Digital Anti-Theft System for Automotive Vehicles

Ranbel Sun

3 November 2008

Abstract

More than one million vehicles are stolen in the United States each year, despite the inclusion of alarm systems in most commercial vehicles. Because standard car alarms are not sufficient to deter professional thieves, we have designed and implemented a vehicle theft prevention device incorporating both standard alarm functionality as well as an extra fuel pump feature, a distinct siren, and custom timing specifications. The alarm system uses various switches and sensors located on the vehicle to automate and secure the arming and disarming of the vehicle: the doors are equipped with sensors to detect open and close, the ignition and brake are monitored, and there is an additional hidden switch. In case the ignition key is acquired by a thief, the fuel pump power will not turn on unless a specific combination of switches is pressed. The siren cycles through a series of 4 tones which correspond to notes in the C-major arpeggio. The design also enables the user to reprogram the system timing parameters to custom values using a set of six switches.

Contents

List of Figures

List of Tables

1 Overview

According to the Federal Bureau of Investigation (FBI)'s 2007 crime report, there are approximately one million thefts of motor vehicles in the United States each year. [1] Cars are a natural target because they are valuable, easy to resell, and provide quick getaways, thus motivating the need for effective anti-theft systems. Although modern commercial vehicles are often sold with some degree of theft protection, the alarm units are standard and may easily be disabled. This design strives to address this problem by not only implementing basic functionality in different ways but by also adding new security features and allowing for user customization.

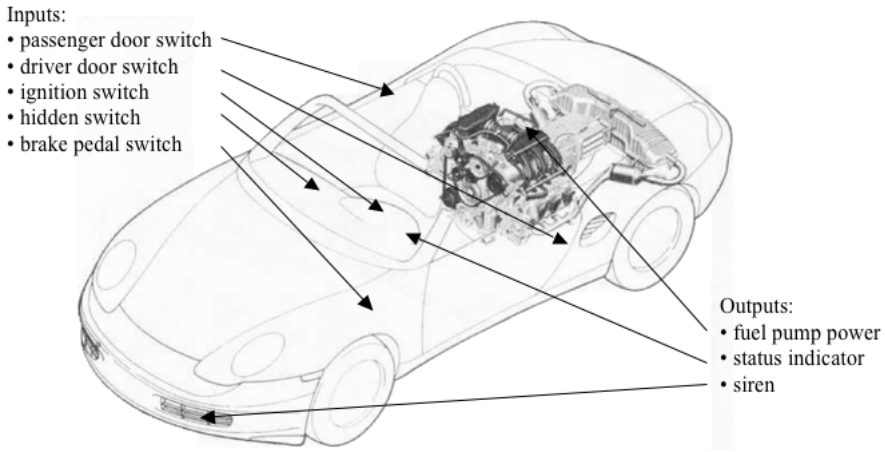

Figure 1: Sensors and actuators
Locations of vehicle sensors (inputs) and actuators (outputs) used in system. [2]

Features The anti-theft system is highly automated by relying on several sensors installed on the vehicle. A diagram of the relevant input and output devices is shown in Figure 1. Using a combination of ignition and door switches, the built-in controller determines when to sound a siren and provides information about the state of the system via the indicator light located on the dashboard. In addition to this standard alarm functionality, there is a hidden mechanism to regulate the power to the fuel pump.

The system is armed by default, which is indicated by a blinking light on the dashboard. After either door is opened and the ignition is not turned on within a specified time interval, a loud siren will sound to direct attention to the car and frighten away potential thieves. The siren stays on while the door remains open. The status light goes from blinking to constantly on when the door stays open, meaning that the alarm is either triggered or on. The siren tone is distinct in order to distinguish the vehicle's alarm from those of other vehicles nearby. The alarm continuously cycles through the 4 notes of the C-major arpeggio at a rate of 1 note per second. Therefore, the sequence will be middle-C, E, G, tenor-C, G, E, and so forth.

When the door is closed, the alarm continues to sound for a specified amount of time and then automatically shuts off and goes back to the default armed state. If the ignition is turned on before the siren sounds, the siren is disabled. To turn off the blaring siren, make sure to close all doors and then either turn on the ignition or wait for the automatic shutdown. Simply turning on the ignition is not enough to start the vehicle, however, due to an additional embedded security feature. In case the ignition key is stolen or duplicated, the car cannot be driven off unless the fuel pump power is

activated with a special switch sequence. The first step is to turn on the ignition. At this point, the entire anti-theft system should be disarmed and the status light off. Finally, press the brake pedal and hidden switch simultaneously and the fuel pump should then remain on as long as the ignition is on. After successfully starting the car and reaching the destination, turning off the ignition also shuts off the fuel pump power.

Now that the ignition is shut off, the anti-theft system rearms itself when everyone has exited the vehicle. That is to say, the status light should be blinking a specified number of seconds after the driver's door has opened and then all doors closed. If the driver were to suddenly reopen and reclose the door before the timer has expired (one could imagine a belonging was forgotten on the seat), the delay is restarted. Similarly, the delay will reset if the ignition is turned on before the system has rearmed.

It may occassionally be desirable to customize the timing delays described above. For example, if the driver first opens the door for a passenger, it takes longer to walk around to the driver's side, start the ignition, and thus disable the alarm. To this end, there are four time delays which may be reprogrammed: the delay to arm the system after exiting the car, the delay until the siren turns on after opening the driver door, the delay for the passenger door, and the length of time during which the alarm sounds. The default values are 6, 8, 15, and 10 seconds respectively. The parameters are specified in 1-second intervals with a maximum value of 15 seconds. Reprogramming the values involves selecting the desired interval, selecting the desired value for that interval, and then pressing the Reprogram button. This will also set the system back to the default armed mode.

