Stop and Go: Building the 6.111 Traffic Light Controller

Michael Salib

October 17, 2001

Abstract

I designed and implemented a traffic light controller. This controller is able to handle a walk request button on the main street and a traffic sensor on the side street. The controller includes a user interface allowing an operator to view and modify the configuration of the device. In addition, the controller properly synchronizes its external inputs making it robust to changes in user usage. The system is implemented using a CPLD and a static RAM. Control is provided by a finite state machine embedded in the CPLD. This report describes the requirements for the traffic light controller, as well as the design, implementation, and testing procedures used. Finally, I discuss possible improvements to my design and offer some suggestions for improving the design and debugging processes.

Contents

1 Overview **1**
1.1 Operational Specification . . . 1
1.2 Traffic Light Protocol . . . 2
1.3 User Interface Specification . . . 2
1.4 Use Case . . . 3

2 Description **3**
2.1 Input Handling . . . 4
2.2 Clock Divider . . . 5
2.3 Finite State Machine . . . 5
2.4 Timer . . . 7
2.5 Storage and Output . . . 7

3 Testing and Debugging **8**
3.1 Design Problems . . . 8
3.2 State Machine Problems . . . 9
3.3 Timing Problems . . . 9

4 Conclusions **10**
4.1 Design Improvements . . . 10
4.2 Process Improvements . . . 11

A VHDL Source Code Listings **11**
A.1 Traffic Light . . . 11
A.2 Clock Generator . . . 15
A.3 Divider . . . 16
A.4 Finite State Machine Controller . . . 17
A.5 Timer . . . 23

List of Figures

1 Finite State Machine Diagram . . . 6

1 Overview

For my second 6.111 lab, I designed and built a traffic light controller. The controller is designed to operate a standard traffic light; it has outputs corresponding to pairs of red, yellow, and green lights on a main and side streets and inputs for a walk request button and a vehicle sensor. In addition to operating the traffic light, the controller also has special modes for viewing and configuring the timing information for that particular light. This allows the behavior of the traffic light to be customized based on the locale and location its installed in. For example, a traffic light installed at the intersection of a heavily traveled road and a much smaller road could be configured to have a much longer "go period" for the heavily trafficed road.

For this report, I'll begin with a brief description of the traffic light controller requirements to motivate the design description. I'll also show how different requirements constrain the design in various ways. Once the problem has been motivated, I'll describe my solution, first by explaining the system architecture and then by drilling down to examine individual subsystems in detail. After describing the system, I'll discuss my testing methodology and the problems I had debugging my traffic light controller. Finally, in my conclusions, I'll review the problem and my solution with an eye to how I could improve my solution in the future.

The traffic light controller is designed to meet a complex specification. That specification documents the requirements that a successful traffic light controller must meet. It consists of an operations specification that describes the different functions the controller must perform, a user interface description specifying what kind of interface the system must present to users, and a detailed protocol for running the traffic lights. Each of these requirements sets imposed new constraints on the design and introduced new problems to solve. In this section, I will describe each of these three requirement sets in turn.

Although the specification is sufficient to describe a "correct" implementation, I found that constructing a use case for how an operator would use the traffic light controller helped unify and simplify the different requirements into a more coherent package. Furthermore, the use case served as a delivery test; I knew that my controller was finished when I could successfully walk it through the use case and verify that everything behaved correctly. By comparing my implementation's behavior at any time to the use case, I could determine exactly how much more work needed to be done. I describe the use case below.

1.1 Operational Specification

The specification for traffic light controllers is quite demanding, and made more so by the need to have units individually configured with different timing parameters. The specification mandates that the controller be programmed with four timing parameters (TYEL, TBASE, TEXTD, and TBLINK) specified in seconds. The unit must have a reading mode where an operator can view the values for these parameters as currently set in the controller's memory. The unit must also have a configuration mode where an operator can write new values for each of these parameters, one at a time, into the controller's memory.

In addition, the unit must have a special blinking mode in which it alternates between

lighting the main yellow and side red lights with lighting the main red and side yellow lights. Each blink period should last as many seconds as are specified in the TBLINK timing parameter. This mode is also to be entered into automatically when the unit detects a system fault or otherwise unexpected condition. Finally, the unit must have a mode in which it runs a standard traffic light pattern, keeping the different lights lit at different times according the protocol described below. In this mode, the controller relies on inputs from the side street traffic sensor and main street walk request button in addition to a standard protocol and preconfigured timing parameters.

INSERT SYSTEM INPUT/OUTPUT DIAGRAM HERE!!! (figure1)

1.2 Traffic Light Protocol

Another source of complexity is the protocol mandated by the specification for running the traffic light. The protocol specifies the sequence and manner in which the different traffic lights are lit. It mandates that in any given cycle, the main street must have a green light while the side street has a red light for two separate time intervals measured by the timing parameters TBASE and TEXTD. This is followed by a warning stage where the main light goes to yellow while the side light displays both red and yellow lights for a period specified by the timing parameter TYEL.

Before transitioning from allowing the main street traffic to go and allowing side street traffic to go, unit is supposed have a walk interval, if the walk request button has been pushed at any point during previous cycle. During a walk interval, the system lights red and yellow lights on both streets for a period of TEXTD seconds. If there has been no walk request during the previous cycle, the unit should immediately light the green light for the side street and the red light for the main street for a period of TBASE seconds. The unit is required to remain in this state for as long as the traffic sensor input indicates additional traffic on the side street. Afterward, the unit must light the yellow light on the side street and the red and yellow lights on the main street for a period of TYEL seconds in preparation for the next cycle.

