# **The Dorm Room Genie:**
Design of a Dorm Room Automation System

David Nedzel and Aaron Stonely
6.111: Introductory Digital Systems Laboratory
December 14, 2005

## Abstract

This paper describes the design and implementation of a dorm room automation system which we call the Dorm Room Genie. Through a system of keyboard inputs, push buttons, sensors, and a VGA display, the Dorm Room Genie provides the student with a customizable way to manage their dorm room with the ultimate goal of making the student's life easier. Students are issued keys by which to access their dorm rooms, but far too often, a key is forgotten inside, leaving the student locked out. The Dorm Room Genie addresses this issue by providing the student with the ability for keyless entry. The Dorm Room Genie also contains a burglar alarm system which will provide the student additional security in their dorm room. This additional security is provided through a loud buzzer and the ability to take photographs of the intruder. Additional functionality includes a digital clock and alarm clock. The Dorm Room Genie also provides the student with the ability to control the lights manually, automatically, or in conjunction with the alarm clock. Finally, student will be able to instantly see the status of the dorm room automation system via a system status display. The objective of this paper is to outline the planning, design, testing, and debugging involved in the implementation of a dorm room automation system on a field-programmable gate array.

**Table of Contents**

## List of Tables and Figures

## Dorm Room Genie Overview

In university residence halls across the country, you'll find lots of busy students – many of whom enjoy gadgets and conveniences which make there lives a little more simple and fun. The Dorm Room Genie enhances a student's dormitory room with several features and conveniences which the user can choose to make use of:

- **Electronic Lock**: After installing an electronic strike plate, the Dorm Room Genie allows the student to input a password which can be used to unlock the door. This is convenient in the event of an accidental lock out, or in the event that the student wishes to grant a friend access to his or her room. The lock can be set to locked or unlocked, and additionally password entry can be disabled if the student wishes for an enhanced level of security.
- **Burglar Alarm**: After installing a magnetic sensor on the door, and mounting a camera, the student can take advantage of the security alarm feature. The security alarm can be armed at the touch of a button, to be tripped by the door sensor and a camera-implemented motion detector. If the student wishes to activate the burglar alarm while at home for added security, motion detection can be disabled. When the alarm is triggered, a buzzer sounds, the time is recorded, and the camera begins capturing images of the intrusion which can be viewed later. The alarm can be deactivated by entering the electronic lock password.
- **Lighting Control**: The Dorm Room Genie also offers advanced lighting control features. A lamp plugged into the system can be brightened or dimmed at the touch of a button. Additionally, the lighting controller will flash the lights during an intrusion to further deter a burglar. The lighting module also interacts with the alarm clock as described below.
- **Alarm Clock**: The Dorm Room Genie also contains an alarm clock which sounds a buzzer at the desired time. Optionally, 15 minutes before the user-programmed wake-up time, the lights can be set to gradually brighten to gently wake the student before the buzzer sounds. A 15 minute snooze feature is also included.
- **Display**: An easy to read LCD display outputs the status of the system to the user, and also allows the user to view images captured during an intrusion.

The user interfaces with the Dorm Room Genie thorough a standard PS/2 keyboard which is used for password setting and entry, setting the clock and wake-up time in addition to sending control signals to the various modules of the system. The keyboard with the control buttons is on the inside of the room and a keyboard without the control buttons is on the outside door. In explaining the architecture of the Dorm Room Genie, the follow section will consider the functionality and implementation of each individual model separately, and also explain how the modules interact with one another.

## Description

### Signal Processing Module

--By David Nedzel

The Signal Processing Module allows the user to input commands into the system. It is responsible for interpreting the user's keyboard inputs, and sending the appropriate commands and signals off to the other modules of the Dorm Room Genie. As illustrated in Figure 1, the Signal Processing module has three primary components: A PS/2 to ASCII converter, a Finite State Machine (FSM), and a key to command mapper.