2 Description

The anti-theft system was built by interconnecting several independent modules. The modules generally fall into one of three categories: anti-theft logic, timing, or input/output processing. The anti-theft logic consists of the main alarm system controller and the fuel pump power regulator, timing takes care of delay countdowns and parameter-setting, and input/output processing involves reading imperfect sensor inputs and generating the siren and approriate status lights. The schematic in Figure 2 illustrates all of the various modules and their overall organization. The module blocks are shaded according to category, and the arrows indicate the inputs and outputs of each. Each module runs off of a single 27 mHz clock so that everything will be in sync. The system was programmed onto a field-programmable gate array (FPGA) chip using the Verilog digital hardware description language (see Appendix for more details). This section will discuss the design considerations and implementations of the individual modules without getting too much into technical details.

Anti-theft Logic

Alarm system The alarm system was designed and implemented as a finite state machine (FSM), because the functionality followed a logical sequence of events depending on the current state of the system and the next sensor inputs. As seen in Figure 2, the inputs to this module are the driver door (DD), passenger door (PD), ignition switch (Ignition), reprogram button (Reprogram), and timer expiration signal (Expired), all of which are simple on or off signals. The module outputs on or off signals to the siren generator (Siren) and timer modules (Start_timer), a mode selector to the status light module (Status), and a time interval selector to the time parameters module (Interval). Figure 3 shows a simplified FSM diagram for the alarm functionality. The four major modes of operation are displayed in gray boxes, while the transitional states are in white.

Originally, the model assigned all outputs as a pure function of state such that a system in a specific state would always assert the designated outputs. For example, in the "Sound_alarm" state, Siren=ON, Status= ON, Interval= T_alarm_on, and Start_timer= TRUE. However, because the delay countdown only needs to be started once, continously asserting Start_timer, i.e. once every clock cycle (37 ns), would have meant that the timer would never expire. This potential problem was

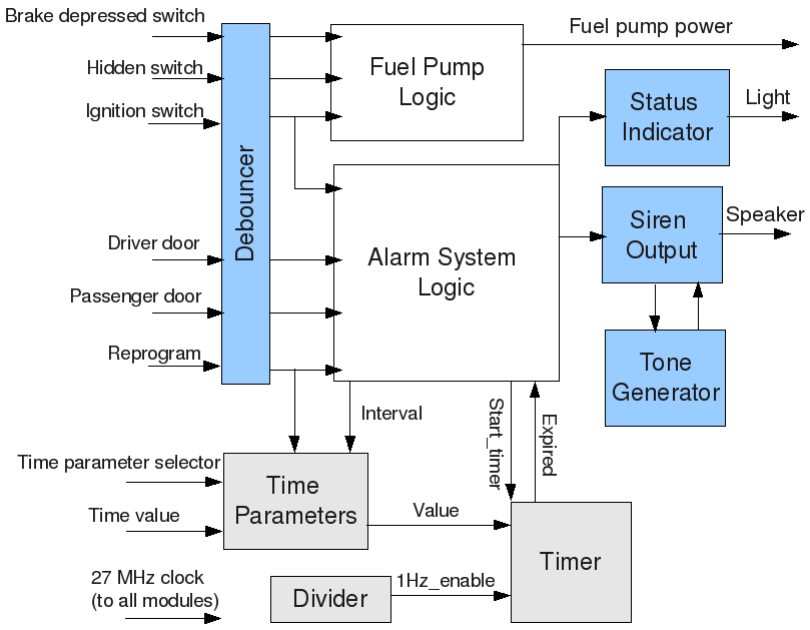

Figure 2: Module organization of device
The anti-theft system can be categorized inro three major categories: anti-theft logic (white), Timing (gray), and Input/Output processing (blue). The device is timed by a master 27 Mhz clock going to all of the modules.

avoided by modifying the design to set Start_timer as a function of both the state and the relevant inputs. Interval needed this modification as well in order to tell the timer what number to start counting from. Therefore, being in the "Sound_alarm" state would only mean that the siren and status light were on, and when all vehicle doors were closed, the timer would start once and only once.

The system powers up to the default state of "Armed," as indicated in Figure 3, in which the status light is blinking and the alarm is silent. Resetting the system or reprogramming the time parameters will bring the system back to this state, as well as set Start_timer = OFF and Interval = T_arm_delay. There is no real significance in setting Interval to T_arm_delay; it is just a arbitrary placeholder value. The vehicle will typically rest in the "Armed" state for the majority of its lifetime as it sits undisturbed in the garage. From here, it is possible to transition to the "Disarmed" state by turning on the ignition, or to the "Triggered" state by opening either DD or PD. Opening PD appropriately sets Interval = T_passenger_delay and starts the timer, while DD sets Interval = T_driver_delay and also starts the timer. Note that only one of the three transition scenarios can occur in a FSM model. Also, unless specified otherwise, Start_timer and Interval are their default values (FALSE and T_arm_delay respectively) for all transitions.

The "Triggered" state boasts a status light that is constantly on and a siren that is off, which is achieved by setting Status = ON and Siren= OFF. This state triggers a countdown of the Interval specified by transitioning from "Armed." If the countdown reaches 0 (i.e. Expired = True) before the driver gets a chance to turn on the ignition, the alarm sounds. If the ignition is first switched on, the alarm becomes disarmed.

As its name implies, the "Sound_alarm" state keeps the siren on and shows a constant light on the dashboard. The only way to turn off the siren is to first close both PD and DD which starts a timer with Interval= T_alarm_on. The next state is "Doors_closed_Alarm_on," which is the intermediate step before rearming the device. Although the status light turns off, Siren remains ON until T_alarm_on