1.3 User Interface Specification

The user interface for these operations consists of one HEX-LED display digit and a series of push buttons and switches. One pair of switches is used to select the function that the controller should next execute from among the four choices of reading a timing parameter from memory, writing a timing parameter to memory, running the traffic light pattern normally, and running the blink pattern (labeled "function" in Figure **??**). The controller doesn't actually begin executing in the specified mode until the go button is pressed. Only then does execution of the previous mode stop and execution of the newly selected mode begin.

In addition to the pair of switches used for selecting which mode to run in, the controller also must have a pair of switches to select which of the four timing parameters to view or change. When actually changing values in memory, a third set of switches is used to specify the new value to be stored. These four value switches are set according to the binary value

of the timing parameter. For example, if a user wanted to specify a timing parameter of six seconds, she would enable the second and third data switches while disabling the first and fourth. These address selection and value selection switches are only effective when using the memory read and memory write modes; in other modes they have no effect.

The unit also has inputs indicating when the walk request button has been pressed and when the vehicle sensor on the side street indicates traffic waiting there. Finally, the unit must be equipped with a reset button that resets the system controller to a known good state from which any mode can be safely entered.

1.4 Use Case

A typical usage scenario scenario looks like this. The operator powers on the device and presses the reset button in order to force the controller into a known good state. At this point, the controller memory hasn't been initialized and will contain random values, so the operator should set the timing parameters needed for this particular traffic light. She does so by setting the function selection switches to "write mode" and then for each timing parameter, writing the correct value. This is accomplished by pressing the run button after selecting the timing parameter number using the address switches and setting the proper value using the timing value switches. If any point the operator wishes to verify a particular setting, she can do so by pressing the go button after selecting which parameter she wants to view using the address switches and ensuring that the function switches are set to "read mode".

Once the timing parameters have been set and to the operator's satisfaction, she can test the unit's blink mode by setting the function switches to "blink mode" and pressing the go button. Similarly, to run the traffic light protocol, she can select "run mode" using the function switches and press go.

2 Description

In order to meet the strict requirements described above, I designed my controller as a collection of components. The heart of the system is a finite state machine (FSM) that directs the unit to light the main and side street lights at appropriate times for the specified time intervals. This unit depends on several inputs which are generated outside the system. In order to safely process these external inputs, I designed an input handler that synchronizes asynchronous inputs to the system clock. The input handler also latches some input signals and guarantees that other input signals will be single pulses, regardless of their duration. This pulsification greatly simplifies the design by ensuring that most external inputs are high for one and only one clock cycle.

In addition to the FSM and input handler, my design also includes a slow clock generator. Because the specification requires that timing parameters be specified in seconds, my controller needs to be informed every second that a second of real time has elapsed. The slow clock solves this problem by generating a slow clock pulse that is high for one cycle on the system clock during every second of real time. In addition to generating a once per second pulse, I need to be able to count down from a specified numbed of seconds. My timer

subsystem does that job. When given a particular number of seconds to count down from, it informs the FSM controller after exactly that number of seconds has elapsed.

Finally, I have storage and output components. In order to store the users timing parameters, I use a static RAM whose address and control lines are supplied by the FSM. The RAM data lines are on a tristate bus which is shared by the timer unit and a tristate enabled version of the the data value signals. This same bus is what drives the HEX-LED display, which comprises the output subsystem along with the actual traffic light LEDs. All of these components are described in detail in the sections below.

The design I've just described could be implemented in any number of ways. However, I chose to implement all components except for the RAM in a single Cypress 374i CPLD using VHDL. The RAM, switches, push buttons, HEX LED and traffic lights are wired to the CPLD's input and output pins using a series of buses.

A common implementation technique for this type of problem is to use a series of counters to generate a one second clock, and then to clock other parts of the system with that one second clock. However, because I chose to implement most of the system on a single CPLD, I was unable to do this. The different logic blocks that comprise the CPLDs in the 6.111 lab kit are all hard wired to the same clock, preventing me from generating a slow clock signal and using it to clock other logic blocks on the same CPLD.

Instead, I clocked all components with the 1.8 MHz system clock, and used a series of counters to generate a once per second enable pulse. I used this pulse as the enable signal to other components that needed a slow clock. This alternative design matched the hardware resources much better than the traditional design, allowing me to fit the entire system in a single CPLD. This design exploits the fact that CPLD flip flops are designed to accept enable signals from many sources, including those generated internally. The use of multiple clock signals is far more constrained since the CPLD must guarantee that all paths from clock drivers to flip flops impose uniform delays in order to minimize clock skew.

2.1 Input Handling

The VHDL code that implements this subsystem is located in the file trafficlight.vhd in Appendix A.1. This file is the top level file for the entire design. Because input handling consists of several small, unconnected tasks, I chose to implement this module as a set of processes. One process handles the the walk request signal by providing the system with a synchronized and latched version of the walk request input. This process also clears that internal signal whenever the FSM instructs it to reset it.

Another process is used to synchronize the go input. In addition to synchronizing the input with the system clock, this process also ensures that the synchronized go signal is high for at most one clock period. In other words, this process converts levels into pulses. Doing so makes the system behave correctly independent of how long the user presses the go push button. Finally, a third process synchronizes the reset and traffic waiting inputs.

