascii_input(clk, reset, clock, data, ascii, ascii_ready);

   // module to generate ascii code for keyboard input
   // this is module works synchronously with the system clock

   input clk; //20mhz clock
   input reset;    // Active high asynchronous reset
   input clock;    // PS/2 clock
   input data;     // PS/2 data
   output [7:0] ascii;   // ascii code (1 character)
   output      ascii_ready;  // ascii ready (one clock_27mhz cycle active high)

   reg [7:0]     ascii_val; // internal combinatorial ascii decoded value
   reg [7:0]     lastkey;   // last keycode
   reg [7:0]     curkey;    // current keycode
   reg [7:0]     ascii;   // ascii output (latched & synchronous)
   reg        ascii_ready; // synchronous one-cycle ready flag

   // get keycodes

   wire       fifo_rd;   // keyboard read request
   wire [7:0]    fifo_data; // keyboard data
   wire       fifo_empty;  // flag: no keyboard data
   wire       fifo_overflow; // keyboard data overflow

   ps2 myps2(reset, clk, clock, data, fifo_rd, fifo_data,
         fifo_empty,fifo_overflow);

   assign      fifo_rd = ~fifo_empty; // continous read
   reg        key_ready;

   always @(posedge clk)
       begin

       // get key if ready

       curkey <= ~fifo_empty ? fifo_data : curkey;
       lastkey <= ~fifo_empty ? curkey : lastkey;
       key_ready  <= ~fifo_empty;

       // raise ascii_ready for last key which was read

       ascii_ready <= key_ready & ~(curkey[7]|lastkey[7]);
       ascii <=  (key_ready & ~(curkey[7]|lastkey[7])) ? ascii_val : ascii;

       end

   always @(curkey) begin //convert PS/2 keyboard make code ==> ascii code
     case (curkey)
       8'h1C: ascii_val = 8'h41;    //A
       8'h32: ascii_val = 8'h42;    //B
       8'h21: ascii_val = 8'h43;    //C
       8'h23: ascii_val = 8'h44;    //D
       8'h24: ascii_val = 8'h45;    //E
       8'h2B: ascii_val = 8'h46;    //F
       8'h34: ascii_val = 8'h47;    //G
       8'h33: ascii_val = 8'h48;    //H
       8'h43: ascii_val = 8'h49;    //I
       8'h3B: ascii_val = 8'h4A;    //J
       8'h42: ascii_val = 8'h4B;    //K
       8'h4B: ascii_val = 8'h4C;    //L
       8'h3A: ascii_val = 8'h4D;    //M
       8'h31: ascii_val = 8'h4E;    //N
       8'h44: ascii_val = 8'h4F;    //O
       8'h4D: ascii_val = 8'h50;    //P
       8'h15: ascii_val = 8'h51;    //Q
       8'h2D: ascii_val = 8'h52;    //R
       8'h1B: ascii_val = 8'h53;    //S
       8'h2C: ascii_val = 8'h54;    //T
       8'h3C: ascii_val = 8'h55;    //U
       8'h2A: ascii_val = 8'h56;    //V
       8'h1D: ascii_val = 8'h57;    //W
       8'h22: ascii_val = 8'h58;    //X
       8'h35: ascii_val = 8'h59;    //Y
       8'h1A: ascii_val = 8'h5A;    //Z

       8'h45: ascii_val = 8'h30;    //0
       8'h16: ascii_val = 8'h31;    //1
```

```
      8'h1E: ascii_val = 8'h32;   //2
      8'h26: ascii_val = 8'h33;   //3
      8'h25: ascii_val = 8'h34;   //4
      8'h2E: ascii_val = 8'h35;   //5
      8'h36: ascii_val = 8'h36;   //6
      8'h3D: ascii_val = 8'h37;   //7
      8'h3E: ascii_val = 8'h38;   //8
      8'h46: ascii_val = 8'h39;   //9

      8'h0E: ascii_val = 8'h60;   // `
      8'h4E: ascii_val = 8'h2D;   // -
      8'h55: ascii_val = 8'h3D;   // =
      8'h5C: ascii_val = 8'h5C;   // \
      8'h29: ascii_val = 8'h20;   // (space)
      8'h54: ascii_val = 8'h5B;   // [
      8'h5B: ascii_val = 8'h5D;   // ]
      8'h4C: ascii_val = 8'h3B;   // ;
      8'h52: ascii_val = 8'h27;   // '
      8'h41: ascii_val = 8'h2C;   // ,
      8'h49: ascii_val = 8'h2E;   // .
      8'h4A: ascii_val = 8'h2F;   // /

      8'h5A: ascii_val = 8'h0D;   // enter (CR)
      8'h66: ascii_val = 8'h08;   // backspace

      //  8'hF0: ascii_val = 8'hF0;   // BREAK CODE

      default: ascii_val = 8'h23;   // #
    endcase
  end
endmodule // ps2toascii

////////////////////////////////////////////////////////////////////////////
// new synchronous ps2 keyboard driver, with built-in fifo, from Chris Terman

module ps2(reset, clk, ps2c, ps2d, fifo_rd, fifo_data,
       fifo_empty,fifo_overflow);

  input clk,reset; //20mhz clock
  input ps2c;      // ps2 clock
  input ps2d;      // ps2 data
  input fifo_rd;     // fifo read request (active high)
  output [7:0] fifo_data; // fifo data output
  output      fifo_empty;  // fifo empty (active high)
  output      fifo_overflow; // fifo overflow - too much kbd input

  reg [3:0]     count;      // count incoming data bits
  reg [9:0]     shift;      // accumulate incoming data bits

  reg [7:0]     fifo[7:0];    // 8 element data fifo
  reg       fifo_overflow;
  reg [2:0]     wptr,rptr;    // fifo write and read pointers

  wire [2:0]    wptr_inc = wptr + 1;

  assign      fifo_empty = (wptr == rptr);
  assign      fifo_data = fifo[rptr];

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0]     ps2c_sync;
  always @ (posedge clk) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire      sample = ps2c_sync[2] & ~ps2c_sync[1];

  always @ (posedge clk) begin
    if (reset) begin
      count <= 0;
      wptr <= 0;
      rptr <= 0;
      fifo_overflow <= 0;
    end
    else if (sample) begin
      // order of arrival: 0,8 bits of data (LSB first),odd parity,1
      if (count==10) begin
        // just received what should be the stop bit
        if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
          fifo[wptr] <= shift[8:1];
          wptr <= wptr_inc;
          fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
        end
        count <= 0;
      end else begin
        shift <= {ps2d,shift[9:1]};
        count <= count + 1;
      end
        end
    // bump read pointer if we're done with current value.
    // Read also resets the overflow indicator
    if (fifo_rd && !fifo_empty) begin
      rptr <= rptr + 1;
      fifo_overflow <= 0;
    end
  end

endmodule
```

6.4.2 PS2 to ASCII

```
`timescale 1ns / 1ps
//
// File:    ps2_kbd.v
// Date:    24-Oct-05
// Author: C. Terman / I. Chuang
//
// PS2 keyboard input for 6.111 labkit
//
// INPUTS:
//
//    clock_27mhz   - master clock
//    reset         - active high
//    clock         - ps2 interface clock
//    data          - ps2 interface data
//
// OUTPUTS:
//
//    ascii         - 8 bit ascii code for current character
//    ascii_ready   - one clock cycle pulse indicating new char received

///////////////////////////////////////////////////////////////////////////////

module ps2_ascii_input(clock, reset, keyboard_clock, data, ascii, ascii_ready);

   // module to generate ascii code for keyboard input
   // this is module works synchronously with the system clock

   input clock;        //27 mhz clock
   input reset;            // Active high asynchronous reset
   input keyboard_clock;            // PS/2 clock
   input data;             // PS/2 data
   output [7:0] ascii;    // ascii code (1 character)
   output      ascii_ready;    // ascii ready (one clock_27mhz cycle active high)

   reg [7:0]     ascii_val;        // internal combinatorial ascii decoded value
   reg [7:0]     lastkey;          // last keycode
   reg [7:0]     curkey;           // current keycode
   reg [7:0]     ascii;            // ascii output (latched & synchronous)
   reg       ascii_ready;      // synchronous one-cycle ready flag

   // get keycodes

   wire      fifo_rd;           // keyboard read request
   wire [7:0]    fifo_data;         // keyboard data
   wire      fifo_empty;       // flag: no keyboard data
   wire      fifo_overflow;    // keyboard data overflow

   ps2 myps2(reset, clock, keyboard_clock, data, fifo_rd, fifo_data,
             fifo_empty,fifo_overflow);



   assign      fifo_rd = ~fifo_empty;    // continous read
   reg       key_ready;

   //   keyboard_interp key(clock, ascii, ascii_ready, we, instr_out,reset,param);

   always @(posedge clock)
      begin

          // get key if ready

          curkey <= ~fifo_empty ? fifo_data : curkey;
          lastkey <= ~fifo_empty ? curkey : lastkey;
          key_ready  <= ~fifo_empty;

          // raise ascii_ready for last key which was read

          ascii_ready <= key_ready & ~(curkey[7]|lastkey[7]);
          ascii <=  (key_ready & ~(curkey[7]|lastkey[7])) ? ascii_val : ascii;

      end

   always @(curkey) begin //convert PS/2 keyboard make code ==> ascii code
     case (curkey)
       8'h1C: ascii_val = 8'h41;    //A - At button
       8'h32: ascii_val = 8'h42; //B - blind
       8'h21: ascii_val = 8'h43;//C - Cue button
       8'h23: ascii_val = 8'h44; //D downtime
       8'h24: ascii_val = 8'h45;//E - clear button (empty)
       8'h2B: ascii_val = 8'h46; //F - follow
       8'h34: ascii_val = 8'h47; // G - go button
       8'h33: ascii_val = 8'h48;//H - Channel button
       8'h43: ascii_val = 8'h49;                 //I
       8'h3B: ascii_val = 8'h4A; //J - j/k - Release
       8'h42: ascii_val = 8'h4B;                 //K
       8'h4B: ascii_val = 8'h4C;// L - link
       8'h3A: ascii_val = 8'h4D;    //M
       8'h31: ascii_val = 8'h4E;                 //N
       8'h44: ascii_val = 8'h4F;    // O- full power (on)
       8'h4D: ascii_val = 8'h50;                 //P
       8'h15: ascii_val = 8'h51;//Q - clears a cue
       8'h2D: ascii_val = 8'h52;//R - Record
       8'h1B: ascii_val = 8'h53; //S - Stage Mode
       8'h2C: ascii_val = 8'h54;                 //T
       8'h3C: ascii_val = 8'h55; //U - uptime
       8'h2A: ascii_val = 8'h56;                 //V
```