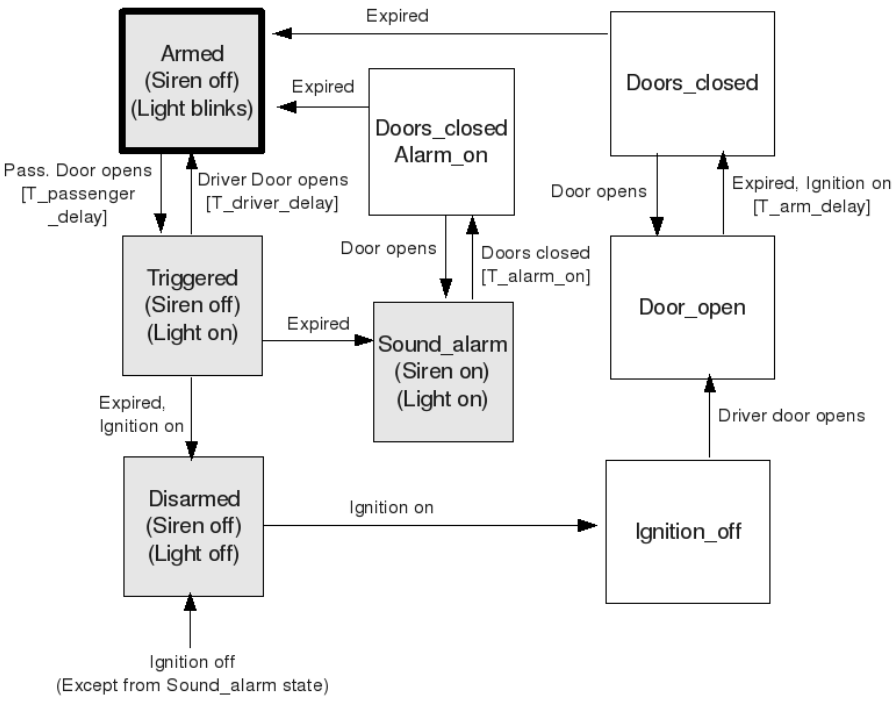

Figure 3: Alarm system FSM
This diagram shows the four major modes of operation (gray) and the transitional states between them (white). The default state is "Armed." At any given state, inputs not specifically included in the diagram will keep the system at that state.

expires or until the ignition is turned on, whichever comes first. An expired timer brings the system back to "Armed," whereas the ignition intervention sets the state to "Disarmed." If a door happens to open before either of these two events occur, the status light turns on and the timer resets as the system moves back to "Sound_alarm."

The vehicle will be in the "Disarmed" state whenever the ignition switch is on, except when doors are open in "Sound_alarm" mode. When disarmed, the status indicator and siren are naturally off. Resetting to the "Armed" state involves switching off the ignition, opening DD, then ensuring that both PD and DD are closed. Turning off the ignition transitions to the intermediate "Ignition_off" state, which transitions to another intermdiate "Door_open" state upon opening DD. Closing DD and PD starts the timer to countdown from Interval= T_arm_delay. The system is back at "Armed" upon expiration, or if either DD or PD reopens, the system is back at "Door_open." At any point in the rearming process, turning on the ignition returns the device back to "Disarmed."

Fuel pump security The fuel pump lockout feature was also implemented as a simple FSM. The inputs to this module are the brake depressed switch, hidden switch, and ignition switch, and the output is a binary signal to the fuel pump power control. The power to the fuel pump is cut off in the default state, or when the ignition switch is off. Although the audible alarm system is typically disarmed upon turning on the ignition switch, the fuel pump power remains off until both the brake depressed and hidden switches are pressed on simultaneously. They may be released and the power will remain on until the ignition is turned off.

Timing

Time parameters selector The time parameters module serves as a memory device for the different time values used in the anti-theft FSM. Each 4-bit value corresponds to a 2-bit interval lookup parameter. Table 1 shows the default values in decimal and binary representation for each of the four time intervals, as well as the corresponding lookup ID. Resetting the system will write the defaults back into the memory. The automated input is the Interval signal from the anti-theft FSM, which selects a value to output to the timer. The potential user inputs are a 2-bit time parameter selector, the desired 4-bit time value for that parameter, and the Reprogram button. The value is specified in seconds using binary representation, so because it only uses 4 bits, the user-defined time must fall in the range of 0-15 seconds. In the event that one or more parameters are set to 0, the anti-theft system will still function correctly. After manipulating the parameter and value inputs, pressing Reprogram will write the appropriate values into memory and also change the anti-theft FSM state to "Armed." Therefore, manipulating the parameter selector and value selectors will have no impact on the device until Reprogram= TRUE.

Table 1: Time parameters

Interval	Default (sec)	Parameter	Value
T_arm_delay	6	00	0110
T_driver_delay	8	01	1000
T_passenger_delay	15	10	1111
T_alarm_on	10	11	1010

System timing uses four parameters: the delay to arm the system after exiting the car (T_arm_delay), the delay until the siren turns on after opening the driver door (T_driver_delay) or the passenger door (T_passenger_delay), and the length of time during which the alarm sounds (T_alarm_on). [2]

Timer The timer module takes a specified number of seconds and counts down to zero. It inputs the time value from the time parameters module, the Start_timer input from the anti-theft system FSM, and a 1Hz_enable signal from the divider module. Upon power on or reset, the counter is initialized to 0. It waits for Start_timer= True, at which point it sets the counter to the time parameters output. At all other times, the counter is decremented once every second as paced by 1Hz_enable. It enables Expired for the FSM when the counter reaches 0.

Divider The divider module was written to convert the 27MHz master clock into a signal that is enabled once per second. The inputs are the clock signal and Start_timer, and the output is a 1Hz_enable used by the timer, siren output, and status indicator modules. Because the module needs to assert 1Hz_enable once every 27,000,000 clock cycles, a 25-bit counter is created to count up to that number. The counter is initialized to 0 and increments once per clock cycle. When it reaches 27,000,000, 1Hz_enable is asserted during that clock cycle and the counter resets itself.

The divider must reset when Start_timer is enabled to ensure that 1Hz_enable arrives at the timer module exactly one second after Start_timer. Otherwise, 1Hz_enable could come less than a second afterwards because Start_timer could potentially turn on while the divider counter is already in the process of incrementing.

Input/Output Processing

Tone generator The purpose of the sound module is to generate a tone at a given frequency for the siren output module to pass to a loudspeaker. A tone is essentially a square wave that remains

Table 2: Siren frequencies

Musical Note	Frequency (Hz)	Tone generator input
C4 (middle C)	261	25,812
E4	329	34,439
G4	392	41,033
C5 (tenor C)	523	51,724

The frequencies correspond to musical notes that form a C-major arpeggio. The tone generator input refers to the number of 27 MHz clock cycles needed to equal half the period.

high for half the period and goes low for the other half. The module inputs the length of half the desired period and behaves similarly to a divider in that a counter keeps track of how many cycles has passed. When the counter increments up to half the period, the sound output is toggled and the counter reset.