2.2 Clock Divider

In order to generate the one second clock needed for effective operation of the traffic light, I needed to divide down the 1.8432 MHz system clock by a factor of 1,843,200. I did so using two modules called clockgen (see Appendix A.2) and divider (see Appendix A.3).

The clockgen unit uses the 1.8 MHz system clock to synthesize a 1.8 kHz pulse clock. In other words, it generates a signal that is high for one clock period out of every 1,800 clock periods. Originally, I had planned to use this signal as the clock signal for many other components in the system (hence the name "clock generator"). However, I've since found that to be unnecessary and instead use the system clock. This module is implemented with a 10 bit counter that counts down by one every clock cycle. Whenever the counter reaches zero, the slow clock output goes high for one cycle.

The divider module is used to further divide down the system clock. Starting with the 1.8 kHz pulse provided by the clockgen module, it generates an output signal that is high for exactly one period out of every 1,843,200 clock cycles. In other words, the divider module's output is high for one clock cycle per second. The divider contains an 11 bit counter that counts down from a fixed value. That value is 1800 in decimal (or "11100001000" in binary), which is exactly the value needed to generate one pulse every second.

2.3 Finite State Machine

The FSM is the heart of my traffic light controller. It responds to the input signals processed by the input handling module and provides the output and control signals needed to make the system function. My design uses a standard two process finite state machine where one process is used to change states on every clock cycle while the other process is used to combinatorically calculate what the next state should be based on the current inputs and the current state. That combinatorial process also sets the outputs for the next clock cycle. The VHDL source code for my FSM is listed in Appendix A.4. A state transition diagram is in Figure 1.

The FSM has four main groups of states corresponding to the four modes in which the traffic light controller can operate. The read function causes the controller to enter the memory read state. Once in that state, the system remains there until reset, using whatever value is on the timing parameter selection switches for the RAM address. The memory read state also ensures that write enable for the RAM is disabled since the system is only trying to read previously stored values in RAM.

The second major state group corresponds to the memory write function. In this mode, the FSM transitions to the memory write state and then returns to the start state. When in the memory write state, the system uses the value of the timing parameter selection switches for the RAM address lines as in the memory read state, but asserts the memory write enable control signal. This ensures that the new value is actually written to RAM. One crucial feature of this design is that the system is only in the memory write state for one cycle; thus the RAM write enable is never high for more than a single clock cycle. That ensures that we never write to the RAM while either data or address values are changing.

The third major group of states correspond to the blink mode. In this mode, the FSM bounces between the states blink main and blink side, alternately lighting the yellow and red