```
      8'h1D: ascii_val = 8'h57; //W wait
      8'h22: ascii_val = 8'h58;                 //X
      8'h35: ascii_val = 8'h59;                 //Y
      8'h1A: ascii_val = 8'h5A; //Z - reset

      8'h45: ascii_val = 8'h30;                 //0
      8'h16: ascii_val = 8'h31;                 //1
      8'h1E: ascii_val = 8'h32;                 //2
      8'h26: ascii_val = 8'h33;                //3
      8'h25: ascii_val = 8'h34;               //4
      8'h2E: ascii_val = 8'h35;                 //5
      8'h36: ascii_val = 8'h36;                 //6
      8'h3D: ascii_val = 8'h37;                 //7
      8'h3E: ascii_val = 8'h38;                 //8
      8'h46: ascii_val = 8'h39;                 //9

      8'h0E: ascii_val = 8'h60;                 // `
      8'h4E: ascii_val = 8'h2D;                 // -
      8'h55: ascii_val = 8'h3D;                 // =
      8'h5C: ascii_val = 8'h5C;                 // \
      8'h29: ascii_val = 8'h20;                 // (space)
      8'h54: ascii_val = 8'h5B;                 // [
      8'h5B: ascii_val = 8'h5D;                 // ]
      8'h4C: ascii_val = 8'h3B;                 // ;
      8'h52: ascii_val = 8'h27;                 // '
      8'h41: ascii_val = 8'h2C;                 // ,
      8'h49: ascii_val = 8'h2E;                 // .
      8'h4A: ascii_val = 8'h2F;                 // /

      8'h5A: ascii_val = 8'h0D; //Enter (CR) -  Enter
      8'h66: ascii_val = 8'h08; //backspace - go to prev cue

      //  8'hF0: ascii_val = 8'hF0;             // BREAK CODE

      default: ascii_val = 8'h23;               // #
    endcase
  end
endmodule // ps2toascii

////////////////////////////////////////////////////////////////////////////
// new synchronous ps2 keyboard driver, with built-in fifo, from Chris Terman

module ps2(reset, clock, ps2c, ps2d, fifo_rd, fifo_data,
           fifo_empty,fifo_overflow);

  input clock,reset;
  input ps2c;                   // ps2 clock
  input ps2d;                   // ps2 data
  input fifo_rd;                // fifo read request (active high)
  output [7:0] fifo_data;       // fifo data output
  output       fifo_empty;      // fifo empty (active high)
  output       fifo_overflow;   // fifo overflow - too much kbd input

  reg [3:0]    count;       // count incoming data bits
  reg [9:0]    shift;       // accumulate incoming data bits

  reg [7:0]    fifo[7:0];    // 8 element data fifo
  reg      fifo_overflow;
  reg [2:0]    wptr,rptr;    // fifo write and read pointers

  wire [2:0]   wptr_inc = wptr + 1;

  assign     fifo_empty = (wptr == rptr);
  assign     fifo_data = fifo[rptr];

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0]    ps2c_sync;
  always @ (posedge clock) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire     sample = ps2c_sync[2] & ~ps2c_sync[1];

  always @ (posedge clock) begin
    if (reset) begin
      count <= 0;
      wptr <= 0;
      rptr <= 0;
      fifo_overflow <= 0;
    end
    else if (sample) begin
      // order of arrival: 0,8 bits of data (LSB first),odd parity,1
      if (count==10) begin
        // just received what should be the stop bit
        if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
          fifo[wptr] <= shift[8:1];
          wptr <= wptr_inc;
          fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
        end
        count <= 0;
      end else begin
        shift <= {ps2d,shift[9:1]};
        count <= count + 1;
      end
        end
    // bump read pointer if we're done with current value.
    // Read also resets the overflow indicator
    if (fifo_rd && !fifo_empty) begin
      rptr <= rptr + 1;
      fifo_overflow <= 0;
    end
  end

endmodule
```

6.4.3 Keyboard Interpreter

```
`timescale 1ns / 1ps
//Maura Cordial
//Keyboard_interp module
//Theatre Lighting Board

//# ******SPECIAL REGISTERS*******
//#            zero register: R0
//#            current cue address: R1
//#            previous basic data: R2
//#            previous ch1-ch4: R3
//#            previous ch5-ch8: R4
//#            current basic data: R5
//#            current extended data: R6
//#            next basic data: R7
//#            current ch1-ch8: R8-R15
//#            function parameter: R16
//#            additional param: R17
//#            captured channels: R18
//#            live mode flag: R19
//#            illop instr: R31
//
//This module takes the input from the user that was entered in by the keyboard and then
//takes the keystroke to send out the proper instruction the the processor. The module also
//outputs flags (mode, error, channel) to the dynamic sprites module.

module keyboard_interp(clk, ascii, ascii_ready,expired,cur_cue,blind_mode,
             we,instr_out,reset,param,state,error,channel);

  // ***MACRO ADDRESSES*************************
  parameter LOAD_CUE = 8'd0;
  parameter RECORD = 8'd63;
  parameter RELEASE = 8'd88;
  parameter SET_CHAN = 8'd92;
  parameter SET_UP = 8'd166;
  parameter SET_DOWN = 8'd178;
  parameter SET_WAIT = 8'd190;
  parameter SET_FOLLOW = 8'd202;
  parameter SET_LINK = 8'd214;
  parameter ABORT = 8'd226;
  parameter GO = 8'd228;
  parameter BLIND = 8'd237;
  parameter LIVE = 8'd241;
  // ***MACRO ADDRESSES**************************

  //Input Wires
  input       clk; //20mhz clock
  input [7:0] ascii;    // ascii code (1 character)
  input       ascii_ready;   // ascii ready (one clock_20mhz cycle active high)
  input       reset;
  input [7:0] param; //register to store the current data that you are editing
  input [6:0] cur_cue; //the current cue
  input       expired; //timer signal from the timer module
  input       blind_mode; //tells which mode we are operating in

  //outputs
  output reg  we; //the write enable to the processor
  output reg [31:0] instr_out; //the instruction to the processor
  output reg [2:0]  error; //error message flag to the dynamic sprites module
  output reg [3:0]  state; //current state sent to the dynamic sprites module
  output reg [2:0]  channel; //the current channel that you are editing
  reg [15:0]        temp_value; //a temporary value register

  //this always block calculates the value of the series of numbers that
  //the user can enter. It stores this value in temp_value after each keystroke.
  always @ (ascii) begin
    case(ascii)
      8'h30: temp_value = param*10;   //for 0
      8'h31: temp_value = param*10+1; //for 1
      8'h32: temp_value = param*10+2; //for 2
      8'h33: temp_value = param*10+3; //for 3
      8'h34: temp_value = param*10+4; //for 4
      8'h35: temp_value = param*10+5; //for 5
      8'h36: temp_value = param*10+6; //for 6
      8'h37: temp_value = param*10+7; //for 7
      8'h38: temp_value = param*10+8; //for 8
      8'h39: temp_value = param*10+9; //for 9
      default: temp_value = temp_value;
      endcase
  end

  //This always block is set up like a finite state machine.
  //It transitions to another state through a series of
  //keystrokes. At each state there is an abort, clear entry,
  //and then the functions that are appropiate to the state.
  //The state is chosen from the keystrokes entered by the
  //user through the keyboad and this ascii value is used
  //in part to determine the appropiate state to enter.

  always @ (posedge clk) begin
      we <= 0;
    if (reset) begin
      state <= 4'b0000;
      error <= 3'b111;
    end
        else begin
      if (expired) begin //if expired is high the next cue needs to load
           we <= 1'b1;
               instr_out <= {24'b001000000000000000000000, GO};
      end
```

```
else begin
      if(ascii_ready) begin //if a key has been sent
    error <= 3'b111; //default error message - blank
    case(state)
      4'b0000: //default
        case(ascii)
          8'h43: begin
            state <= 4'b0001; //cue, C
            error <= 3'b000; //Enter a number 1-127
          end
          8'h48: begin
            state <= 4'b0010; //channel, H
                        error <= 3'b001;  //Enter a number between 1-8
          end
                    8'h57: begin
                          state <= 4'b0100; //wait, W
                            error <= 3'b011; //Enter a number between 0-255
          end
          8'h4C: begin
            state <= 4'b0101; //link, L
            error <= 3'b000; //Enter a number between 1-127
          end
          8'h52: begin
            state <= 4'b0110; //record, R
            error <= 3'b101; //Press enter to confirm
          end
          8'h46: begin
            state <= 4'b0111; //follow, F
            error <= 3'b100; //Enter a number between 1-255
          end
          8'h55: begin
            state <= 4'b1000; //uptime, U
            error <= 3'b011; //Enter a number between 0-255
          end
          8'h44: begin
            state <= 4'b1001; //downtime, D
            error <= 3'b011; //Enter a number between 0-255
          end
          8'h5A: state <= 4'b0000; //reset, Z
          8'h53: begin //live mode, S
            we <= 1'b1;
            instr_out <= {24'b001000000000000000000000, LIVE};
          end
          8'h42: begin //blind mode, B
            we <= 1'b1;
            instr_out <= {24'b001000000000000000000000, BLIND};
            end
          8'h4A: begin //release,J
            we <= 1'b1;
            instr_out <= {24'b001000000000000000000000,RELEASE};
          end
          8'h51: begin //Q, clears a cue
            we <= 1'b1;
            instr_out <= {24'b001000000000000000000000,LOAD_CUE};
          end
          8'h47: //G, go
                          if(~blind_mode) begin //because live is default low
                              we<=1'b1;
                          instr_out <= {24'b001000000000000000000000,GO};
                          end
          8'h08: //backspace - load previous cue
          begin
            we <= 1'b1;
            instr_out <= {25b1100001000000000000000000,prev_num};
            state <= 4'b0001;
          end
          default: state <= 4'b0000;
                    endcase
      4'b0001: begin //cue
                    if((ascii>=8'h30) && (ascii<=8'h39)) begin //enters 0-9
          we<=1'b1;
          if(temp_value <= 127) //makes sure that it is in a valid range
                          instr_out <= {16'b110000_10000_00000_,temp_value};
          else begin
            state <= 4'b0001; //takes you back to this state
            error <= 3'b000; //Enter a number between 1-127
            instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                    end //for the else
        end //of the begin after the if statement
        else begin
                    case (ascii) //this will handle the key inputs that are not numbers
                          8'h52: begin //Record
                              state <= 4'b0110; //record state
              error <= 3'b101;  //Press enter to confirm
            end
            8'h5A: begin //reset
              we <= 1'b1;
              state <= 4'b0000; //Default
              instr_out <= 32'b10000010000000000000000000000000; //Clears the registers
            end
            8'h45: begin //clears the entry
                              we <= 1'b1;
                              state <= 4'b0001; //Cue State
                              instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                          end
            8'h0D: begin //Enter
              we <= 1'b1;
              //this checks to make sure that the number entered is inside
              //the range of valid cue assignments
              if ((param<128) && (param>0)) begin
                //load cue
                instr_out <= {24'b001000000000000000000000,LOAD_CUE};
```