Siren output This module is the actual siren generator which is turned on and off according to the Siren output from the anti-theft FSM. The alarm is unique from standard car alarms in order to avoid unnecessary confusion. The sequence of tones was programmed as a progression of musical notes, more specifically a C-major arpeggio. First, the tone generator creates sounds with the correct standard frequencies, as listed in Table 2. The tone generator inputs were calculated with the following equation: Tone generator input $= \frac{27*10^6}{2*f}$, where f is the frequency of the note in Hz.

The arpeggio was modeled as a simple 6-state FSM which begins at C4 (middle C), ascends up E4, G4, and C5 (tenor C), then descends to G4, E4, and back to the initial state. A different note is played every second by changing state only when 1Hz_enable is asserted.

Status indicator This module controls the status light on the dashboard by inputting the Status signal of the main anti-theft module and selecting between three different modes: on, off, or blinking. When Status= on, the indicator module turns on an LED light, and vice versa. For the blinking mode, the module utilizes the 1Hz_enable signal from the divider and toggles the LED on and off with a 2-second period. This is also the default indicator mode because the default anti-theft system state is "Armed."

Sensor inputs Because of inherent mechanical imperfections in switch design, pressing a button or toggling a switch may unintentionally send rapid on/off signals to the synchronized system before stabilizing to a steady state. The debounce module developed by Terman and colleagues filters out the perturbations by requiring an input to be stable for 0.01 seconds before reporting the transition.[2] The module was used as is to filter the brake depressed switch, hidden switch, Ignition, DD, PD, and Reprogram before passing the signals through to the various system modules. The time parameter selector and time value selector were not debounced because a user would typically have set them for at least 1 second before pressing Reprogram.

3 Testing and Debugging

Several testing and debugging steps were performed to ensure the correct functionality of the device prototype. The car sensors were simulated using electrical pushbuttons and switches, an LED represented the status indicator, another LED replaced the fuel pump power, and a small external speaker delivered sound. For debugging purposes, an external LCD display was set to display 8 numerical digits that provided real-time information on the following variables: the timer's countdown clock, the Status indicator mode, the anti-theft FSM's state, the fuel pump's state, the Siren on/off signal, the

Start_timer signal, the Expired signal, and the selected time Interval. A user-reset button was also added to facilitate resetting to default values.

Timing Some problems were discovered with the timer module early on in the debugging process, thanks to the countdown, Start_timer, and Expired signals displayed on the LCD. At first, upon PD or DD opening, the countdown would initialize to the correct time but would not decrement until after the door had closed. This was not consistent with the specifications, which require the countdown to start upon input trigger. Because the Start_timer display constantly showed 0, it was discovered that the old timer assumed Start_timer stayed on for the duration of the countdown. The bug was resolved with a minor software modification.

Once the timer was shown to be working, the time parameters were modified to test the memory rewrite. At an arbitrary state in the anti-theft FSM, T_passenger_delay was set to a custom value. After pressing Reprogram, the FSM was brought to the "Armed" mode. Next, the passenger door was opened and the newly programmed counter value was observed on the external display. This general procedure was repeated for all 4 parameters. Finally, all of the parameters were set to 0 and the device behavior was shown to be unchanged, albeit the timers expired instantaneously.

Input/Output Processing The status indicator module was tested by comparing the digit on the Status indicator display to the expected behavior of the connected LED. The debouncer module was not explicitly tested because it had been tested by its developers. The tone generator and siren output modules were debugged after first manuevering through various system states and ensuring that the Siren digit matched the expected values. The speaker was then connected to the Siren output module. To test the tone generator, the siren was set to a single tone. The first test produced a very short siren blip before becoming silent, which, upon inspection of the code, was due to the sound was accidentally being set to 0 when incrementing the counter. The next test produced a lower pitch than expected because the input to the tone generator was a full period rather than a half. After fixing the issue, multiple tones were added in and auditory perception validated the final siren output.

Anti-theft Logic The anti-theft systems were verified using the countdown clock and state information from the LCD display. Beginning in the default "Armed" mode, the alarm FSM was taken through the various input scenarios described in the previous section. For example, when a door was opened from the "Armed" state, the counter value was confirmed on the display. When the counter showed 0, the alarm would sound. If the ignition switch was turned on instead, the status LED turned off and the system was taken through the rearming process. The fuel pump logic was tested in a similar way so that the fuel pump LED turned on after entering the appropriate switch combination.

4 Conclusion

The anti-theft device described in this report successfully passed the testing and debugging stage and was proven to meet the functional specifications. The design has customized control features and outputs as well as an extra layer of protection. The device utilizes sensors located in strategic locations on the vehicle to automate the system. Sensors on the driver and passenger doors, the ignition switch, a brake pedal switch, and a secret switch below the seat regulate the siren alarm, the dashboard status light, and the fuel pump power. The siren serves the purpose of drawing attention to potential thieves with its distinct cycle of 4 loud tones. The status light provides clear, useful information to the driver about the current mode of the alarm system. In addition, the 4 time intervals used by the system may be reprogrammed externally, allowing for a greater degree of flexibility and user control.

Many additional features may be incorporated in future designs. For example, the current design does not protect against towing. The installation of a tilt or motion sensor can enable alarm activation when the car experiences unusual movement. Other considerations may involve playing a recorded message before blaring the alarm or paging the owner when the alarm is activated. However, all of

these features invoke significant hardware costs which were not deemed necessary for an effective alarm system.

References

[1] United States Department of Justice, Federal Bureau of Investigation. (September 2008). Crime in the United States, 2007 <http://www.fbi.gov/ucr/07cius.htm>.

[2] Terman, Chris and colleagues. 6.111 Lab #3, 2008 <http://web.mit.edu/6.111/www/f2008/index.html>.