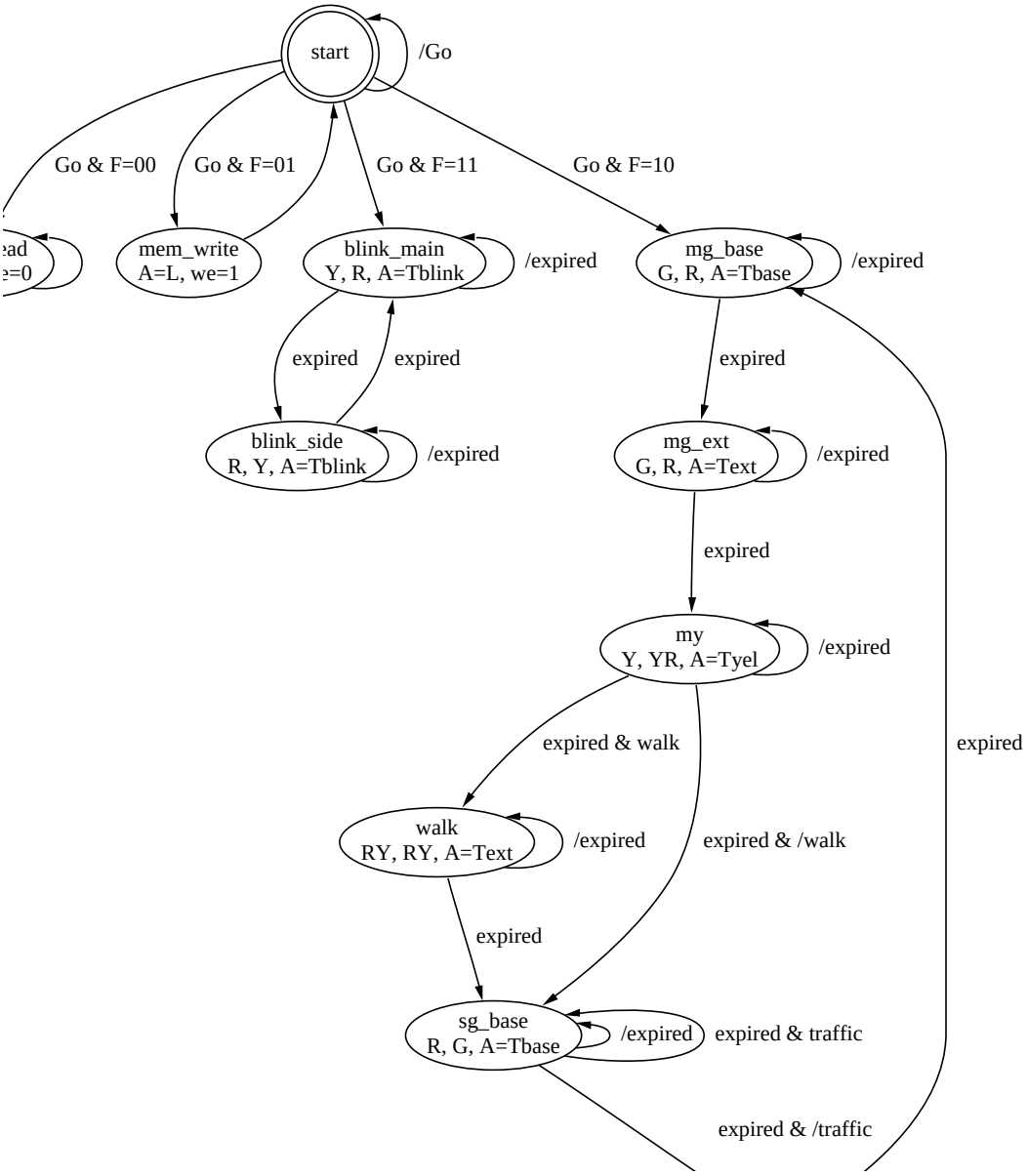

Figure 1: Finite State Machine Diagram

lights on both streets. Both states set the RAM address to the value needed for the blink interval timing parameter, thus giving the timer access to that number. Control remains in either state until the timer indicates that the amount of time specified by the RAM has expired. When that happens, control transfers to the other blink state and the process repeats indefinitely.

The final set of states corresponds to normal operating mode of the traffic light controller. In this mode, the FSM traverses a loop of states,. Control originates in the main green state and remains there for a period of time equal to TBASE before transferring to the main green extended state for TEXTD seconds. Thus, at the beginning of every cycle, the FSM keeps the main street green light and side street red light lit for a timer period equal to TBASE + TEXTD. Afterwards, the FSM enters the main yellow state for a period of TYEL seconds. If the walk request button has been pressed at any point during the cycle, the FSM will enter into the walk state. Otherwise, it enters the side green state.

In the walk state, all lights on both streets are lit. The FSM remains in this state for a period of time equal to TEXTD an clears the walk request latch. After the walk state, the FSM enters the side green state where it remains for a time period of TBASE. As long as the traffic sensor indicates additional side street traffic, the system will remain in the side green state. After either TBASE seconds have passed and the traffic sensor input goes low, the FSM enters the side yellow state and remains there for TYEL seconds. Afterwards the FSM returns to the main green state in order to repeat the cycle.

In the event the FSM encounters an unknown state, it enters the blinking mode since that mode of operation is intended to be used when a system fault occurs.

2.4 Timer

The timer module is responsible for counting down a specified number of seconds and informing the FSM when that allotted time has expired. The VHDL source code for this module is listed in Appendix A.5. The timer has a 4-bit input connected to the RAM data bus and produces an expired signal indicating when the amount of time specified by the RAM has elapsed. It relies on the once per second pulse that the enable signal provides in order to increment an internal count once every second. When that count reaches the value specified by the RAM, the timer resets its internal counter to zero and brings the expired output signal high.

This design allowed me to eliminate the starttimer signal from the timer module. Because the timer resets itself automatically to zero whenever it brings the expired signal high, this design doesn't have to worry about synchronizing the starttimer signal between the FSM and the timer module.

MENTION STARTTIMER / EXPIRED NASTINESS: when expired goes high, how do we manage to not have it kill the next state? INCLUDE TIMING DIAGRAM**********

2.5 Storage and Output

The storage and output module contains the HEX-LED used to display the alphanumeric value of the system's timing parameters and the RAM used to store those parameters.

Using the HEX-LED display on the 6.111 lab kit was quite simple. I used the pin numbers attribute in my top level VHDL source file (see Appendix A.1) to force the VHDL compiler to place the data lines for the RAM data bus on a particular set of pins. Those pins corresponded to the Nubus data lines A0 through A3. Once I enabled a few control signals on the Nubus, the 6.111 lab kit displayed the whatever value was on the RAM data bus using the first HEX-LED display. I used the inverted write enable signal I was generating for the RAM as the display clock. Whenever that signal went high, the RAM was writing data to the bus (i.e. a read cycle was being executed).

Interfacing to the RAM required more thought. In order to avoid violating the abstraction barrier, I programmed my FSM to have a RAM write enable output. That signal is later inverted before being routed to the RAM's inverted write enable input. This architecture allows the FSM to be relatively ignorant about how the RAM works and how to control it. Instead, the FSM code only needs to know when to tell the RAM to start writing.

Inherant in this architecture is the notion that whenever the RAM isn't being written to, it should be reading. In other words, the RAM should normally be outputting its data to the system bus except when its being written to; then it should store the data on the bus. This is accomplished by connecting the RAM's inverted chip select input to an inverted version of the clock, ensuring that the RAM is effectively always on. For those times when the RAM is being written to, the value associated with the timing parameter value switches must be written to the system bus. This is accomplished by seperating those switches from the system bus with a set of tristate buffers. Those buffers allow the value on the switches to be written to the system bus only when their enable signal is high. By connecting the tristate enable to the RAM write enable signal, we ensure that the switches only drive the system bus when the RAM is being written to. This prevents multiple drivers from driving the bus at the same time.

Beyond interface and implementation simplicity, the approach taken has another significant benefit. While running through the traffic light pattern, the value of the timing parameter currently being used is displayed in the HEX-LED. That feature makes testing and debugging easier and comes at no cost.