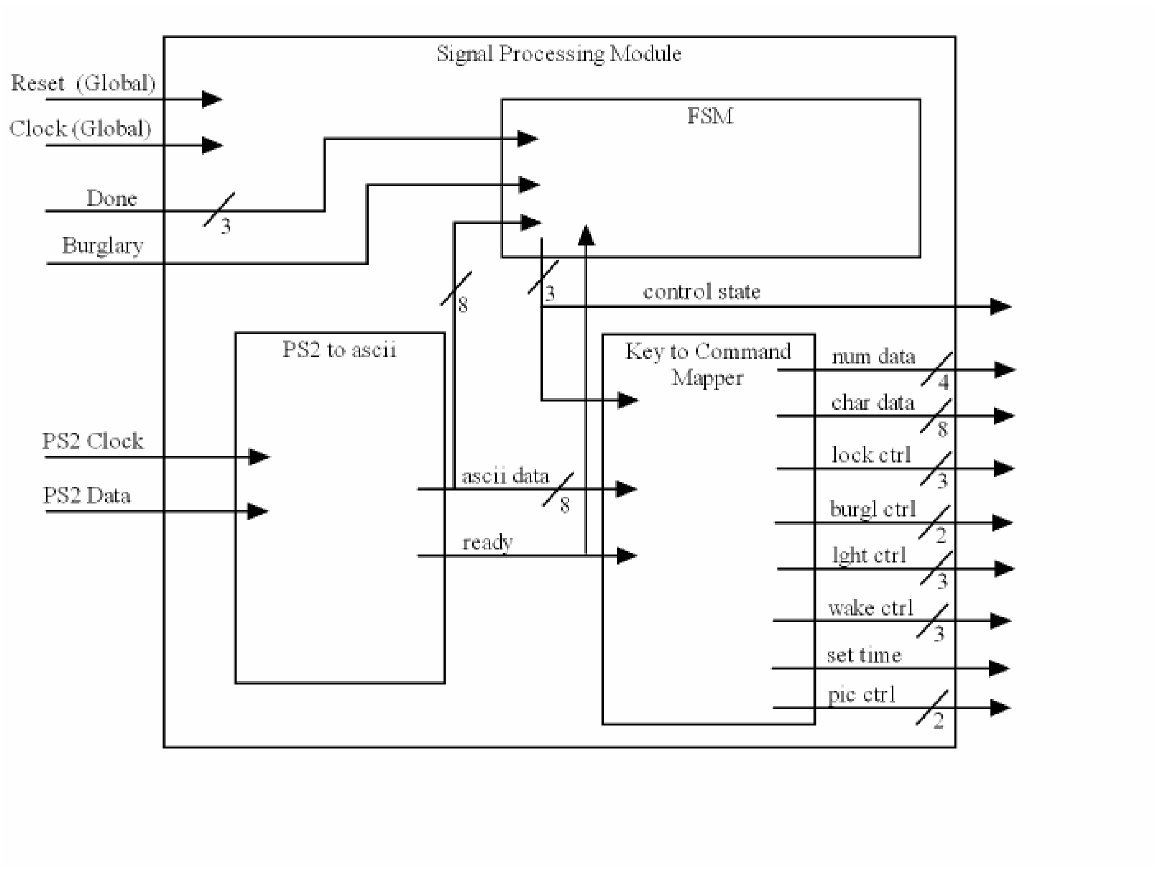

Figure 1

Both the PS/2 clock and data enter the signal processing module where they are fed into the PS2 to ASCII converter. The converter's job is to interpret the native keyboard data and output 8-bit ASCII code. The PS/2 module provided by the 6.111 staff correctly handled letters, assigning them the capital letter ASCII codes, numbers and punctuation marks as summarized in Table 1. Unfortunately, we also required the use of the keyboard's function keys and numeric keypad to send control signals to the lock, burglar alarm, clock, wake-up alarm, lighting and video modules. It was therefore necessary to expand the functionality of this module to include those additional keys listed in Table 2 along with their intended function. Note that the ASCII values assigned are not actually associated with those keys, but were used for convenience since they fall outside of the standard 128 basic ASCII codes. It is of little concern that these codes overlap with the 256 extended ASCII codes, since the extended ASCII symbols are unimportant to this implementation. Once the data is converted to ASCII codes, it is sent to both the FSM and command mapper unit.