```
          state <= 4'b0000; //default state
        end
        else begin
                          state <= 4'b001; //Cue State
          error <= 3'b000; //Enter a number between 1-127
          instr_out <= 32'b10000010000000000000000000000000; //clears the registers
        end //for else
      end
      8'h47: //G, go
        if(~blind_mode) begin //because live is default low
                          we<=1'b1;
          instr_out <= {24'b001000000000000000000000,GO};
          state <= 4'b0000; //Default state
        end
      default: state <= 4'b0000;
    endcase
  end
end
            4'b0010: begin //Channel - keeps track of the channel that you are editing
  case(ascii)
                  8'h31: channel <= 3'b000; //Channel 1
    8'h32: channel <= 3'b001; //Channel 2
    8'h33: channel <= 3'b010; //Channel 3
    8'h34: channel <= 3'b011; //Channel 4
    8'h35: channel <= 3'b100; //Channel 5
    8'h36: channel <= 3'b101; //Channel 6
    8'h37: channel <= 3'b110; //Channel 7
    8'h38: channel <= 3'b111; //Channel 8
    8'h5A: begin //reset
                    state <= 4'b0000; //default state
      channel <= 3'b000; //Channel 1 is the default
      we <= 1'b1;
      instr_out <= 32'b10000010000000000000000000000000; //clears the registers
    end
    8'h41: begin //At key
                    instr_out <= {29'b110000_10001_00000_00000_00000_000,channel};
      state <= 4'b0011; //at state
      we <= 1'b1;
      error <= 3'b110; //Enter intensity
    end
    8'h45: begin //clears the entry
      we <= 1'b1;
      state <= 4'b0010; //Channel State
      channel <= 3'b000; //Defaulted to Channel 1
      instr_out <= 32'b10000010000000000000000000000000; //clears the registers
    end
    default: begin
      channel <= 3'b000; //Channel 1 is the default channel
      error <= 3'b001;  //Enter a number between 1-8
    end
              endcase
            end //of the channel state
            4'b0011: //At State- sets the intensity level of a channel
              begin
                if ((ascii>=8'h30) && (ascii<=8'h39)) begin //Keystroke between 0-9
          we<=1'b1;
                    if(temp_value <= 100) //Tests to make sure that it is a valid intensity level
                      instr_out <= {16'b110000_10000_00000_,temp_value};
                    else begin //if an invalid entry
                        state <= 4'b0011; //back to the At State
                        error <= 3'b010; //Enter a number between 0-100
                        instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                    end //of the else begin
                end //of the if
    else begin
      case (ascii) //Handles all of the keystrokes that are not 0-9
        8'h5A: begin //RESET
          we <= 1'b1;
          state <= 4'b0000; //Default State
          //set_abort
          instr_out <= {24'b001000000000000000000000,ABORT}; //resets the registers
        end
        8'h45: begin //clears the entry
          we <= 1'b1;
          state <= 4'b0011; //At STATE
          instr_out <= 32'b10000010000000000000000000000000; //clears the registers
        end
        8'h4F: begin //Full power (100%)
          //this sets the channel intensity level to 100
          we <=1'b1;
          instr_out <= 32'b110000_10000_00000_0000_0000_0110_0100;
        end
        8'h0D: begin //Enter
          if (param<=100) begin //Tests to make sure that is is a valid entry
                                  we <= 1'b1;
                                  //set_chan
                                  instr_out <= {24'b001000000000000000000000,SET_CHAN};
                                  state <= 4'b0000; //Default state
          end
          else begin
            we <= 1'b1;
            state <= 4'b0011; //At State
            error <= 3'b010; //Enter a number 0-100
            instr_out <= 32'b10000010000000000000000000000000; //resets the register value
          end //of else
                        end //of Enter State
                        default: state <= 4'b0011; //At State
                    endcase
                end //of else
              end //of the state
            4'b0100: begin //WAIT STATE - this allows you to set the wait time to load the next cue (0-255)
                if((ascii>=8'h30) && (ascii<=8'h39)) begin //Enter in a value 0-9
```

```
                    we<=1'b1;
                        if(temp_value <= 255) //If the temp_value is within the range of wait
                          instr_out <= {16'b110000_10000_00000_,temp_value}; //store that value
                        else begin
                            state <= 4'b0100; //Wait State
                            error <= 3'b011; //Enter a number between 0-255
                            instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                        end //of the else
                    end //of the if
else begin
  case(ascii) //This handles the other keystrokes
    8'h5A: begin //Reset
      we <= 1'b1;
      state <= 4'b0000; //Default State
      instr_out <= 32'b10000010000000000000000000000000; //clears the registers
    end
    8'h45: begin //clears the entry
      we <= 1'b1;
      state <= 4'b0100; //Wait State
      instr_out <= 32'b10000010000000000000000000000000; //clears the registers
    end
    8'h0D: begin //Enter
      if (param<256) begin //if the value stored in wait is within bounds
                                  we <= 1'b1;
                                  //set wait
                                  instr_out <= {24'b001000000000000000000000,SET_WAIT};
                                  state <= 4'b0000; //Default State
      end //of if
      else begin //the value entered is not within the bounds of wait
                                  we <= 1'b1;
                                  state <= 4'b0100; //Wait State
                                  error <= 3'b011;  //Enter a number 0-255
                                  instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                              end //of else
                          end
                          default: state <= 4'b0100; //Wait State
                      endcase
                  end //of the else
              end //of the wait state
              4'b0101: begin //LINK state - this allows you to link a cue unsequentially to another cue (1-127)
                  if((ascii>=8'h30) && (ascii<=8'h39)) begin //keystrokes between 0-9
                  we<=1'b1;
                      if(temp_value <= 127) //tests to see if the number entered is within bounds
                        instr_out <= {16'b110000_10000_00000_,temp_value};
                      else begin //if the value is out of bounds
                          state <= 4'b0101; //LINK state
                          error <= 3'b000; //Enter a number between 1-127
                          instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                      end //of the else
                  end //of the if
else begin
  case(ascii) //Handles all of the other keystrokes
    8'h5A: begin //Reset
      we <= 1'b1;
      state <= 4'b0000; //Default state
      instr_out <= 32'b10000010000000000000000000000000; //clears the registers
    end
    8'h45: begin //clears the entry
      we <= 1'b1;
      state <= 4'b0101; //Link State
      instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                          end
                          8'h0D: begin //Enter
      //This tests to make sure that the number entered is within bounds
      if ((param >0) && (param<128)) begin
                                  we <= 1'b1;
                                  //set_link
                                  instr_out <= {24'b001000000000000000000000,SET_LINK};
                                  state <= 4'b0000; //Default state
      end
      else begin
                                  we <= 1'b1;
                                  state <= 4'b0101; //Link State
                                  error <= 3'b000; //Enter a number between 1-127
                                  instr_out <= 32'b10000010000000000000000000000000; //clears the registers
      end //of else
                          end
                          default: state <= 4'b0101; //LINK state
                      endcase
                  end //of else begin
              end //of link state
              4'b0110: begin //RECORD STATE - this allows you to record a cue
                  case(ascii)
                      8'h0D: begin //Enter
                          //this tests to make sure that the cue number that you have entered is between 1 and 127
                          if (((param < 128) && (cur_cue != 0)) || ((param > 0) && (param < 128)))
      begin
                                we <= 1'b1;
                                //record
                                instr_out <= {24'b001000000000000000000000,RECORD};
                                state <= 4'b0000; //Default State
                            end
                          else begin
                              we <= 1'b1;
                              state <= 4'b0110;
                              error <= 3'b000; //1-127
                              instr_out <= 32'b10000010000000000000000000000000;
                          end //of else
                      end //of enter state
                      8'h5A: begin //Reset
                          we <= 1'b1;
                          state <= 4'b0000; //Default State
```