INCLUDE WIRING DIAGRAM OF RAM AND NUBUS/L1/L2

3 Testing and Debugging

During the course of designing and building this traffic light controller, I encountered several problems and errors. These included being unable to fit my design into the CPLDs, having my FSM transition into unexpected states at unexpected times, and having the system occasionally not see the expired signal generated by the timer module.

3.1 Design Problems

My original design was based on a pair of counters much like those found in the clockgen and divider modules. These counters would produce a clock signals that oscillated at a much lower frequency than the system clock. I planned on having most of the system being driven by the one kilohertz clock that came from the clockgen module while the timer was to be

sourced using the one hertz clock from the divider module. This design would have worked had I used LSI chips to implement it, but didn't work when placed into the CPLD. The problem was that the CPLD uses a very specialized path for clock signals to all of its flip flops. This makes it difficult to route internally generated signals to the clock inputs of other components on the same CPLD.

After unsuccessfully trying to get my original VHDL design to fit into one CPLD, I discovered that the last three CPLDs could use a clock signal generated by the first CPLD. I considered using this design, but found it too cumbersome. Instead, I opted to use the output of the clockgen and divider modules as enable signals. The CPLD hardware is able to route and generate flip flop enable signals much better than clock signals. To do so, I had to change the divider and clockgen modules to produce outputs that were pulses rather than levels, since the components that depended on them would need to be active for only once cycle out of many. These changes simplified the design and allowed the entire thing to fit in a single CPLD.

3.2 State Machine Problems

Initially, my FSM was appeared to transition into the wrong states at the wrong times. I attempted to add a test output to my design so that I could view the present state signal on the logic analyzer. But since I was using automatically generated state assignments at the time, I couldn't do so. Eventually, I rewrote my FSM code to use hard coded states assignments so that I could more easily view the current state signal using the logic analyzer. This helped a great deal. After doing this, I was able to fix several design and implementation bugs.

3.3 Timing Problems

The next major problem was in correctly handling the the expired and starttimer signals. I found that because the expired signal was updated only once per second, whenever it went high, it would force a state transition as expected. But because it remained high for considerable timer after, the next state to be entered would transition to it's next state as well. This process would repeat until enough time had passed for the expired signal to go low. In the meantime, the system would go crazy.

My solution to this problem was to have the FSM convert the expired signal from a level to a pulse internally. That way, regardless of how long the timer kept expired high for, the FSM would only see its internal version of expired high for exactly one clock cycle. The starttimer signal that the FSM generates to force the divider module to reset its count is simply a delayed version of the internal expired signal.

REFERENCE TIMING DIAGRAM FROM DESCRIPTION.TIMER WHEN TALKING ABOUT EXPIRED/STARTTIMER

4 Conclusions

In this report, I've described the requirements, design, and implementation process for my traffic light controller.

I started by describing the requirements for the traffic light controller. These requirements included the ability to store user entered parameters in a static RAM, a functional user interface allowing the system operator to examine and change the configuration of the system, and the ability to follow a complex lighting protocol whose exact behavior in any given iteration depended heavily on external inputs.

Those requirements, in turn, introduced several challenges. The reliance on dynamic external inputs necessitated the introduction of complex synchronization hardware. The requirement to use a static RAM necessitated the creation of hardware to generate RAM control signals. Because RAM timing is somewhat difficult, this also required additional debugging time. Likewise, the need to have the system operate in multiples of one second required a great deal of design and debugging time, in addition to a large amount of hardware resources.

The end result of my design, implementation, and debugging work is a controller implementation that satisfied all aspects of the use case described in section 1.4, and in general met all the of the specifications for this lab. However, both my design and the process used to generate and test it could be improved, so I'll discuss both of those issues next.

4.1 Design Improvements

Although I'm satisfied with my design, there are a number of improvements I would like to make now that I've completed the first version. Specifically, I would like to merge the clockgen and divider modules into one unit, clean up the existing architecture to better isolate components, remove the now unneeded testing code, and change signal names to better represent what is happening.

Merging clockgen and divider would simplify the system since the divider module is really the only system that needs the output of the clockgen module. Since they're relatively similar, combining them would be straightforward and make the system much easier to understand. In fact, the idea of separating the two seems to be an artifact of the traditional LSI design.

Additionally, the code base is filled with vestiges of old designs and testing features which are no longer necessary. During the debugging stage, I added a lot of code to output internal signals so that I could debug what was happening using the logic analyzer. Removing that extra cruft would simplify the code base considerably. Likewise, as the design evolved, I didn't change signal and component names as much as I should have to reflect the current state of the design. Renaming signals and entities would greatly improve code readability and hence, system maintainability.

4.2 Process Improvements

In addition to the design improvements I would make, there are several problems with the design process I noticed as well. Looking back, I found the most frustrating and time consuming part of this lab was the time debugging process. Even though I had fairly sophisticated tools at my disposal, like a digital oscilloscope and logic analyzer, I still spent far too long on debugging and testing. The main problem is that the only debugging tools available to me could only be used with hardware. This approach works fine for assembling simple systems out of a small number of LSI chips, but it falls apart for almost any other design, including any real world design. For most designs, a large component of the implementation will consist of VHDL code. Because the simulation tools I was given were of very, very poor quality, I was unable to adequately test and simulate my designs prior to burning them into the hardware. Once the design was in the hardware, I couldn't see inside of it to understand what was going wrong.