Appendix

```
////////////////////////////////////////////////////////////////////////
// Switch Debounce Module
////////////////////////////////////////////////////////////////////////
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce #(parameter DELAY=270000) // .01 sec with a 27Mhz clock
(input reset, clock, noisy,
output reg clean);
reg [18:0] count;
reg new;
always @(posedge clock)
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
```

```
//////////////////////////////////////////////////////////////////////////////////
// 6.111 FPGA Labkit – Hex display driver
// File: display_16hex.v
// Date: 24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes). These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
//////////////////////////////////////////////////////////////////////////////////
module display_16hex (reset, clock_27mhz, data,
disp_blank, disp_clock, disp_rs, disp_ce_b,
disp_reset_b, disp_data_out);
input reset, clock_27mhz; // clock and reset (active high reset)
input [63:0] data; // 16 hex nibbles to display
output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
disp_reset_b;
reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;
//////////////////////////////////////////////////////////////////////////////////
//
// Display Clock
//
// Generate a 500kHz clock for driving the displays.
//
//////////////////////////////////////////////////////////////////////////////////
reg [4:0] count;
reg [7:0] reset_count;
reg clock;
wire dreset;
always @(posedge clock_27mhz)
begin
if (reset)
begin
count = 0;
clock = 0;
end
else if (count == 26)
begin
clock = ~clock;
count = 5'h00;
end
else
count = count+1;
end
always @(posedge clock_27mhz)
if (reset)
```

```
reset_count <= 100;
else
reset_count <= (reset_count==0) ? 0 : reset_count-1;
assign dreset = (reset_count != 0);
assign disp_clock = ~clock;
//////////////////////////////////////////////////////////////////////////////
//
// Display State Machine
//
//////////////////////////////////////////////////////////////////////////////
reg [7:0] state; // FSM state
reg [9:0] dot_index; // index to current dot being clocked out
reg [31:0] control; // control register
reg [3:0] char_index; // index of current character
reg [39:0] dots; // dots for a single digit
reg [3:0] nibble; // hex nibble of current character
assign disp_blank = 1'b0; // low <= not blanked
always @(posedge clock)
if (dreset)
begin
state <= 0;
dot_index <= 0;
control <= 32'h7F7F7F7F;
end
else
casex (state)
8'h00:
begin
// Reset displays
disp_data_out <= 1'b0;
disp_rs <= 1'b0; // dot register
disp_ce_b <= 1'b1;
disp_reset_b <= 1'b0;
dot_index <= 0;
state <= state+1;
end
8'h01:
begin
// End reset
disp_reset_b <= 1'b1;
state <= state+1;
end
8'h02:
begin
// Initialize dot register (set all dots to zero)
disp_ce_b <= 1'b0;
disp_data_out <= 1'b0; // dot_index[0];
if (dot_index == 639)
state <= state+1;
else
dot_index <= dot_index+1;
end
```

```
8'h03:
begin
// Latch dot data
disp_ce_b <= 1'b1;
dot_index <= 31; // re-purpose to init ctrl reg
disp_rs <= 1'b1; // Select the control register
state <= state+1;
end
8'h04:
begin
// Setup the control register
disp_ce_b <= 1'b0;
disp_data_out <= control[31];
control <= {control[30:0], 1'b0}; // shift left
if (dot_index == 0)
state <= state+1;
else
dot_index <= dot_index-1;
end
8'h05:
begin
// Latch the control register data / dot data
disp_ce_b <= 1'b1;
dot_index <= 39; // init for single char
char_index <= 15; // start with MS char
state <= state+1;
disp_rs <= 1'b0; // Select the dot register
end
8'h06:
begin
// Load the user's dot data into the dot reg, char by char
disp_ce_b <= 1'b0;
disp_data_out <= dots[dot_index]; // dot data from msb
if (dot_index == 0)
if (char_index == 0)
state <= 5; // all done, latch data
else
begin
char_index <= char_index - 1; // goto next char
dot_index <= 39;
end
else
dot_index <= dot_index-1; // else loop thru all dots
end
endcase
always @ (data or char_index)
case (char_index)
4'h0: nibble <= data[3:0];
4'h1: nibble <= data[7:4];
4'h2: nibble <= data[11:8];
4'h3: nibble <= data[15:12];
4'h4: nibble <= data[19:16];
```

```
4'h5: nibble <= data[23:20];
4'h6: nibble <= data[27:24];
4'h7: nibble <= data[31:28];
4'h8: nibble <= data[35:32];
4'h9: nibble <= data[39:36];
4'hA: nibble <= data[43:40];
4'hB: nibble <= data[47:44];
4'hC: nibble <= data[51:48];
4'hD: nibble <= data[55:52];
4'hE: nibble <= data[59:56];
4'hF: nibble <= data[63:60];
endcase
always @(nibble)
case (nibble)
4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
endcase
endmodule
```

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Divider
//////////////////////////////////////////////////////////////////////////////////
module divider(clk, reset, start_timer, oneHZ_enable);
input clk, reset, start_timer;
output oneHZ_enable;
//1 MHz pulse
parameter delay = 'd27_000_000;
reg [24:0] counter; //25 bits to represent #s up to 27,000,000
always @ (posedge clk)
begin
if (start_timer||reset) //either master reset or start_timer, must synchronize
begin
counter <= 0;
end
else if (counter == delay) //cycle counter when maxed
begin
counter <= 0;
end
else
begin
counter <= counter+1;
end
end
assign oneHZ_enable = (counter == delay); //enable when counter is maxed
endmodule
```