```
                        instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                      end
                      default: begin
          state <= 4'b0110; //RECORD state
                        instr_out <= 32'b10000010000000000000000000000000; //clears the regisers
        we <= 1'b1;
      end
    endcase
              end //of record
              4'b0111: begin //FOLLOW state - this allows you to set the auto follow for the next cue (1-255)
                  if((ascii>=8'h30) && (ascii<=8'h39)) begin //Enter a number 0-9
                we<=1'b1;
                      if(temp_value <= 255) //tests to make sure that it is within bounds
                        instr_out <= {16'b110000_10000_00000_,temp_value};
                      else begin //if it is out of bounds
                        state <= 4'b0111; //FOLLOW state
                        error <= 3'b100; //Enter a number 1-255
                        instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                      end //of the else
                  end //of the if
            else begin
                      case(ascii)
                        8'h5A: begin //Reset
                            we <= 1'b1;
                            state <= 4'b0000; //Default state
                            instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                        end
                        8'h45: begin //clears the entry
                            we <= 1'b1;
                            state <= 4'b0111; //FOLLOW state
                            instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                        end
                        8'h0D: begin //Enter
                            //checks to make sure a valid number was entered
                            if (param<256) begin
                                we <= 1'b1;
                                //set_follow
                                instr_out <= {24'b001000000000000000000000,SET_FOLLOW};
                                state <= 4'b0000; //Default state
                            end
                            else begin
                                we <= 1'b1;
                                state <= 4'b0111; //Follow State
                                error <= 3'b100; //Enter a number 1-255
                                instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                            end //of else
                        end
                        default: state <= 4'b0111; //FOLLOW State
                      endcase
                  end
              end //of the follow state
              4'b1000: begin //UP TIME STATE - this allows you to set the uptime of a cue (0-255)
                  if((ascii>=8'h30) && (ascii<=8'h39)) begin //enter a keystroke between 0-9
                we<=1'b1;
                      if(temp_value <= 255) //if the number entered was below 255
                        instr_out <= {16'b110000_10000_00000_,temp_value};
                      else begin
                        state <= 4'b1000;// UP TIME state
                        error <= 3'b011; //Enter a number 0-255
                        instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                      end //of the else
                  end //of the if
      else begin
                      case(ascii) //to handle the other keys entered
                        8'h45: begin //clears the entry
                            we <= 1'b1;
                            state <= 4'b1000; //UP TIME STATE
                            instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                        end
                        8'h5A: begin //Reset
                            we <= 1'b1;
                            state <= 4'b0000; //Default State
                            instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                        end
                        8'h0D: begin //Enter
                            if (param<256) begin //checks to see if it is within the bounds
                                we <= 1'b1;
                                //set_up
                                instr_out <= {24'b001000000000000000000000,SET_UP};
                                state <= 4'b0000; //Default state
                            end
                            else begin
                                we <= 1'b1;
                                state <= 4'b1000; //UP TIME STATE
                                error <= 3'b011; //Enter a number 0 - 255
                                instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                            end //of else
                        end //of enter
                        default: state <= 4'b1000; //UP TIME STATE
                      endcase
                  end
end
              4'b1001: //DOWNTIME STATE - this allows you to set the downtime of a cue
                begin
                    if((ascii>=8'h30) && (ascii<=8'h39)) begin //keys entered 0-9
                  we<=1'b1;
                        if(temp_value <= 255) //if the number entered is less than 256
                          instr_out <= {16'b110000_10000_00000_,temp_value};
                        else begin
                          state <= 4'b1001; //DOWN TIME STATE
                          error <= 3'b011; //Enter a number 0-255
                          instr_out <= 32'b10000010000000000000000000000000; //clears the registers
```

```
                                 end //of the else
                               end //of the if
                         else begin
                                 case(ascii) //handles the other keystrokes
                                   8'h45: begin //clears the entry
                                     we <= 1'b1;
                                     state <= 4'b1001; //DOWNTIME STATE
                                     instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                                   end
                                   8'h5A: begin //Reset
                                     we <= 1'b1;
                                     state <= 4'b0000; //Default State
                                     instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                                   end
                                   8'h0D: begin //Enter
                                     if (param<256) begin //tests to make sure that it is within bounds
                                       we <= 1'b1;
                                       //set_down
                                       instr_out <= {24'b001000000000000000000000,SET_DOWN};
                                       state <= 4'b0000; //Default State
                                     end
                                     else begin
                                       we <= 1'b1;
                                       state <= 4'b1001; //DOWNTIME STATE
                                       error <= 3'b011; //Enter a number 0-255
                                       instr_out <= 32'b10000010000000000000000000000000; //clears the registers
                                     end //of else
                                   end //of enter state
                                   default: state <= 4'b1001; //DOWNTIME STATE
                                 endcase
                               end //of the else
                             end //of the downtime state
                            default: state <= 4'b0000; //DEFAULT STATE
                        endcase
                      end
                    end
                  end
                end
endmodule // keyboard_interp
```

6.4.4 Keyboard Interpreter Test Bench

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:13:35 12/12/2006
// Design Name:    keyboard_interp
// Module Name:    keyboard_tb.v
// Project Name:   lightboard
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: keyboard_interp
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module keyboard_tb_v;

  // Inputs
  reg clk;
  reg [7:0] ascii;
  reg ascii_ready;
  reg expired;
  reg [6:0] cur_cue;
  reg blind_mode;
  reg [6:0] prev_num;
  reg reset;
  reg [7:0] param;

  // Outputs
  wire we;
  wire [31:0] instr_out;
  wire [3:0] state;
  wire [2:0] error;
  wire [2:0] channel;

  // Instantiate the Unit Under Test (UUT)
  keyboard_interp uut (
    .clk(clk),
    .ascii(ascii),
    .ascii_ready(ascii_ready),
    .expired(expired),
    .cur_cue(cur_cue),
    .blind_mode(blind_mode),
    .prev_num(prev_num),
    .we(we),
    .instr_out(instr_out),
    .reset(reset),
    .param(param),
    .state(state),
    .error(error),
    .channel(channel)
  );

  always #100 clk = ~clk;
  initial begin
    // Initialize Inputs
    clk = 0;
    ascii = 0;
    ascii_ready = 0;
    expired = 0;
    cur_cue = 0;
    blind_mode = 0;
    prev_num = 0;
    reset = 0;
    param = 0;

      #300;
    clk = 0;
    ascii = 0;
    ascii_ready = 0;
    expired = 0;
    cur_cue = 0;
    blind_mode = 0;
    prev_num = 0;
    reset = 1;
    param = 0;
    #300;
    ascii = 8'h43;  //cue
    ascii_ready = 1;
    reset = 0;
    param = 0;
    expired = 0;
    cur_cue = 0;
    blind_mode = 0;
    prev_num = 0;
    #200;
    ascii = 8'h31;  //1
    ascii_ready = 1;
    reset = 0;
    param = 0;
```

```
expired = 0;
cur_cue = 0;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h31;  //1
ascii_ready = 1;
reset = 0;
param = 1;
expired = 0;
cur_cue = 1;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h52;//record
ascii_ready = 1;
reset = 0;
param = 11;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h0D;     //enter
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#500;
ascii = 8'h48;     //channel
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h35;     //5
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h41;     //at
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 8'h34;     //4
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 0;
ascii_ready = 0;
reset = 0;
param = 4;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h0D;     //enter
ascii_ready = 1;
reset = 0;
param = 4;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
  #500;
ascii = 8'h57;     //wait
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 8'h37;     //7
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
```

```
prev_num = 0;
#300;
ascii = 0;
ascii_ready = 0;
reset = 0;
param = 7;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h0D;     //enter
ascii_ready = 1;
reset = 0;
param = 7;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 8'h55;     //uptime
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 8'h32;     //2
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 0;
ascii_ready = 0;
reset = 0;
param = 2;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h0d;     //enter
ascii_ready = 1;
reset = 0;
param = 2;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 8'h44;     //downtime
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 8'h32;     //2
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 0;
ascii_ready = 0;
reset = 0;
param = 2;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h0d;     //enter
ascii_ready = 1;
reset = 0;
param = 2;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#300;
ascii = 8'h4C;     //link
ascii_ready = 1;
reset = 0;
param = 0;
expired = 0;
cur_cue = 11;
blind_mode = 0;
prev_num = 0;
#200;
ascii = 8'h38;     //8
```

```
      ascii_ready = 1;
      reset = 0;
      param = 0;
      expired = 0;
      cur_cue = 11;
      blind_mode = 0;
      prev_num = 0;
      #300;
      ascii = 0;
      ascii_ready = 0;
      reset = 0;
      param = 8;
      expired = 0;
      cur_cue = 11;
      blind_mode = 0;
      prev_num = 0;
      #200;
      ascii = 8'h0d;     //enter
      ascii_ready = 1;
      reset = 0;
      param = 8;
      expired = 0;
      cur_cue = 11;
      blind_mode = 0;
      prev_num = 0;
      #500;
      ascii = 8'h46;     //follow
      ascii_ready = 1;
      reset = 0;
      param = 0;
      expired = 0;
      cur_cue = 11;
      blind_mode = 0;
      prev_num = 0;
      #200;
      ascii = 8'h39;     //9
      ascii_ready = 1;
      reset = 0;
      param = 0;
      expired = 0;
      cur_cue = 11;
      blind_mode = 0;
      prev_num = 0;
      #200;
      ascii = 8'h39;     //9
      ascii_ready = 1;
      reset = 0;
      param = 9;
      expired = 0;
      cur_cue = 11;
      blind_mode = 0;
      prev_num = 0;
      #300;
      ascii = 0;
      ascii_ready = 0;
      reset = 0;
      param = 99;
      expired = 0;
      cur_cue = 11;
      blind_mode = 0;
      prev_num = 0;
      #200;
      ascii = 8'h0d;     //enter
      ascii_ready = 1;
      reset = 0;
      param = 99;
      expired = 0;
      cur_cue = 11;
      blind_mode = 0;
      prev_num = 0;
      #300;

   end
endmodule
```

6.5 Processor Verilog Files

This section contains the Verilog files for the processor used for this project.