| **Keyboard Signal** | **ASCII Code** | **Character** | | **Keyboard Signal** | **ASCII Code** | **Character** |
|---|---|---|---|---|---|---|
| 8'h1C: | 8'h41 | A | | 8'h45: | 8'h30 | 0 |
| 8'h32: | 8'h42 | B | | 8'h16: | 8'h31 | 1 |
| 8'h21: | 8'h43 | C | | 8'h1E: | 8'h32 | 2 |
| 8'h23: | 8'h44 | D | | 8'h26: | 8'h33 | 3 |
| 8'h24: | 8'h45 | E | | 8'h25: | 8'h34 | 4 |
| 8'h2B: | 8'h46 | F | | 8'h2E: | 8'h35 | 5 |
| 8'h34: | 8'h47 | G | | 8'h36: | 8'h36 | 6 |
| 8'h33: | 8'h48 | H | | 8'h3D: | 8'h37 | 7 |
| 8'h43: | 8'h49 | I | | 8'h3E: | 8'h38 | 8 |
| 8'h3B: | 8'h4A | J | | 8'h46: | 8'h39 | 9 |
| 8'h42: | 8'h4B | K | | | | |
| 8'h4B: | 8'h4C | L | | 8'h0E: | 8'h60 | ` |
| 8'h3A: | 8'h4D | M | | 8'h4E: | 8'h2D | - |
| 8'h31: | 8'h4E | N | | 8'h55: | 8'h3D | = |
| 8'h44: | 8'h4F | O | | 8'h5D: | 8'h5C | \ |
| 8'h4D: | 8'h50 | P | | 8'h29: | 8'h20 | (space) |
| 8'h15: | 8'h51 | Q | | 8'h54: | 8'h5B | [ |
| 8'h2D: | 8'h52 | R | | 8'h5B: | 8'h5D | ] |
| 8'h1B: | 8'h53 | S | | 8'h4C: | 8'h3B | ; |
| 8'h2C: | 8'h54 | T | | 8'h52: | 8'h27 | ' |
| 8'h3C: | 8'h55 | U | | 8'h41: | 8'h2C | , |
| 8'h2A: | 8'h56 | V | | 8'h49: | 8'h2E | . |
| 8'h1D: | 8'h57 | W | | 8'h4A: | 8'h2F | / |
| 8'h22: | 8'h58 | X | | 8'h5A: | 8'h0D | (enter) |
| 8'h35: | 8'h59 | Y | | | | |
| 8'h1A: | 8'h5A | Z | | | | |

Table 1: Listing of ASCII translations by PS/2 to ASCII Module

| Keyboard Signal | "ASCII" Code | Key | Function |
|---|---|---|---|
| 8'h05: | 8'h81 | F1 | Lock |
| 8'h06: | 8'h82 | F2 | Unlock |
| 8'h04: | 8'h83 | F3 | Inactive |
| 8'h0C: | 8'h84 | F4 | <not used> |
| 8'h03: | 8'h85 | F5 | Arm |
| 8'h0B: | 8'h86 | F6 | Disarm |
| 8'h83: | 8'h87 | F7 | Next Picture |
| 8'h0A: | 8'h88 | F8 | Previous Picture |
| 8'h01: | 8'h89 | F9 | Set Wake Time |
| 8'h09: | 8'h8A | F10 | Set Clock |
| 8'h78: | 8'h8B | F11 | Set Password |
| 8'h07: | 8'h8C | F12 | <not used> |
| | | | |
| 8'h69: | 8'h91 | numeric 1 | Alarm Clock On |
| 8'h72: | 8'h92 | numeric 2 | Alarm Clock Off |
| 8'h7A: | 8'h93 | numeric 3 | |
| 8'h6B: | 8'h94 | numeric 4 | Lights Off |
| 8'h73: | 8'h95 | numeric 5 | Lights Sleep |
| 8'h74: | 8'h96 | numeric 6 | Lights Dim |
| 8'h6C: | 8'h97 | numeric 7 | Lights On |
| 8'h75: | 8'h98 | numeric 8 | Lights Auto |
| 8'h7D: | 8'h99 | numeric 9 | Lights Bright |
| 8'h70: | 8'h9A | numeric 0 | Alarm Clock Snooze |
| 8'h71: | 8'h9B | numeric . | |

Table 2: Listing of extended ASCII assignments

The job of the command mapper is to convert the incoming ASCII data to an appropriate format for each of the modules, and route those data and commands to the modules. The signal processing modules sends three distinct types of signals: data signals, control commands and state information. There are two types of data signals: The first, character data sent to the lock, is simply the ASCII codes received by the command mapper. When such ASCII data corresponds to one of the characters listed in Table 1, that data is simply passed along to the lock module. The second type of data, numbed data sent to the clock and wake modules, involves slightly more processing. If the ASCII data is recognized as a number, that number is reduced to 4-bit hexadecimal format before being passed along to the clock and wake modules. This is actually quite simple to engineer, since the four low bits of the ASCII code for numbers are the hexadecimal representation of the numbers.