```
‘timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//FSM
////////////////////////////////////////////////////////////////////////////////
module fsm(clk, reset, expired, ignition, driverdoor, passdoor, reprogram, interval, start_timer,
siren, led, state);
input clk;
input reset;
input expired;
input ignition;
input driverdoor;
input passdoor;
input reprogram;
output reg [1:0] interval;
output reg start_timer;
output siren;
output [1:0] led; //0= off, 1= on, 2= blink
output reg [2:0] state;
reg [2:0] next_state;
//List of states
parameter S_armed = 0;
parameter S_triggered = 1;
parameter S_soundalarm = 2;
parameter S_doorsclosed_alarm = 3;
parameter S_doorsclosed = 4;
parameter S_dooropen = 5;
parameter S_ignitionoff = 6;
parameter S_disarmed = 7;
always @ * begin
if (reset || reprogram) //goes to armed state if time values reprogrammed, or if master reset
begin
next_state = S_armed;
interval = 0;
start_timer = 0;
end
else
case (state)
////////////////////////////////
S_armed: begin
if (ignition) begin
next_state = S_disarmed;
start_timer = 0;
end
else if (passdoor) begin
next_state = S_triggered;
interval = 2'b10; //t_passenger_delay
start_timer=1;
end
else if (driverdoor) begin
next_state = S_triggered;
interval = 2'b01; //t_driver_delay
start_timer=1;
```

```
end
else begin
next_state = state;
start_timer=0;
end
end
//////////////////////////////
S_triggered: begin
if (ignition) begin
next_state = S_disarmed;
start_timer =0;
end
else if (expired) begin
next_state = S_soundalarm;
start_timer=0;
end
else begin
start_timer = 0;
next_state = state;
end
end
//////////////////////////////
S_soundalarm: begin
if (~driverdoor && ~passdoor) begin //all doors closed
next_state = S_doorsclosed_alarm;
interval = 2'b11; //t_alarm_on delay
start_timer = 1;
end
else begin
start_timer = 0;
next_state = state;
end
end
//////////////////////////////
S_doorsclosed_alarm: begin
if (ignition) begin
next_state = S_disarmed;
start_timer = 0;
end
else if (driverdoor || passdoor) begin //if any door is opened
next_state = S_soundalarm;
start_timer = 0;
end
else if (expired) begin
next_state = S_armed;
start_timer = 0;
end
else begin
start_timer = 0;
next_state = state;
end
end
```

```
//////////////////////////
S_doorsclosed: begin
if (ignition) begin
next_state = S_disarmed;
start_timer = 0;
end
else if (driverdoor || passdoor) begin //if any door open
next_state = S_dooropen;
start_timer = 0;
end
else if (expired) begin
next_state = S_armed;
start_timer = 0;
end
else begin
start_timer = 0;
next_state = state;
end
end
//////////////////////////
S_dooropen: begin
if (ignition) begin
next_state = S_disarmed;
start_timer = 0;
end
else if (~driverdoor && ~passdoor) begin //all doors closed
next_state = S_doorsclosed;
interval = 2'b00; //t_arm_delay
start_timer = 1;
end
else begin
start_timer = 0;
next_state = state;
end
end
//////////////////////////
S_ignitionoff: begin
if (ignition) begin
next_state = S_disarmed;
start_timer = 0;
end
else if (driverdoor) begin
next_state = S_dooropen;
start_timer= 0;
end
else begin
start_timer = 0;
next_state = state;
end
end
//////////////////////////
S_disarmed: begin
```