6.5.1 Processor Module

```
//                                    -*- Mode: Verilog -*-
// Filename         : processor.v
// Description      : Processor used to coordinate between modules
// responsible for store cues
// responsible for sending signals to the dmx module to load cues
// and update channel values

`timescale 1ns / 1ps

module processor(clk, reset, instr_in, bwe,
                 //current cue num
                 cue_num,prev_num,
                 //previous cue basic data
                 prev_up, prev_down, prev_wait,
                 //previous cue channels
                 prev_ch1, prev_ch2, prev_ch3, prev_ch4,
                 prev_ch5, prev_ch6, prev_ch7, prev_ch8,
                 //current cue basic data
                 cur_up, cur_down, cur_wait,
                 //current cue extended data
                 cur_link, cur_follow,
                 //next cue basic data
                 next_up, next_down, next_wait,
                 //current cue channels
                 ch1, ch2, ch3, ch4, ch5, ch6, ch7, ch8,
                 //function paramter
                 param,
                 //flag indicating we are in blind mode
                 blindmode_flag,
                 //instruction buffer full
                 full,
                 //load cue to stage signal
                 start,
                 //update dmx channel intensities signal
                 update,
                 //captured flags
                 captured);

  //clock, reset, instruction buffer write enable inputs
  input clk, reset, bwe;
  //instruction into buffer
  input [31:0] instr_in;

  //current cue num
  output [6:0] cue_num,prev_num;
  //previous cue basic data
  output [7:0] prev_up, prev_down, prev_wait;
  //previous cue channels
  output [7:0] prev_ch1, prev_ch2, prev_ch3, prev_ch4,
               prev_ch5, prev_ch6, prev_ch7, prev_ch8;
  //current cue basic data
  output [7:0] cur_up, cur_down, cur_wait;
  //current cue extended data
  output [6:0] cur_link;
  output [7:0] cur_follow;
  //next cue basic data
  output [7:0] next_up, next_down, next_wait;
  //current cue channels
  output [7:0] ch1, ch2, ch3, ch4, ch5, ch6, ch7, ch8;
  //function paramter
  output [7:0] param;
  //blind mode flag
  output       blindmode_flag;
  //captured flags
  output [7:0] captured;


  //******REGISTERS*******
  reg                    signed [31:0] registers[31:0];
  /******SPECIAL REGISTERS*******
   zero register         : R0
   current cue address   : R1
   previous basic data   : R2
   previous ch1-ch4      : R3
   previous ch5-ch8      : R4
   current basic data    : R5
   current extended data: R6
   next basic data       : R7
   current ch1-ch8       : R8-R15
   function parameter    : R16
   additional param      : R17
   captured channels     : R18
   blind mode flag       : R19
   illop instr (debug)   : R31
     */


  //output assignments
  //current cue number
  assign                cue_num = registers[1][8:2];
  assign                prev_num = registers[1][17:11];
  //previous cue basic data
```

```
assign                  prev_up = registers[2][31:24];
assign                  prev_down = registers[2][23:16];
assign                  prev_wait = registers[2][15:8];
//previous cue channels
assign                  prev_ch1 = registers[3][31:24];
assign                  prev_ch2 = registers[3][23:16];
assign                  prev_ch3 = registers[3][15:8];
assign                  prev_ch4 = registers[3][7:0];
assign                  prev_ch5 = registers[4][31:24];
assign                  prev_ch6 = registers[4][23:16];
assign                  prev_ch7 = registers[4][15:8];
assign                  prev_ch8 = registers[4][7:0];
//current cue basic data
assign                  cur_up = registers[5][31:24];
assign                  cur_down = registers[5][23:16];
assign                  cur_wait = registers[5][15:8];
//current cue extended data
assign                  cur_follow = registers[6][31:24];
assign                  cur_link = registers[6][23:17];
//next cue basic data
assign                  next_up = registers[7][31:24];
assign                  next_down = registers[7][23:16];
assign                  next_wait = registers[7][15:8];
//current channels
assign                  ch1 = registers[8][7:0];
assign                  ch2 = registers[9][7:0];
assign                  ch3 = registers[10][7:0];
assign                  ch4 = registers[11][7:0];
assign                  ch5 = registers[12][7:0];
assign                  ch6 = registers[13][7:0];
assign                  ch7 = registers[14][7:0];
assign                  ch8 = registers[15][7:0];
//function paramter
assign                  param = registers[16][7:0];
//blind mode flag
assign                  blindmode_flag = registers[19][0];
//captured flags
assign                  captured = registers[18][7:0];


//state
reg                    state;
//instruction buffer full
output                  full;
//control signals
wire                   empty, bre, instr_ready, illop;
wire                   werf;
//control signals
wire                   bsel, cmwe, ra2sel,mpcsel,wdsel;
//dmx control signals
output wire                start,update;
//alu control signal
wire [3:0]                 alufn;
//macro address
wire [7:0]                 maddr;
//previous macro address
reg [7:0]                  prev_maddr;
//instructions
output wire [31:0]             instruction;
wire [31:0]                binstr;
wire [31:0]                minstr;
//split up instruction
wire [31:0]                c;
wire [15:0]                sext;
//parameter register numbers
wire [4:0]                 ra,rb,rc;
//register file outputs
wire [31:0]                rd1,rd2;
//is ra 0?
wire                   z;
// ALU second parameter
wire [31:0]                param_b;
//ALU output
wire [31:0]                alu_out;
//memory out
wire [31:0]                mem_out;
//write data
wire [31:0]                wd;

//instruction buffer
instr_buffer buff(.clk(clk), .sinit(reset),
                   .din(instr_in),.wr_en(bwe),.rd_en(bre),
                   .dout(binstr),.full(full),.empty(empty),
                   .rd_ack(binstr_ready));

//macro pc
assign                  maddr = (mpcsel) ? instruction[7:0] : (prev_maddr+1);
//(MACROS)TODO: get rid of control signals
macros  m_inst(maddr,clk,minstr);

//select valid instruction signal
assign                  instr_ready =  (state) ? 1 : binstr_ready;

//select instruction out
assign                  instruction =  (state) ? minstr : binstr;

//** split up instruction

//replicate highest bit of constant
assign                  sext = {16{instruction[15]}};
//make 32 bit constant
assign                  c = {sext,instruction[15:0]};
```

```
  //register selections
  assign                  ra = instruction[20:16];
  assign                  rb = instruction[15:11];
  assign                  rc = instruction[25:21];

  //value out of the reg file
  assign                  rd1 = registers[ra];
  assign                  rd2 = (ra2sel) ? registers[rc] : registers[rb];

  //is rd1 0
  assign                  z = (rd1==0);

  //control logic
  ctrl_logic decode(.reset(reset),.opcode(instruction[31:26]),
                     .empty(empty),.instr_ready(instr_ready),
                     .state(state),.z(z),
                     //output control signals
                     .werf(werf),.bsel(bsel),.wdsel(wdsel),
                     .cmwe(cmwe),.ra2sel(ra2sel),
                     .mpcsel(mpcsel),.bre(bre),.illop(illop),.alufn(alufn),
           .start(start),.update(update));

  //select sign-extended constant or reg file value for second parameter
  assign                  param_b = (bsel) ? c : rd2;

  //alu
  alu arthunit(.param_a(rd1), .param_b(param_b), .alufn(alufn), .out(alu_out));

  //cue memory
  cue_mem mem(alu_out[8:0],clk,rd2,mem_out,cmwe);

  //register file write data
  assign                  wd = (illop) ? instruction : ((wdsel) ? mem_out : alu_out);

  //counter for register instantiation
  integer                 index;
  //sequential logic
  always @ (posedge clk) begin
    if(reset) begin
      state<=0;
      for (index=0;index<32;index=index+1)
        registers[index]<=0;
    end
        //switch to macro mode for illop handling
        if (illop)
          state<=1;

        if(instr_ready) begin
            //switch to macro mode on macro instruction
            if (instruction[31:26]==6'b001000)
              state<=1;
            //switch to buffer mode on return instruction
            if (instruction[31:26] == 6'b001001)
              state<=0;
        end

        //save previous macro address
    prev_maddr <= maddr;

        //write to reg file
    if(werf)
          if (rc!=0)
      registers[(illop) ? 31 : rc] <= wd;
  end
endmodule
```

6.5.2 Control Logic

```
//                                              -*- Mode: Verilog -*-
// Filename         : ctrl_logic.v
// Description      : Takes in an opcode and sets all of the control signals

`timescale 1ns / 1ps

module ctrl_logic(reset,opcode,empty, instr_ready,state,z,
                  //output control signals
                  werf,bsel,wdsel,cmwe,ra2sel,
                  mpcsel,bre,illop,alufn,start,update);

   input [5:0]  opcode;
   input        reset,empty,instr_ready,state,z;

   //control signals
   output    reg werf,cmwe,illop;
   output       mpcsel,bre,bsel,ra2sel,wdsel;
   output [3:0]  alufn;
   output       start,update;

   //address selection for the macros ROM
   //0 by default, 1 for macro and branch instructions
   assign       mpcsel = (opcode == 6'b001000) ? 1: (opcode==6'b001010) ? ~z : (opcode==6'b001011) ? z : 0;
   //instruction buffer read enable
   //1 if not executing a macro and the buffer isn't empty, 0 otherwise
   assign       bre = (state) ? 0 : (empty) ? 0 : 1;
   //select whether to use the value from the register file
   //or the sign extended constant as the second operand in operations
   assign       bsel = opcode[4];
   //selects which register to get the second instruction from
   //usually ra, but rc on a store instruction
   assign       ra2sel = (opcode == 6'b011001);
   //selection of where the data out comes from
   //usually the alu, except on load instructions
   //when it comes from the memory
   assign       wdsel = (opcode == 6'b011000);
   //alu operator selection
   //usually the last 4 bits of the opcode, except load and store
   //which both require add
   assign       alufn = ((opcode==6'b011000) | (opcode==6'b011001)) ? 0 : opcode[3:0];
   //start signal to the dmx module
   //only high on a start instruction
   assign       start = (opcode == 6'b010000);
   //update signal to the dmx module
   //only high on an update instruction
   assign       update = (opcode==6'b010001);

   always @ (reset or opcode) begin
         illop = 0;

      //if in buffer mode and buffer is empty
      //lock down processor and idle
      if (~instr_ready) begin
        werf = 0;
        cmwe = 0;
      end
      else begin
        //write enable register file
        //true for any arithmetic operation or LD
        werf =  (opcode[5] | (opcode == 6'b011000));
        //cue memory write enable
        //only true for store
        cmwe = (opcode == 6'b011001);

               //illop checks
        if (opcode[5])
                if ((opcode[1:0]==2'b11)|(opcode[3:0]==4'b0010)) begin
                     illop=1;
                     werf=1;
                end
                else begin
                     if ((opcode[5:3]==3'b000)|(opcode[5:2]==4'b0011)| (opcode[5:1]==5'b01001) | (opcode[5:2]==4'b0101)) begin
                          illop = 1;
                          werf = 1;
                     end
                     if (opcode[5:2]==4'b0011) begin
                          illop = 1;
                          werf = 1;
                     end
                end // else: !if(opcode[5])
          end // else: !if(~instr_ready)
   end
endmodule
```

6.5.3 ALU

```
//                                          -*- Mode: Verilog -*-
// Filename          : alu.v
// Description       : Arithmetic logic unit for the processor,
// takes two 32-bit inputs and a operation selector, and outputs
// the 32-bit result of the operator applied to the operands

`timescale 1ns / 1ps

module alu(param_a, param_b, alufn, out);
  input signed [31:0] param_a, param_b;
  input [3:0]       alufn;
  output reg signed [31:0] out;

  //operand a shifted by operand b, either sign extended or with 0s
  wire [31:0]          shifted;

  //module to right shift, written by Prof. Terman
  shift_right shifter(.sxt(alufn[1]),.a(param_a),.b(param_b[4:0]),
            .shift_right(shifted));


  always @ (param_a or param_b or alufn or shifted) begin
    //operand selection
    case(alufn[3:0])
      //ADD
      4'b0000: out = param_a + param_b;
      //AND
      4'b1000: out = param_a & param_b;
      //OR
      4'b1001: out = param_a | param_b;
      //SHL
      4'b1100: out = param_a << param_b;
      //SHR
      4'b1101: out = shifted;
      //SRA
      4'b1110: out = shifted;
      //SUB
      4'b0001: out = param_a - param_b;
      //XOR
      4'b1010: out = param_a ^ param_b;
      //CMPEQ
      4'b0100: out = (param_a==param_b);
      //CMPLT
      4'b0101: out = (param_a<param_b);
      //CMPLE
            4'b0110: out = (param_a<=param_b);
            default:
              out = param_b;
          endcase
    end
endmodule
```

6.5.4 Right Shift

```
//                              -*- Mode: Verilog -*-
// Filename        : shift_right.v
// Description     : Module written by Prof. Terman for
// handling right shifts in the processor (unchanged from original)

`timescale 1ns / 1ps

module shift_right(sxt,a,b,shift_right);
  input sxt;
  input [31:0] a;
  input [4:0] b;
  output [31:0] shift_right;

  wire [31:0] w,x,y,z;
  wire sin;

  assign sin = sxt & a[31];
  assign w = b[0] ? {sin,a[31:1]} : a;
  assign x = b[1] ? {{2{sin}},w[31:2]} : w;
  assign y = b[2] ? {{4{sin}},x[31:4]} : x;
  assign z = b[3] ? {{8{sin}},y[31:8]} : y;
  assign shift_right = b[4] ? {{16{sin}},z[31:16]} : z;
endmodule
```

6.6 DMX Controller Verilog Files

This section contains the Verilog files that control the DMX.