The only solution I found was to change my VHDL code to export some piece of internal state to an output pin so that I could view it using the logic analyzer. This approach was very limited though; I could only see a limited amount of information at a time and it was difficult to get the logic analyzer to catch suspicious events. The end result was that the debugging cycle of finding broken system behavior, tracing that behavior back to the bug that caused it, and trying different solutions to the bug took much longer than it should have. The extra time was spent changing source code specifically to expose internal state, waiting for access to the CPLD programmer, reprogramming the CPLD, wiring the newly exposed state to the logic analyzer, trying to get the logic analyzer to see the problematic event and then trying to reconstruct what was wrong with the code based on what the logic analyzer was showing.

Much of that extra time could have been avoided if I had the tools needed to test and simulate designs before burning them into hardware. CPLDs and FPGAs are the ultimate black box; the worst time to debug a design is once it has been burned into a CPLD. Having the right tools to do the right job is essential. Ten years ago when 6.111 labs consisted of wiring LSI chips on a protoboard, a logic analyzer was the right tool for debugging. Now, when many designs in the lab and almost all designs in the real world can't be debugged using a logic analyzer, decent simulation tools are the right tool needed for debugging. Coupled with a strong emphasis on proactive testing, they can dramatically reduce the time needed to complete projects. Had better design and simulation tools been available, I would have used them to build test benches for my components so that I could verify the design before burning it into hardware.

A VHDL Source Code Listings

A.1 Traffic Light

```
-- ram_data and expired in fsm
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.numeric_std.all;

library work;
use work.timer;
use work.divider;
use work.fsm_controller;
use work.clockgen;
--use work.ram;

entity traffic_light is
  port (
    location, func           : in std_logic_vector(1 downto 0);
    async_go, async_traffic  : in std_logic;
    async_walk, async_reset  : in std_logic;
    clk                 : in std_logic;
    switch_val               : in unsigned(3 downto 0);

    --must_be_zero : out std_logic;
    -- testing only
    --test :out std_logic_vector(6 downto 0);
    --clk_out : out std_logic;

    ram_addr       : out std_logic_vector(1 downto 0);
    not_we, not_ce : out std_logic;
    ram_data       : inout unsigned(3 downto 0);
    main, side_st  : out std_logic_vector(2 downto 0));  --in red, yellow,
                                                         --green order

  ATTRIBUTE pin_avoid of traffic_light :ENTITY is
    "11 21 22 32 42 43 44 53 63 64 74 83"& -- Vdd, Gnd, VPP
    " 13         "&     -- This is I0-9. Can screw up the clock of C1. Be
                          -- careful when using this.
    " 23 62 65 "&
    --" 71 "&                              --must be grounded for K1 interface

    -- this line lists all the logic analyzer connections...
    --"3 4 5 6 7 8 9 10 15 16 17 18 67 68 69 70 71 75 76 77 78 79 80 81 82"&

    " 14 35 41 51 72 "; -- Used by Programmer. No external connection.

  attribute pin_numbers of traffic_light :entity is
    "location(0):28 location(1):29 func(0):30 func(1):31 async_go:40 "&
    "async_traffic:37 async_walk:38 async_reset:39 switch_val(0):45 switch_val(1):46 swi
    "ram_addr(0):52 ram_addr(1):54 not_we:55 not_ce:56 "&
```

```
    "ram_data(0):24 ram_data(1):25 ram_data(2):26 ram_data(3):27 clk:20 "&
    "main(0):3 main(1):4 main(2):5 side_st(0):6 side_st(1):7 side_st(2):8 ";
    --"must_be_zero:71 "&
    -- these are only for testing purposes
    --"test(6):75 test(5):76 test(4):77 test(3):78 test(2):79 test(1):80 test(0):81";


end traffic_light;

architecture x of traffic_light is
  signal walk_reset, x, y, z : std_logic;
  signal traffic, internal_not_we : std_logic;
  signal starttimer, expired, always_one : std_logic;
  signal go, walk_request, enable, timer_clk, reset : std_logic;
  signal the_state_x : std_logic_vector(3 downto 0);
  signal out_time : unsigned(3 downto 0);

begin  -- x
--  test(4) <= '1';
  -- test(4) <= the_state(4);

--   test(3) <= the_state_x(3);
--   test(2) <= the_state_x(2);
--   test(1) <= the_state_x(1);
--   test(0) <= the_state_x(0);
--   test(3) <= out_time(0);
--   test(2) <= out_time(1);
--   test(1) <= out_time(2);
--   test(0) <= out_time(3);

--   must_be_zero <= '0';
--   always_one <= '1';

  not_ce <= not(clk);
  not_we <= internal_not_we;
  tristate_out: process(internal_not_we, switch_val)
    begin
      if internal_not_we = '0' then
        ram_data <= switch_val;
      else
        ram_data <= "ZZZZ";
      end if;
    end process tristate_out;
```

```
  sync_traffic_reset: process(clk)
    begin
      if rising_edge(clk) then
        if enable = '1' then
          traffic <= async_traffic;
          reset <= async_reset;
        end if;
      end if;
    end process sync_traffic_reset;

  sync_go: process(clk, y, z)
    -- secret sauce to ensure that go is active for at most one clk cycle
    begin
--       if rising_edge(clk) then
--         if enable = '1' then
--           go <= async_go;
--         end if;
--       end if;
      if rising_edge(clk) then
x <= async_go;
y <= x;
      end if;
    end process sync_go;
  go <= x and (not y);



  latch_walk: process(clk)
    begin
      if rising_edge(clk) then
        if walk_reset = '1' then
          walk_request <= '0';
        elsif async_walk = '1' then
          walk_request <= '1';
        end if;
      end if;
    end process latch_walk;

  divider1 : clockgen port map (
    clk      => clk,
    slow_clk => enable);

  divider2 : divider port map (
    clk        => clk,
```

```
    starttimer => starttimer,
    slow_clk   => timer_clk,
    enable => enable);

  timer : timer port map (
    starttimer => starttimer,
    enable        => timer_clk,
    clk => clk,
    expired    => expired,
    fsm_enable => enable,
    ram_data   => ram_data,
    out_time   => out_time);

  fsm : fsm_controller port map (
    enable       => enable,
    F            => func,
    L            => location,
    go           => go,
    traffic      => traffic,
    walk_request => walk_request,
    clk          => clk,
    reset        => reset,
    l_expired      => expired,
    A            => ram_addr,
    not_we       => internal_not_we,
    starttimer   => starttimer,
    walk_reset   => walk_reset,
    main         => main,
    side_st      => side_st
    --the_state => the_state_x,
    --the_expired => test(6),
    --the_starttimer => test(5)
    );

end x;
```