```
if (~ignition) begin
next_state = S_ignitionoff;
start_timer = 0;
end
else begin
start_timer = 0;
next_state = state;
end
end
//////////////////////////
default: next_state = S_armed;
endcase
end
always @ (posedge clk) state <= next_state;
//assign outputs which are dependent only on state
assign siren = (state == S_doorsclosed_alarm)||(state == S_soundalarm);
assign led = (state == S_armed) ? 2:
(state == S_triggered) ? 1:
(state == S_soundalarm) ? 1: 0;
endmodule
```

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//Fuel Pump
//////////////////////////////////////////////////////////////////////////////////
module fuelpump(clk, reset, brake, hidden, ignition, fuel, state);
input clk, reset, brake, hidden, ignition;
output fuel;
output reg [1:0] state;
//state assignments:
parameter S_pOFFcOFF = 2’b00; //fuel power is off
parameter S_pOFFcON = 2’b01; //fuel power is off, ignition on
parameter S_pONcON = 2’b11; //fuel power is on
reg [1:0] next_state;
always @ *
begin
if (reset)
next_state = S_pOFFcOFF;
else
case (state)
2’b00: if (ignition) next_state = S_pOFFcON;
else next_state = state;
2’b01: if (hidden & brake) next_state = S_pONcON;
else if (~ignition) next_state = S_pOFFcOFF;
else next_state = state;
2’b11: if (~ignition) next_state = S_pOFFcOFF;
else next_state = state;
default: next_state = S_pOFFcOFF;
endcase
end
always @ (posedge clk) state <= next_state;
assign fuel = (state == S_pONcON);
endmodule
```

```
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit – Template Toplevel Module
//
// For Labkit Revision 004
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes, 6.111 staff
//
////////////////////////////////////////////////////////////////////////////////
module labkit(
// Remove comment from any signals you use in your design!
// AC97
/*
output wire beep, audio_reset_b, ac97_synch, ac97_sdata_out,
input wire ac97_bit_clock, ac97_sdata_in,
*/
// VGA
/*
output wire [7:0] vga_out_red, vga_out_green, vga_out_blue,
output wire vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync, vga_out_vsync,
*/
// NTSC OUT
/*
output wire [9:0] tv_out_ycrcb,
output wire tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
output wire tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
output wire tv_out_subcar_reset;
*/
// NTSC IN
/*
input wire [19:0] tv_in_ycrcb,
input wire tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
output wire tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso, tv_in_reset_b,
tv_in_clock,
inout wire tv_in_i2c_data,
*/
// ZBT RAMS
/*
inout wire [35:0] ram0_data,
output wire [18:0] ram0_address,
output wire ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b,
output wire [3:0] ram0_bwe_b,
inout wire [35:0]ram1_data,
output wire [18:0]ram1_address,
output wire ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b,
output wire [3:0] ram1_bwe_b,
input wire clock_feedback_in,
output wire clock_feedback_out,
*/
// FLASH
/*
```

```
inout wire [15:0] flash_data,
output wire [23:0] flash_address,
output wire flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b,
input wire flash_sts,
*/
// RS232
/*
output wire rs232_txd, rs232_rts,
input wire rs232_rxd, rs232_cts,
*/
// PS2
//input wire mouse_clock, mouse_data,
//input wire keyboard_clock, keyboard_data,
// FLUORESCENT DISPLAY
output wire disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b,
//input wire disp_data_in,
output wire disp_data_out,
// SYSTEM ACE
/*
inout wire [15:0] systemace_data,
output wire [6:0] systemace_address,
output wire systemace_ce_b, systemace_we_b, systemace_oe_b,
input wire systemace_irq, systemace_mpbrdy,
*/
// BUTTONS, SWITCHES, LEDS
input wire button0,
input wire button1,
input wire button2,
input wire button3,
input wire button_enter,
//input wire button_right,
//input wire button_left,
input wire button_down,
//input wire button_up,
input wire [7:0] switch,
//output wire [7:0] led,
output wire [1:0] led,
// USER CONNECTORS, DAUGHTER CARD, LOGIC ANALYZER
inout wire [31:0] user1,
//inout wire [31:0] user2,
//inout wire [31:0] user3,
//inout wire [31:0] user4,
//inout wire [43:0] daughtercard,
//output wire [15:0] analyzer1_data, output wire analyzer1_clock,
//output wire [15:0] analyzer2_data, output wire analyzer2_clock,
//output wire [15:0] analyzer3_data, output wire analyzer3_clock,
//output wire [15:0] analyzer4_data, output wire analyzer4_clock,
// CLOCKS
//input wire clock1,
//input wire clock2,
input wire clock_27mhz
);
```

```
    ////////////////////////////////////////////////////////////////////////////
    //
    // Reset Generation
    //
    // A shift register primitive is used to generate an active-high reset
    // signal that remains high for 16 clock cycles after configuration finishes
    // and the FPGA's internal clocks begin toggling.
    //
    ////////////////////////////////////////////////////////////////////////////
    wire power_on_reset, user_reset, reset;
    SRL16 reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),
    .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
    defparam reset_sr.INIT = 16'hFFFF;
    debounce db(.reset(power_on_reset), .clock(clock_27mhz),
    .noisy(~button_down), .clean(user_reset));
    assign reset = power_on_reset | user_reset; //system can be reset either upon poweron or by
pushing button_down
    ////////////////////////
    /// Anti-Theft system ///
    ////////////////////////
    //inputs and outputs
    wire hidden, brake, driverdoor, passdoor, reprogram;
    wire ignition;
    wire [1:0] time_parameter_selector;
    wire [3:0] time_value;
    wire [1:0] status;
    wire pumppower;
    wire sirenoutput;
    //submodule interconnections
    wire [1:0] fsm_interval;
    wire [3:0] timeparameters_value;
    wire fsm_start_timer;
    wire divider_oneHz_enable;
    wire timer_expired;
    wire [1:0] fsm_status;
    wire fsm_siren;
    //debugging tools
    wire [2:0] fsm_state;
    wire [3:0] showcount;
    wire [1:0] fuel_state;
    //debounce buttons and switches
    //assign inputs to labkit ports
    debounce debouncebutton0(.reset(reset), .clock(clock_27mhz), .noisy(~button0), .clean(hidden));
    debounce debouncebutton1(.reset(reset), .clock(clock_27mhz), .noisy(~button1), .clean(brake));
    debounce debouncebutton2(.reset(reset), .clock(clock_27mhz), .noisy(~button2), .clean(driverdoor));
    debounce debouncebutton3(.reset(reset), .clock(clock_27mhz), .noisy(~button3), .clean(passdoor));
    debounce debouncebutton_enter(.reset(reset), .clock(clock_27mhz), .noisy(~button_enter), .clean(reprogram));
    debounce debounceswitch7 (.reset(reset), .clock(clock_27mhz), .noisy(switch[7]), .clean(ignition));
    //assign inputs to labkit ports
    //do not need debouncing because their values are stable by the time reprogram is pressed
    assign time_parameter_selector = switch[5:4];
    assign time_value = switch[3:0];
```

```
//assign outputs to labkit ports
assign led[0] = ~status; //led on = 0, led off = 1
assign led[1] = ~pumppower;
assign user1[0] = sirenoutput;
//zero unused ports
assign user1[31:1] = 0;
//instantiate instances of submodules
divider divider0 (.clk(clock_27mhz), .reset(reset), .start_timer(fsm_start_timer),
.oneHZ_enable(divider_oneHz_enable));
fuelpump fuelpump0 (.clk(clock_27mhz), .reset(reset), .brake(brake), .hidden(hidden),
.ignition(ignition), .fuel(pumppower), .state(fuel_state));
timeparameters timeparameters0(.clk(clock_27mhz), .reset(reset), .time_value(time_value),
.time_parameter_selector(time_parameter_selector), .interval(fsm_interval),
.reprogram_button(reprogram), .value(timeparameters_value));
timer timer0 (.clk(clock_27mhz), .reset(reset), .start_timer(fsm_start_timer),
.oneHz_enable(divider_oneHz_enable), .initValue(timeparameters_value),
.expired(timer_expired), .counter(showcount));
fsm fsm0 (.clk(clock_27mhz), .reset(reset), .expired(timer_expired), .ignition(ignition),
.driverdoor(driverdoor), .passdoor(passdoor), .reprogram(reprogram),
.interval(fsm_interval), .start_timer(fsm_start_timer), .siren(fsm_siren),
.led(fsm_status), .state(fsm_state));
leddriver leddriver0 (.clk(clock_27mhz), .reset(reset), .oneHz_enable(divider_oneHz_enable),
.fsm_status(fsm_status), .ledout (status));
siren siren0 (.clk(clock_27mhz), .reset(reset), .oneHZ_enable(divider_oneHz_enable),
.turn_on(fsm_siren), .siren_out(sirenoutput));
////////////////////////////
/////////debugging//////////
////////////////////////////
wire [63:0] data;
display_16hex display_16hex0(reset, clock_27mhz, data,
disp_blank, disp_clock, disp_rs, disp_ce_b,
disp_reset_b, disp_data_out);
assign data [3:0] = showcount;
assign data [7:4] = fsm_siren;
assign data [11:8] = fsm_state;
assign data [15:12] = fsm_status;
assign data [19:16] = fuel_state;
assign data [23:20] = fsm_start_timer;
assign data [27:24] = fsm_interval;
assign data [31:28] = timer_expired;
endmodule
```