6.6.1 Channel Twiddler

```
//                                        -*- Mode: Verilog -*-
// Filename        : channeltwiddler.v
// Description     : Intensity control module for one channel.
// This module stores the current intensity level for one dmx channel.
// On an update signal the module switches the channel over to the new intensity level.
// On a start signal the module gradually changes the intensity to the new value,
// over the period inputted into the module.

`timescale 1ns / 1ps

module channeltwiddler(clock_20mhz,reset,start,update,period,ch,cur,loading);

  //module clock at 20mhz
  input clock_20mhz;
  //reset button
  input reset;
  //start signal for the start of an up or down time
  input start;
  //signal to update the channel intensity to a new value
  input update;

  //new value of channel (0-100)
  input [7:0] ch;
  //delta period for up or down time
  input [7:0] period;

  //current value of channel
  output reg [7:0] cur;


  //start the timer to count until time for next increment
  reg          start_timer;
  //flag indicating currently loading cue
  output reg      loading;

  //value we starting from during an up or down time
  reg [7:0]       start_value;
  //expired signal from timer indicating
  // that its time to increment the intensity value
  wire         expired;

  //period between 1 intensity level increments
  wire [32:0]     count_to;
  //absolute value of the difference between
  //the current value of the channel and the end value
  wire [7:0]      diff;
  //new value (0-255)
  wire [7:0]      next;
  //intermediate value in calculating the period between increments
  wire [10:0]     per_second;

  //convert 0-100 scale to 0-255 scale
  assign         next = ((ch[6]) ? 163:0) + ((ch[5]) ? 82:0) + ((ch[4]) ? 41:0) + ((ch[3]) ? 20:0)
      + ((ch[2]) ? 10:0) + ((ch[1]) ? 5:0) + ((ch[0]) ? 3:0);

  //find difference between new and old values
  assign         diff = (next>start_value) ? next-start_value : start_value-next;

  // (2025/|new value - old value|)
  division d(11'b11111101001, diff, per_second, remd, clock_20mhz, rfd,1'b0,1'b0,1'b1);

  //multiply by period and factor for 20.25mhz clock
  assign         count_to = period*10000*per_second;

  //timer to count between increments
  timer load_timer(clock_20mhz,start_timer,reset,expired,count_to);
  defparam           load_timer.SIZE=33;


  always @ (posedge clock_20mhz) begin
        start_timer <=0;

    //just set current value to new value if reset or update
    if ((reset) || (update)) begin
          cur<=next;
            start_value <=1;
            loading <=0;
        end

    //if start uptime or downtime
        if (start) begin
      //if no difference or uptime/downtime is 0,
      //just set current value to new value
      if ((diff==0) || (period==0))
        cur<=next;
      //otherwise start loading the cue
      else begin
        start_value<=cur;
              loading <=1;
              start_timer <=1;
      end
    end // if (start)
```

```
    //if time to increment
    if ((expired) && (loading)) begin
            //if the current value is the right one
      if (cur==next) begin
        //stop twiddling!!
                loading<=0;
        start_value<=1;
      end
      //otherwise, increment by 1 and start the timer again
            else begin
                cur <= (cur>next) ? cur-1:cur+1;
                start_timer<=1;
            end
        end
  end // always @ (posedge clk)
endmodule
```

6.6.2 DMX output

```
//                                  -*- Mode: Verilog -*-
// Filename         : dmxcontroller.v
// Description      : Changes the current channel values into the dmx serial protocol format.

`timescale 1ns / 1ps

module dmxcontroller(clk,reset,ch1,ch2,ch3,ch4,ch5,ch6,ch7,ch8,bit_out);

  //250khz clock
  input clk;
  //reset button
  input reset;

  //serial output
  output reg bit_out;

  //state 0: break space
  //state 1: mark after break
  //state 2: data
  //state 3: mart before break
  reg [1:0]     state;
  //bit counter, 4us/bit
  reg [5:0]     counter;
  //slot counter, used when tranmitting data in state 2
  reg [3:0]     slot;

  //current channel intensities
  input [7:0]   ch1,ch2,ch3,ch4,ch5,ch6,ch7,ch8;
  //holds channel intensities, only updated on every packet
  reg [7:0]     chans[7:0];

  //output logic
  always @ (posedge clk) begin
    if(reset) begin
            state<=0;
            counter<=0;
            bit_out<=0;
        end
        else begin
            counter<=counter+1;
            case (state)
                //break (low for 44*4us = 176us)
                0: begin
                    if (counter<44)
                      bit_out<=0;
                    else begin
            state<=1;
                        bit_out<=1;
                        counter<=0;
            chans[0] <= ch1;
            chans[1] <= ch2;
              chans[2] <= ch3;
              chans[3] <= ch4;
              chans[4] <= ch5;
              chans[5] <= ch6;
              chans[6] <= ch7;
              chans[7] <= ch8;
                    end
                end
                //mark after break (high for 52*4us = 208us)
                1: begin
                    if (counter<52)
                      bit_out<=1;
                    else begin
                        state<=2;
                        bit_out<=0;
                        counter<=0;
                        slot<=0;
                    end
                end
                //data slots (each slot 20*4us = 80us)
        //slot 0 = null start code
        //slot 1-8 = channel intensities
                2: begin
                    if (counter<8)
                      if (slot==0)
            //output null start code
            bit_out<=0;
            else
            //output channel value
            bit_out<= chans[slot-1][counter];
                    else begin
            //mark after data
                        if (counter<20)
                          bit_out<=1;
                        else begin
              //break at the beginning of next slot
                            counter<=0;
                            slot<=slot+1;
                            if (slot==8) begin
                                state<=3;
                                bit_out<=1;
                            end
                            else
                              bit_out<=0;
                        end // else: !if(counter<11)
                    end // else: !if(counter<8)
                end // case: 2
                //mark before break (52*4us = 208us)
```

```
            3: begin
                if (counter < 52)
                    bit_out <=1;
                else begin
                    bit_out <=0;
                    state <=0;
                    counter <=0;
                end
            end
        endcase // case(state)
    end // else: !if(reset)
  end // always @ (posedge clk)
endmodule // dmx
```

6.6.3 Divider

```
//                                    -*- Mode: Verilog -*-
// Filename          : divider.v
// Description       : Divides any clock by any value
`timescale 1ns / 1ps


module divider(divided_clk, clk, reset_sync);
  parameter COUNT_SIZE = 26;
  parameter COUNT_TO = 19999999;

  input      clk, reset_sync;
  output reg divided_clk;

  reg [COUNT_SIZE-1:0] counter;

  always @(posedge clk)
    begin
      if(reset_sync)
        begin
        divided_clk <=1'b0;
        counter <=0;
      end
        else
      begin
        if(counter == COUNT_TO)
          begin
            divided_clk <= 1'b1;
            counter <= 0;
          end
        else
          begin
            divided_clk <= 1'b0;
            counter <= counter+1;
          end
      end
    end
endmodule
```