The command mapper must also pass along control signals from the various user inputs listed in Table 2. I will consider each control signal in turn, explaining exactly when and why it is sent based upon user input. For each module, a “zero” signal corresponds to “no signal being sent”. The lock module receives the signals summarized in Table 3. The lock signal is sent when the user pushes “lock,” but also if the user arms the burglar alarm. The unlock, inactive and change password signals are sent when the user pushes the corresponding “unlock,” “inactive” and change password buttons. Each signal is sent for exactly one clock cycle.

| **Lock Control Signals** | |
|---|---|
| Signal | Meaning |
| 3'h1 | Lock |
| 3'h2 | Unlock |
| 3'h3 | Inactive |
| 3'h4 | <not used> |
| 3'h5 | Change Password |
| 3'h6 | <not used> |
| 3'h7 | <not used> |
| 3'h0 | No signal |

Table 3: Lock Control Signals

| **Burglar Alarm Control Signals** | |
|---|---|
| Signal | Meaning |
| 2'h1 | Arm |
| 2'h2 | Disarm |
| 2'h3 | <not used> |
| 2'h0 | No Signal |

Table 4: Burglar Alarm Control Signals

The burglar alarm module receives the signals summarized in Table 4. The arm signal is sent when the user pushes “arm”, and the disarm signal is sent when the user pushes “disarm”, but also when the user either pushes the unlock button, or when the user enters set password mode. After all, you wouldn’t be able to disarm the system if a burglary occurred during a password set phase, since you wouldn’t be able to enter the code to disarm the system. For the remaining lighting, wake alarm, clock and picture playback module (part of the video module) the signals summarized in Tables 5, 6, 7 and 8 are sent exactly when the user pushes the corresponding button.

| **Lighting Control Signals** | |
|---|---|
| Signal | Meaning |
| 3'h1 | Off |
| 3'h2 | On |
| 3'h3 | Auto |
| 3'h4 | Bright |
| 3'h5 | Dim |
| 3'h6 | Sleep |
| 3'h7 | <not used> |
| 3'h0 | No Signal |

Table 5: Lighting Control Signals

| **Wake Alarm Control Signals** | |
|---|---|
| Signal | Meaning |
| 3'h1 | Wake Alarm On |
| 3'h2 | Wake Alarm Off |
| 3'h3 | Snooze |
| 3'h4 | Wake Alarm Set |
| 3'h5 | <not used> |
| 3'h6 | <not used> |
| 3'h7 | <not used> |
| 3'h0 | No Signal |

Table 6: Wake Control Signals

| **Clock Control Signals** | |
|---|---|
| Signal | Meaning |
| 1'b1 | Set Clock |
| 1'b0 | No Signal |

Table 7: Clock Control Signals

| **Picture Control Signals** | |
|---|---|
| Signal | Meaning |
| 2'b01 | Next Picture |
| 2'b10 | Previous Picture |
| 2'b11 | <not used> |
| 2'b00 | No Signal |

Table 8: Picture Control Signals

The final data output is state information. The signal processing module routes control commands, and part of that routing involves knowing when, and when not to send commands – specifically, that is the function of the FSM. As illustrated in Figure 2, the Signal Processing Module's FSM has five states: normal, lock mode, clock mode, wake mode and burglary. In each of these states, the command mapper is only allowed to send certain signals, and this state information is output to the display.

When in "normal mode," the command mapper routes all the signals as described above. However, if the FSM is in any other state, transmission of the control signals is blocked. It is important to successful operation of the Dorm Room Genie that while one is the process of programming either the password, system time, or wake time, that other control signals be withheld. When the user enters one such program mode summarized in Figure 2, the system will remain in that state until the corresponding module sends a "done" signal, releasing the semaphore.

A burglary is handled in a similar way. When the alarm module sends the burglary signal, the FSM locks the control commands such that the burglar cannot simply push "disarm". The FSM returns to normal mode following an alarm once the burglary module de-asserts the burglary signal. This process will be explained in the discussion of the burglary module.