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Led Driver
//////////////////////////////////////////////////////////////////////////////////
module leddriver(clk, reset, oneHz_enable, fsm_status, ledout);
input clk, reset, oneHz_enable;
input [1:0] fsm_status;
output reg ledout;
always @ (posedge clk) begin
if (reset) begin
if (oneHz_enable)
ledout <= ~ledout; //blink in default armed state
end
else
case (fsm_status)
0: ledout <= 0; //led off
1: ledout <= 1; //led on
2: begin
if (oneHz_enable)
ledout <= ~ledout; //blink
end
default: ledout <=0;
endcase
end
endmodule
```

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Siren
// Additional Comments: Realized that the "freq" input needs to be halved so that the tone is high 50% of the period.
//////////////////////////////////////////////////////////////////////////////////
module siren(clk, reset, oneHZ_enable, turn_on, siren_out);
input clk, reset, oneHZ_enable, turn_on;
output siren_out;
/* standard functionality
//generate tones
wire [24:0] freq400, freq700;
wire sound400, sound700;
assign freq400 = 25'd33_750; // 27 Mhz/400 hz =67500 clock cycles -> 33750
assign freq700 = 25'd19_285; // 27 Mhz/700 hz = 38571 clock cycles -> 19285
sound sound0(clk, reset, freq400, sound400);
sound sound1 (clk, reset, freq700, sound700);
//Alternate tones
reg toggle;
always @ (posedge clk) begin
if (reset) toggle <= 0;
if (oneHZ_enable) toggle <= ~toggle;
end
assign siren_out = (~turn_on)? 0: ((toggle)? sound400: sound700);
*/
// play a C major arpeggio
wire [24:0] freqC1, freqE, freqG, freqC2;
wire soundC1, soundE, soundG, soundC2;
assign freqC1 = 25'd51_724; //261 hz
assign freqE = 25'd41_033; //329 hz
assign freqG= 25'd34_439; //392
assign freqC2= 25'd25_812; //523
//generate tones
sound sound0(clk, reset, freqC1, soundC1);
sound sound1(clk, reset, freqE, soundE);
sound sound2(clk, reset, freqG, soundG);
sound sound3(clk, reset, freqC2, soundC2);
//set up fsm
reg [2:0] state, next_state;
parameter S_C1 = 0;
parameter S_ Eup= 1;
parameter S_Gup = 2;
parameter S_C2 = 3;
parameter S_Edown = 4;
parameter S_Gdown = 5;
always @ * begin
if (reset) next_state=S_C1;
else
case (state)
S_C1: next_state = S_E;
S_E: next_state = S_G;
S_G: next_state = S_C2;
```

```
S_C2: next_state = S_Edown;
S_Edown: next_state = S_Gdown;
S_Gdown: next_state = S_C1;
endcase
end
always @ (posedge clk)
if (oneHZ_enable) state <= next_state; //change note every second
//logic to output correct tone
assign siren_out = (~ turn_on) ? 0:
(state == S_C1) ? sound0 :
((state == S_Eup) || (state == S_Edown)) ? sound1:
((state == S_Gup)||(state == S_Gdown)) ? sound2:
(state == S_C2) ? sound3 : 0;
endmodule
```

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Sound
//////////////////////////////////////////////////////////////////////////////////
module sound(clk, reset, freq, sound);
input clk, reset;
input[24:0] freq;
output reg sound;
reg [24:0] counter;
//generates tone with desired frequency
//freq = half the desired period
always @ (posedge clk) begin
if (reset) begin
counter <= 0;
sound <= 0;
end
else if (counter == freq) begin
counter <= 0;
sound <= ~sound; //square wave high for freq# of clk cycles, then goes low
end
else begin
counter <= counter+1;
end
end
endmodule
```

```
‘timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Time parameters
////////////////////////////////////////////////////////////////////////////////
module timeparameters(clk, reset, time_value, time_parameter_selector, interval, reprogram_button,
value);
input clk, reset, reprogram_button;
input [3:0] time_value;
input [1:0] time_parameter_selector, interval;
output wire [3:0] value;
//timetable for 4-bit time values
reg [3:0] t_arm_delay, t_driver_delay, t_passenger_delay, t_alarm_on;
//select parameter to be used by Timer
assign value = (interval == 2'b00) ? t_arm_delay :
(interval == 2'b01) ? t_driver_delay :
(interval == 2'b10) ? t_passenger_delay : t_alarm_on;
always @ (posedge clk)
begin
if (reset) //set timetable to default values
begin
t_arm_delay <= 4'd6;
t_driver_delay <= 4'd8;
t_passenger_delay <= 4'd15;
t_alarm_on <= 4'd10;
end
else if (reprogram_button) //write new values to timetable
begin
case (time_parameter_selector) //pick which entry to change
2'b00: t_arm_delay <= time_value; //change entry to specified value
2'b01: t_driver_delay <= time_value;
2'b10: t_passenger_delay <= time_value;
2'b11: t_alarm_on <= time_value;
endcase
end
//if !reset or !reprogram_button, then do nothing
end
endmodule
```

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Timer
//////////////////////////////////////////////////////////////////////////////////
module timer(clk, reset, start_timer, oneHz_enable, initValue, expired, counter);
input clk, reset, start_timer, oneHz_enable;
input [3:0] initValue;
output expired;
output reg [3:0] counter;
always @ (posedge clk) begin
if (reset) counter <= 0;
else if (start_timer) counter <= initValue; //sets counter to value specified by Interval
else if (oneHz_enable) begin //every second, decrement counter until 0
if (counter != 0) counter <= counter-1;
end
end
assign expired = (counter == 0);
endmodule
```