6.6.4 DMX controller

```
//                                    -*- Mode: Verilog -*-
// Filename          : dmx.v
// Description       : Controller for the dimmer box.
// This module stores the intensity value for each channel
// and formulates the serial signal that gets sent to the dimmer box

module dmx(clk,reset,ch1,ch2,ch3,ch4,ch5,ch6,ch7,ch8,waittime,uptime,downtime,update,start,
       bit_out,loading_flag,cur0, cur1, cur2, cur3, cur4, cur5, cur6, cur7);

   //20mhz clock
   input clk;
   //reset button
   input reset;
   //new channel values
   input [7:0] ch1,ch2,ch3,ch4,ch5,ch6,ch7,ch8;
   //wait, up and down time for the current cue transition
   input [7:0] waittime,uptime,downtime;
   //update and start signals from the processor
   input        update, start;

   //current intensity of each channel
   output [7:0] cur0;
   output [7:0] cur1;
   output [7:0] cur2;
   output [7:0] cur3;
   output [7:0] cur4;
   output [7:0] cur5;
   output [7:0] cur6;
   output [7:0] cur7;

   //flag indicating that the channel intensities are
   //gradually changing for an up or down time
   output reg [7:0] loading_flag;
   //serial bit for the dimmer box
   output         bit_out;
   //clock for the dmx serial protocol
   output wire     clock_250khz;
   //expired signal from the wait timer
   //indicating it is time to start the uptime
   wire          wait_expired;

   //clock for the serial output
   dividerdmx  dmx_clock(clock_250khz, clk, reset);

   //period for the wait time
   wire [32:0]       count_to;
   assign          count_to = waittime*20250000;
   timer wait_timer(clk, start, reset,wait_expired,count_to);
   defparam        wait_timer.SIZE = 33;

   //current value of each channel 0-255
   wire [7:0]        cur[7:0];
   //start signal for the loading of each channel
   wire [7:0]        start_timer;
   //flag indicating whether each channel is currently changing
   wire [7:0]        loading;
   //delta period for each channel
   wire [7:0]        period[7:0];

   //select the delta period: uptime if the channel is non-zero in the next cue
   assign          period[0] = (ch1==0) ? downtime : uptime;
   assign          period[1] = (ch2==0) ? downtime : uptime;
   assign          period[2] = (ch3==0) ? downtime : uptime;
   assign          period[3] = (ch4==0) ? downtime : uptime;
   assign          period[4] = (ch5==0) ? downtime : uptime;
   assign          period[5] = (ch6==0) ? downtime : uptime;
   assign          period[6] = (ch7==0) ? downtime : uptime;
   assign          period[7] = (ch8==0) ? downtime : uptime;

   //select when to start changing, right away if the channel is going down to zero
   //after the wait time if the channel is going up
   assign          start_timer[0] = ((ch1==0) || (waittime == 0)) ? start : wait_expired;
   assign          start_timer[1] = ((ch2==0) || (waittime == 0)) ? start : wait_expired;
   assign          start_timer[2] = ((ch3==0) || (waittime == 0)) ? start : wait_expired;
   assign          start_timer[3] = ((ch4==0) || (waittime == 0)) ? start : wait_expired;
   assign          start_timer[4] = ((ch5==0) || (waittime == 0)) ? start : wait_expired;
   assign          start_timer[5] = ((ch6==0) || (waittime == 0)) ? start : wait_expired;
   assign          start_timer[6] = ((ch7==0) || (waittime == 0)) ? start : wait_expired;
   assign          start_timer[7] = ((ch8==0) || (waittime == 0)) ? start : wait_expired;

   assign          cur0 = cur[0];
   assign          cur1 = cur[1];
   assign          cur2 = cur[2];
   assign          cur3 = cur[3];
   assign          cur4 = cur[4];
   assign          cur5 = cur[5];
   assign          cur6 = cur[6];
   assign          cur7 = cur[7];

   //set flag high for all of the wait time and loading time for each channel
   //used for screen display
   always @ (posedge start or negedge loading[0]) begin
     if (start)
       loading_flag[0] = 1;
     else
       loading_flag[0] = 0;
   end
   always @ (posedge start or negedge loading[1]) begin
     if (start)
```

```
      loading_flag[1] = 1;
    else
      loading_flag[1] = 0;
  end
  always @ (posedge start or negedge loading[2]) begin
    if (start)
      loading_flag[2] = 1;
    else
      loading_flag[2] = 0;
  end
  always @ (posedge start or negedge loading[3]) begin
    if (start)
      loading_flag[3] = 1;
    else
      loading_flag[3] = 0;
  end
  always @ (posedge start or negedge loading[4]) begin
    if (start)
      loading_flag[4] = 1;
    else
      loading_flag[4] = 0;
  end
  always @ (posedge start or negedge loading[5]) begin
    if (start)
      loading_flag[5] = 1;
    else
      loading_flag[5] = 0;
  end
  always @ (posedge start or negedge loading[6]) begin
    if (start)
      loading_flag[6] = 1;
    else
      loading_flag[6] = 0;
  end
  always @ (posedge start or negedge loading[7]) begin
    if (start)
      loading_flag[7] = 1;
    else
      loading_flag[7] = 0;
  end

  //intenstiy controller for each of the channels
  channeltwiddler ch1t(clk,reset,start_timer[0],update,period[0],ch1,cur[0],loading[0]);
  channeltwiddler ch2t(clk,reset,start_timer[1],update,period[1],ch2,cur[1],loading[1]);
  channeltwiddler ch3t(clk,reset,start_timer[2],update,period[2],ch3,cur[2],loading[2]);
  channeltwiddler ch4t(clk,reset,start_timer[3],update,period[3],ch4,cur[3],loading[3]);
  channeltwiddler ch5t(clk,reset,start_timer[4],update,period[4],ch5,cur[4],loading[4]);
  channeltwiddler ch6t(clk,reset,start_timer[5],update,period[5],ch6,cur[5],loading[5]);
  channeltwiddler ch7t(clk,reset,start_timer[6],update,period[6],ch7,cur[6],loading[6]);
  channeltwiddler ch8t(clk,reset,start_timer[7],update,period[7],ch8,cur[7],loading[7]);

  //formulate dmx serial signal
  dmxcontroller dmxc(clock_250khz,reset,cur[0],cur[1],cur[2],cur[3],cur[4],cur[5],cur[6],cur[7],bit_out);

endmodule
```

6.6.5 Timer

```
//                                   -*- Mode: Verilog -*-
// Filename          : timer.v
// Description       : a timer that times for a variable period

module timer(clk, start_timer, reset_sync, expired, count_to);

  parameter SIZE = 26;
  input     clk, reset_sync, start_timer;
  input  [SIZE-1:0] count_to;
  output reg     expired;

  reg [SIZE-1:0]    counter;

  always @(posedge clk)
    begin
      if(reset_sync || start_timer)
        begin
        expired <= 0;
        counter <= 0;
      end
        else
        begin
        if(counter == count_to)
          begin
            expired <= 1'b1;
            counter <= 0;
          end
        else
          begin
            expired <= 1'b0;
            counter <= counter+1;
          end
      end
    end

endmodule
```

6.7 Labkit Verilog Files

This section contains the miscellaneous Verilog files used to integrate the project in the labkit.

6.7.1 Debounce

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce (reset, clk, noisy, clean);
  parameter DELAY = 270000;   // .01 sec with a 27Mhz clock
  input      reset, clk, noisy;
  output     clean;

  reg [18:0] count;
  reg        new, clean;

  always @(posedge clk)
      if (reset)
    begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
    end
      else if (noisy != new)
    begin
      new <= noisy;
      count <= 0;
    end
      else if (count == DELAY)
    clean <= new;
      else
    count <= count+1;

endmodule
```

6.7.2 Follow TImer

```
//                                    -*- Mode: Verilog -*-
// Filename          : followtimer.v
// Description       : Timer for follow time. Only expires if the period is not 0

module followtimer(clk, start_timer, reset_sync, expired, cur_follow);
  //20mhz clock, reset and start signal
  input clk, reset_sync, start_timer;
  //follow period
  input [7:0] cur_follow;
  //expired signal
  output reg  expired;

  reg [32:0]  counter;
  wire [32:0] count_to;

  assign     count_to = cur_follow*20250000;

  always @(posedge clk) begin
    expired <= 0;
    counter <= counter+1;
    if(reset_sync || start_timer) begin
      counter <= 0;
    end
      else begin
      if(counter == count_to)
        if(cur_follow >0) begin
          expired <= 1'b1;
          counter <= 0;
        end
    end
  end
endmodule
```

6.7.3 Labkit

```
//                              -*- Mode: Verilog -*-
// Filename         : labkit.v
// Description      : Labkit module adapted from 6.111 website

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
         ac97_bit_clock,

         vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
         vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
         vga_out_vsync,

         tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
         tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
         tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

         tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
         tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
         tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
         tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

         ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
         ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

         ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
         ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

         clock_feedback_out, clock_feedback_in,

         flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
         flash_reset_b, flash_sts, flash_byte_b,

         rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

         mouse_clock, mouse_data, keyboard_clock, keyboard_data,

         clock_27mhz, clock1, clock2,

         disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
         disp_reset_b, disp_data_in,

         button0, button1, button2, button3, button_enter, button_right,
         button_left, button_down, button_up,

         switch,

         led,

         user1, user2, user3, user4,

         daughtercard,

         systemace_data, systemace_address, systemace_ce_b,
         systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

         analyzer1_data, analyzer1_clock,
         analyzer2_data, analyzer2_clock,
         analyzer3_data, analyzer3_clock,
         analyzer4_data, analyzer4_clock);

  output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
  input  ac97_bit_clock, ac97_sdata_in;

  output [7:0] vga_out_red, vga_out_green, vga_out_blue;
  output     vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
         vga_out_hsync, vga_out_vsync;

  output [9:0] tv_out_ycrcb;
  output     tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
         tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
         tv_out_subcar_reset;

  input [19:0] tv_in_ycrcb;
  input      tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
         tv_in_hff, tv_in_aff;
  output     tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
         tv_in_reset_b, tv_in_clock;
  inout      tv_in_i2c_data;

  inout [35:0] ram0_data;
  output [18:0] ram0_address;
  output      ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
  output [3:0]  ram0_bwe_b;

  inout [35:0]  ram1_data;
  output [18:0] ram1_address;
  output      ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
  output [3:0]  ram1_bwe_b;

  input       clock_feedback_in;
  output      clock_feedback_out;

  inout [15:0]  flash_data;
  output [23:0] flash_address;
  output      flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
  input       flash_sts;

  output      rs232_txd, rs232_rts;
  input       rs232_rxd, rs232_cts;
```

```
input        mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input        clock_27mhz, clock1, clock2;

output       disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input        disp_data_in;
output       disp_data_out;

input        button0, button1, button2, button3, button_enter, button_right,
       button_left, button_down, button_up;
input [7:0]   switch;
output [7:0]  led;

inout [31:0]  user1, user2, user3, user4;

inout [43:0]  daughtercard;

inout [15:0]  systemace_data;
output [6:0]  systemace_address;
output       systemace_ce_b, systemace_we_b, systemace_oe_b;
input        systemace_irq, systemace_mpbrdy;

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
       analyzer4_data;
output       analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign       beep= 1'b0;
assign       audio_reset_b = 1'b0;
assign       ac97_synch = 1'b0;
assign       ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// Video Output
assign       tv_out_ycrcb = 10'h0;
assign       tv_out_reset_b = 1'b0;
assign       tv_out_clock = 1'b0;
assign       tv_out_i2c_clock = 1'b0;
assign       tv_out_i2c_data = 1'b0;
assign       tv_out_pal_ntsc = 1'b0;
assign       tv_out_hsync_b = 1'b1;
assign       tv_out_vsync_b = 1'b1;
assign       tv_out_blank_b = 1'b1;
assign       tv_out_subcar_reset = 1'b0;