A.2 Clock Generator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity clockgen is
```

```
  port (
    clk      : in  std_logic;
    slow_clk : out std_logic);
end clockgen;

architecture x of clockgen is
  signal count : unsigned(9 downto 0);  --9
begin  -- x
  countdown: process(clk)
    begin
      if rising_edge(clk) then
        count <= count - 1;
      end if;
    end process countdown;
  slow_clk <= '1' when count = 0 else '0';
end x;
```

A.3 Divider

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity divider is
  port (
    clk, starttimer, enable : in  std_logic;
    slow_clk        : out std_logic);
end divider;

architecture x of divider is
  signal count         : unsigned(10 downto 0);
  constant start_value : unsigned(10 downto 0) := "11100011111";
  signal internal_slow_clk, x, y : std_logic;
begin  -- x
  internal_slow_clk <= '1' when count = 0 else '0';

  pulsify_slow_clk: process(clk, internal_slow_clk)
    begin
      if rising_edge(clk) then
        x <= internal_slow_clk;
        y <= x;
      end if;
    end process pulsify_slow_clk;
```

```
  slow_clk <= not(y) and x;


  process(starttimer, clk)
  begin  -- process
    if (starttimer = '1') then
      count <= start_value;
    elsif rising_edge(clk) then
      if enable = '1' then
        if count = 0 then
          count <= start_value;
        else
          count <= count - 1;
        end if;
      end if;
    end if;
  end process;


end x;
```