// Video Input
assign       tv_in_i2c_clock = 1'b0;
assign       tv_in_fifo_read = 1'b0;
assign       tv_in_fifo_clock = 1'b0;
assign       tv_in_iso = 1'b0;
assign       tv_in_reset_b = 1'b0;
assign       tv_in_clock = 1'b0;
assign       tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign       ram0_data = 36'hZ;
assign       ram0_address = 19'h0;
assign       ram0_adv_ld = 1'b0;
assign       ram0_clk = 1'b0;
assign       ram0_cen_b = 1'b1;
assign       ram0_ce_b = 1'b1;
assign       ram0_oe_b = 1'b1;
assign       ram0_we_b = 1'b1;
assign       ram0_bwe_b = 4'hF;
assign       ram1_data = 36'hZ;
assign       ram1_address = 19'h0;
assign       ram1_adv_ld = 1'b0;
assign       ram1_clk = 1'b0;
assign       ram1_cen_b = 1'b1;
assign       ram1_ce_b = 1'b1;
assign       ram1_oe_b = 1'b1;
assign       ram1_we_b = 1'b1;
assign       ram1_bwe_b = 4'hF;
assign       clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign       flash_data = 16'hZ;
assign       flash_address = 24'h0;
assign       flash_ce_b = 1'b1;
assign       flash_oe_b = 1'b1;
assign       flash_we_b = 1'b1;
assign       flash_reset_b = 1'b0;
assign       flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign       rs232_txd = 1'b1;
assign       rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs
```

```
// LED Displays
assign      disp_blank = 1'b1;
assign      disp_clock = 1'b0;
assign      disp_rs = 1'b0;
assign      disp_ce_b = 1'b1;
assign      disp_reset_b = 1'b0;
assign      disp_data_out = 1'b0;
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
assign      led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign      user1[30:0] = 31'hZ;
assign      user2 = 32'hZ;
assign      user3 = 32'hZ;
assign      user4 = 32'hZ;

// Daughtercard Connectors
assign      daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign      systemace_data = 16'hZ;
assign      systemace_address = 7'h0;
assign      systemace_ce_b = 1'b1;
assign      systemace_we_b = 1'b1;
assign      systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs


////////////////////////////////
// START OF LIGHTING BOARD CODE //
//////////////////////////////////



//make 40mhz clock
wire      clock_40mhz_unbuf,clock_40mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 4
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 6
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));

//***RESET BUTTON*******************************************************************
wire      reset;
debounce deb_reset(reset, clock_40mhz, ~button_enter, reset);

//***PIXELS*************************************************************************
//static sprite pixel
wire [2:0]    pixel;
//dynamic sprite pixel
wire [2:0]    pixel_d;
//output pixel
wire [2:0]    rgb;
assign      rgb = pixel | pixel_d;

//***SCREEN STUFF*******************************************************************
// generate basic XVGA video signals
wire [10:0]   hcount;
wire [9:0]    vcount;
wire      hsync,vsync,blank;

reg       clock_20mhz;

xvga xvga1(clock_40mhz,hcount,vcount,hsync,vsync,blank);

// feed XVGA signals to static sprites
wire      static_hsync,static_vsync,static_blank;
wire      d_hsync,d_vsync,d_blank;
reg       b,hs,vs;

// VGA Output.  In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
assign      vga_out_red = {8{rgb[2]}};
assign      vga_out_green = {8{rgb[1]}};
assign      vga_out_blue = {8{rgb[0]}};
assign      vga_out_sync_b = 1'b1;    // not used
assign      vga_out_blank_b = ~b;
assign      vga_out_pixel_clock = ~clock_40mhz;
assign      vga_out_hsync = hs;
assign      vga_out_vsync = vs;

always @(posedge clock_40mhz) begin
    hs <= d_hsync | static_hsync;
    vs <= d_vsync | static_vsync;
    b  <= d_blank | static_blank;

  //make 20mhz clock
  if(reset)
    clock_20mhz<=0;
  clock_20mhz<=~clock_20mhz;
end


//**WIRE DECLARATIONS***************************************************************

//Processor Wires
//macro parameter (used for typing display)
```

```
wire [7:0] param;
//previous channel values
wire [7:0] prev_ch1, prev_ch2, prev_ch3, prev_ch4, prev_ch5, prev_ch6, prev_ch7, prev_ch8;
//current channel values
wire [7:0] ch1, ch2, ch3, ch4, ch5, ch6, ch7, ch8;
//current cue number
wire [6:0] cue_num;
//previous cue number
wire [6:0] prev_num;
//uptime of previous cue
wire [7:0] prev_up;
//downtime of previous cue
wire [7:0] prev_down;
//wait time of previous cue
wire [7:0] prev_wait;
//uptime of current cue
wire [7:0] cur_up;
//downtime of current cue
wire [7:0] cur_down;
//link of current cue
wire [6:0] cur_link;
//follow of current cue
wire [7:0] cur_follow;
//wait time of current cue
wire [7:0] cur_wait;
//wait time of next cue
wire [7:0] next_wait;
//uptime of next cue
wire [7:0] next_up;
//downtime of next cue
wire [7:0] next_down;
//captured channels
wire [7:0] captured_flags;
//state of the keyboard module
wire      key_state;
//start loading a cue
wire      start_timer;
//update channel values
wire      update_dmx;
//current channel values on stage
//ch8
wire [7:0] loading_value7;
wire [7:0] loading_value6;
wire [7:0] loading_value5;
wire [7:0] loading_value4;
wire [7:0] loading_value3;
wire [7:0] loading_value2;
wire [7:0] loading_value1;
//ch1
wire [7:0] loading_value0;
//which channels are currently loading
wire [7:0] loading_flag;

// Keyboard wires
//current channel being editted
wire [2:0] channel;
//blind mode flag
wire      blind_mode;
//ascii from keyboard
wire [7:0] ascii;
//ready signal from keyboard
wire      char_rdy;
//error message code
wire [2:0] error;
//instruction buffer write enable
wire      bwe;
//instruction into processor
wire [31:0] instr_in;
//follow timer expired
wire     expired;

//DMX Wires
//dmx serial
wire     bit_out;
assign     user1[31] = bit_out;


//****KEYBOARD DECLARATIONS*****************************************************
ps2_ascii_input kbd(clock_20mhz, reset, keyboard_clock,
          keyboard_data, ascii, char_rdy);

keyboard_interp key(.clk(clock_20mhz),.ascii(ascii), .ascii_ready(char_rdy),.expired(expired),.cur_cue(cue_num),.blind_mode(bli
          .we(bwe), .instr_out(instr_in), .reset(reset), .param(param),.state(key_state),.error(error),.channel(channel));
//***KEYBOARD DECLARATIONS*****************************************************

//***FOLLOW TIMER DECLARATIONS**************************************************
followtimer timerf(clock_20mhz, start_timer, reset, expired, cur_follow);

//***PROCESSOR*****************************************************************
processor test(.clk(clock_20mhz), .reset(reset), .instr_in(instr_in), .bwe(bwe),
                  //current cue num
                  .cue_num(cue_num),.prev_num(prev_num),
                  //previous cue basic data
                  .prev_up(prev_up), .prev_down(prev_down), .prev_wait(prev_wait),
                  //previous cue channels
                  .prev_ch1(prev_ch1), .prev_ch2(prev_ch2), .prev_ch3(prev_ch3), .prev_ch4(prev_ch4),
                  .prev_ch5(prev_ch5), .prev_ch6(prev_ch6), .prev_ch7(prev_ch7), .prev_ch8(prev_ch8),
                  //current cue basic data
                  .cur_up(cur_up), .cur_down(cur_down), .cur_wait(cur_wait),
                  //current cue extended data
                  .cur_link(cur_link), .cur_follow(cur_follow),
                  //next cue basic data
```

```
                    .next_up(next_up), .next_down(next_down), .next_wait(next_wait),
                    //current cue channels
                    .ch1(ch1), .ch2(ch2), .ch3(ch3), .ch4(ch4), .ch5(ch5), .ch6(ch6), .ch7(ch7), .ch8(ch8),
                    //function paramter
                    .param(param),
                    .blindmode_flag(blind_mode),
                    //instruction buffer full
                    .full(full), .start(start_timer),.update(update_dmx),.captured(captured_flags),
            );
  //***PROCESSOR*****************************************************************

  //***DMX***********************************************************************
  dmx controller(.clk(clock_20mhz),.reset(reset),.ch1(ch1),.ch2(ch2),.ch3(ch3),
          .ch4(ch4),.ch5(ch5),.ch6(ch6),.ch7(ch7),.ch8(ch8),.waittime(prev_wait),
          .uptime(cur_up), .downtime(prev_down), .update(update_dmx), .start(start_timer),
          .bit_out(bit_out),.loading_flag(loading_flag), .cur0(loading_value0), .cur1(loading_value1),
          .cur2(loading_value2), .cur3(loading_value3), .cur4(loading_value4),
          .cur5(loading_value5), .cur6(loading_value6), .cur7(loading_value7));
  //***DMX***********************************************************************

  //***SPRITE DECLARATIONS*******************************************************
  dynamic_sp dyn1(.vclock(clock_40mhz),.hcount(hcount),.vcount(vcount),.hsync(hsync),.vsync(vsync),.blank(blank),
           .d_hsync(d_hsync),.d_vsync(d_vsync),.d_blank(d_blank),.pixel_d(pixel_d), .cur_cue_num(cue_num),
         .prev_num(prev_num),.prev_wait(prev_wait), .cur_wait(cur_wait),
         .ch1(ch1), .ch2(ch2), .ch3(ch3), .ch4(ch4), .ch5(ch5), .ch6(ch6), .ch7(ch7), .ch8(ch8),
         .prev_chan1(prev_ch1),.prev_chan2(prev_ch2), .prev_chan3(prev_ch3), .prev_chan4(prev_ch4),
         .prev_chan5(prev_ch5), .prev_chan6(prev_ch6), .prev_chan7(prev_ch7), .prev_chan8(prev_ch8),
         .prev_uptime(prev_up), .prev_downtime(prev_down), .cur_uptime(cur_up), .cur_downtime(cur_down),
         .cur_link(cur_link), .cur_follow(cur_follow), .next_uptime(next_up), .next_downtime(next_down),
         .next_wait(next_wait), .live_mode(blind_mode), .state(key_state), .channel(channel),
         .param(param), .error(error), .captured_flags(captured_flags), .loading_flag(loading_flag),
         .loading_value0(loading_value0),.loading_value1(loading_value1), .loading_value2(loading_value2),
           .loading_value3(loading_value3), .loading_value4(loading_value4), .loading_value5(loading_value5),
         .loading_value6(loading_value6), .loading_value7(loading_value7));

  static_sp static_sp1(clock_40mhz,hcount,vcount,hsync,vsync,blank,
              static_hsync,static_vsync,static_blank,pixel);
  //***SPRITE DECLARATIONS*******************************************************

endmodule
```