A.4 Finite State Machine Controller

```
library ieee;
use ieee.std_logic_1164.all;

entity fsm_controller is
  port (
    F, L                       : in  std_logic_vector(1 downto 0);
    go, traffic, walk_request : in  std_logic;
    enable, clk, reset, l_expired       : in  std_logic;
    A                          : out std_logic_vector(1 downto 0);
    not_we, starttimer         : out std_logic;
    walk_reset                 : out std_logic;
    --mr, my, mg, sr, sy, sg  : out std_logic;
    exp : out std_logic;
    --the_state : out std_logic_vector(3 downto 0);
    --the_expired, the_starttimer : out std_logic;
    main, side_st              : out std_logic_vector(2 downto 0));
end fsm_controller;


architecture x of fsm_controller is
--   type StateType is (start, blink_main,
```

```
--                      blink_side, mg_base,
--                      mg_ext, my,
--                      walk, sg_base,
--                      sg_ext, sy, mem_read, mem_write );
--   attribute enum_encoding of StateType:
--     type is "0000 0001 0010 0011  0100 0101 0110 0111  1000 1001 1010 1011  1100 1101

  constant start : std_logic_vector(3 downto 0) := "0000";
  constant blink_main : std_logic_vector(3 downto 0) := "0001";
  constant blink_side : std_logic_vector(3 downto 0) := "0010";
  constant mg_base : std_logic_vector(3 downto 0) := "0011";

  constant mg_ext : std_logic_vector(3 downto 0) := "0100";
  constant my : std_logic_vector(3 downto 0) := "0101";
  constant walk : std_logic_vector(3 downto 0) := "0110";
  constant sg_base : std_logic_vector(3 downto 0) := "0111";

  constant sg_ext : std_logic_vector(3 downto 0) := "1000";
  constant sy : std_logic_vector(3 downto 0) := "1001";
  constant mem_read : std_logic_vector(3 downto 0) := "1010";
  constant mem_write : std_logic_vector(3 downto 0) := "1011";

--   constant start : std_logic_vector(3 downto 0) := "1100";
--   constant start : std_logic_vector(3 downto 0) := "1101";
--   constant start : std_logic_vector(3 downto 0) := "1110";
--   constant start : std_logic_vector(3 downto 0) := "1111";

  signal present_state, next_state : std_logic_vector(3 downto 0); --StateType;
  signal we                        : std_logic;
  signal x, y, z, aa, b, c : std_logic;
  signal expired : std_logic;

  constant Tyel   : std_logic_vector(1 downto 0) := "00";
  constant Tbase  : std_logic_vector(1 downto 0) := "01";
  constant Textd  : std_logic_vector(1 downto 0) := "10";
  constant Tblink : std_logic_vector(1 downto 0) := "11";

  constant func_mem_read  : std_logic_vector(1 downto 0) := "00";
  constant func_mem_write : std_logic_vector(1 downto 0) := "01";
  constant func_run       : std_logic_vector(1 downto 0) := "10";
  constant func_blink     : std_logic_vector(1 downto 0) := "11";
begin  -- x
  not_we <= not(we);
  --the_state <= present_state(3) & present_state(2) & present_state(1) & present_state(
```

```
  --the_expired <= expired;


  process(clk, enable, x, y, z, l_expired)
    begin
      if rising_edge(clk) then
          x <= l_expired;
          y <= x;
          z <= y;
      end if;
    end process;
  expired <= not(z) and y;


 process(clk, aa, b, expired)
   begin
     if rising_edge(clk) then
       aa <= expired;
       b <= aa;
       starttimer <= b;
       --the_starttimer <= b;
     end if;
   end process;

--  expired <= l_expired;
--  starttimer <= expired;

--   process(clk)
--   begin
--     if rising_edge(clk) then
--       starttimer <= expired ;
--       the_starttimer <= expired ;
--     end if;
--   end process;



  state_clk: process(clk, enable, reset)
  begin
    if rising_edge(clk) then
        if reset = '1' then
          present_state <= start;
        else
          present_state <= next_state;
```

```
      end if;
  end if;
end process state_clk;

state_comb: process(present_state, next_state, F, L,
                    go, traffic, walk_request, expired)
  begin
    case present_state is
      when start =>
        main <= "101"; --default
        side_st  <= "101"; --default
        we <= '0'; walk_reset <= '0';
        if go = '0' then
          next_state <= start;
          A <= "00";
        elsif F = func_mem_read then
          next_state <= mem_read;
          A <= L;
        elsif F = func_mem_write then
          next_state <= mem_write;
          A <= L;
        elsif F = func_run then
          next_state <= mg_base;
          A <= Tbase;
        else
          next_state <= blink_main;
          A <= Tblink;
        end if;

      when mem_read =>
        A <= L;
        we <= '0';
        next_state <= mem_read;
      when mem_write =>
        A <= L;
        we <= '1';
        next_state <= start;

      when blink_main =>
A <= Tblink;
        we <= '0'; walk_reset <= '0';
        main <= "010";
        side_st  <= "100";
        if expired = '1' then
```

```
    next_state <= blink_side;
  else
    next_state <= blink_main;
  end if;

when blink_side =>
  A <= Tblink;
    we <= '0';
  main <= "100";
  side_st <= "010";
  if expired = '1' then
    next_state <= blink_main;
  else
    next_state <= blink_side;
  end if;

when mg_base =>
    we <= '0';
    A <= Tbase;
    main <= "001";
    side_st <= "100";
    if expired = '1' then
      next_state <= mg_ext;
    else
      next_state <= mg_base;
    end if;

when mg_ext =>
    we <= '0';
    A <= Textd;
    main <= "001";
    side_st <= "100";
    if expired = '1' then
      next_state <= my;
    else
      next_state <= mg_ext;
    end if;

when my =>
    we <= '0';
    A <= Tyel;
    main <= "010";
    side_st <= "110";
    if expired = '1' then
```

```
            if walk_request = '1' then
              next_state <= walk;
            else
              next_state <= sg_base;
            end if;
          else
            next_state <= my;
          end if;

        when walk =>
            we <= '0';
            A <= Textd;
            main <= "111";
            side_st <= "111";
            walk_reset <= '1';
            if expired = '1' then
              next_state <= sg_base;
            else
              next_state <= walk;
            end if;

        when sg_base =>
            we <= '0';
            A <= Tbase;
            main <= "100";
            side_st <= "001";
            if expired = '1' then
              if traffic = '1' then
                next_state <= sg_base;  --changed from sg_ext
              else
                next_state <= sy;
              end if;
            else
              next_state <= sg_base;
            end if;

--         when sg_ext =>
--             we <= '0';
--             A <= Textd;
--             main <= "100";
--             side_st <= "001";
--             if expired = '1' then
--               next_state <= sy;
--             else
```

```
--                next_state <= sg_ext;
--              end if;

        when sy =>
            we <= '0';
            A <= Tyel;
            main <= "110";
            side_st <= "010";
            if expired = '1' then
              next_state <= mg_base;
            else
              next_state <= sy;
            end if;

        when others =>
          next_state <= blink_main;
      end case;
    end process state_comb;
end x;
```

A.5 Timer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity timer is
  port (
    starttimer, clk, enable : in  std_logic;
    ram_data              : in  unsigned(3 downto 0);
    fsm_enable : in std_logic;
    out_time : out unsigned(3 downto 0);
    expired         : out std_logic);
end timer;

architecture x of timer is
  signal  internal_time : unsigned(3 downto 0);
  signal internal_expired : std_logic;
begin  -- x
  --internal_expired <= '1' when (internal_time = 0) else '0';
  expired <= internal_expired;

  out_time <= internal_time;
```

```
  countdown: process(clk, starttimer, enable)
    begin
      if rising_edge(clk) then
--        if starttimer = '1' then
--          internal_time <= ram_data;
        if (enable = '1') then
          if (internal_time = ram_data) then
            internal_time <= "0000";
            internal_expired <= '1';
          else
            internal_time <= internal_time + 1;
            internal_expired <= '0';
          end if;
        end if;
      end if;
    end process countdown;
end x;
```