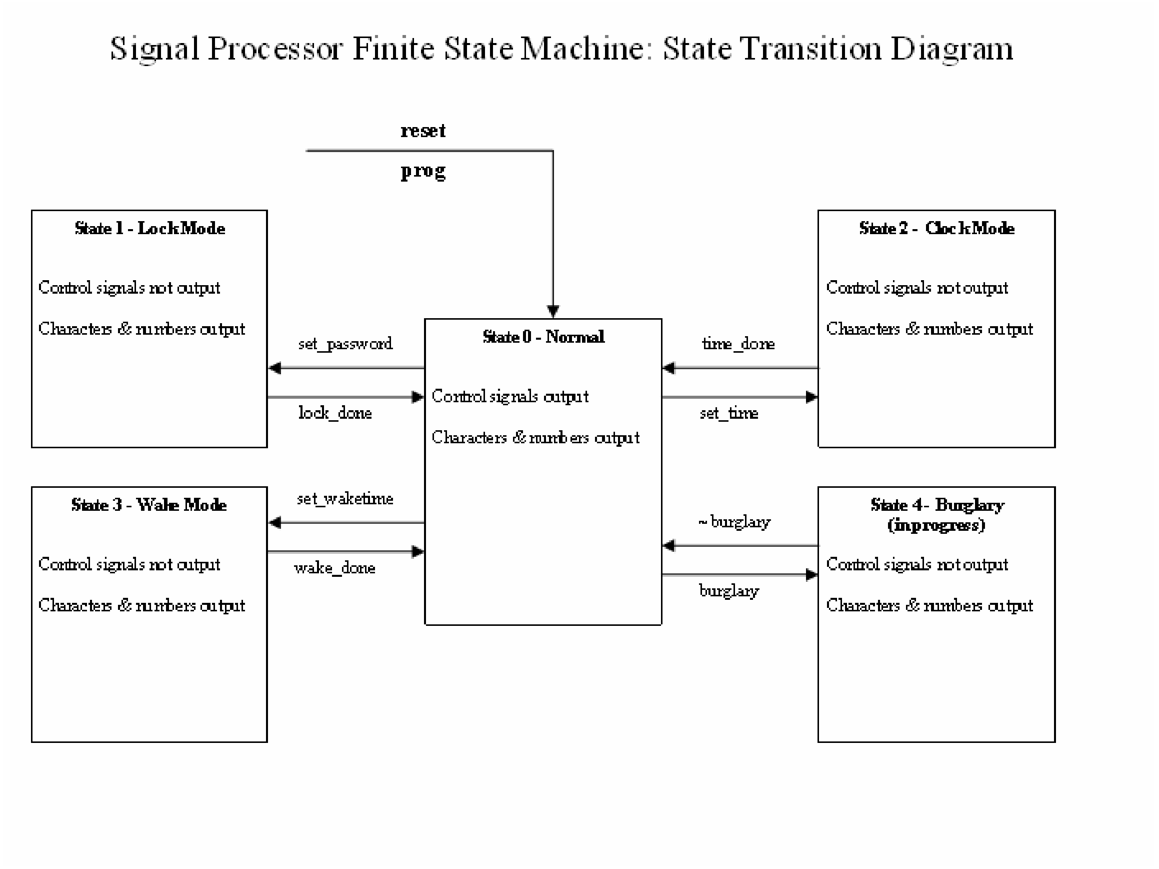
Signal Processor Finite State Machine: State Transition Diagram
reset
prog
State 1 - LockMode
Control signals not output
Characters & numbers output
State 2 - ClockMode
Control signals not output
Characters & numbers output
State 0 - Normal
Control signals output
Characters & numbers output
set_password
lock_done
time_done
set_time
State 3 - Wake Mode
Control signals not output
Characters & numbers output
set_waketime
wake_done
~ burglary
burglary
State 4 - Burglary (inprogress)
Control signals not output
Characters & numbers output

Figure 2: Signal Processor's FSM

## Lock Module

--By David Nedzel

The two primary purposes of the lock module are to control whether or not an electric latch is unlocked, and to manage the user's password – both the storage of the password, and user authentication with the password. Each of the lock's primary subunits works toward those goals: a block RAM stores the user's password, while an FSM controls the electronic latch and user authentication. As illustrated in Figure 3, the lock module takes character data and control signals from the signal processing module and input from a magnetic door switch; it outputs its current state to the display, sends a control signal to unlock the door, and sends out a "correct password entered" signal when the user completes entering the current password.

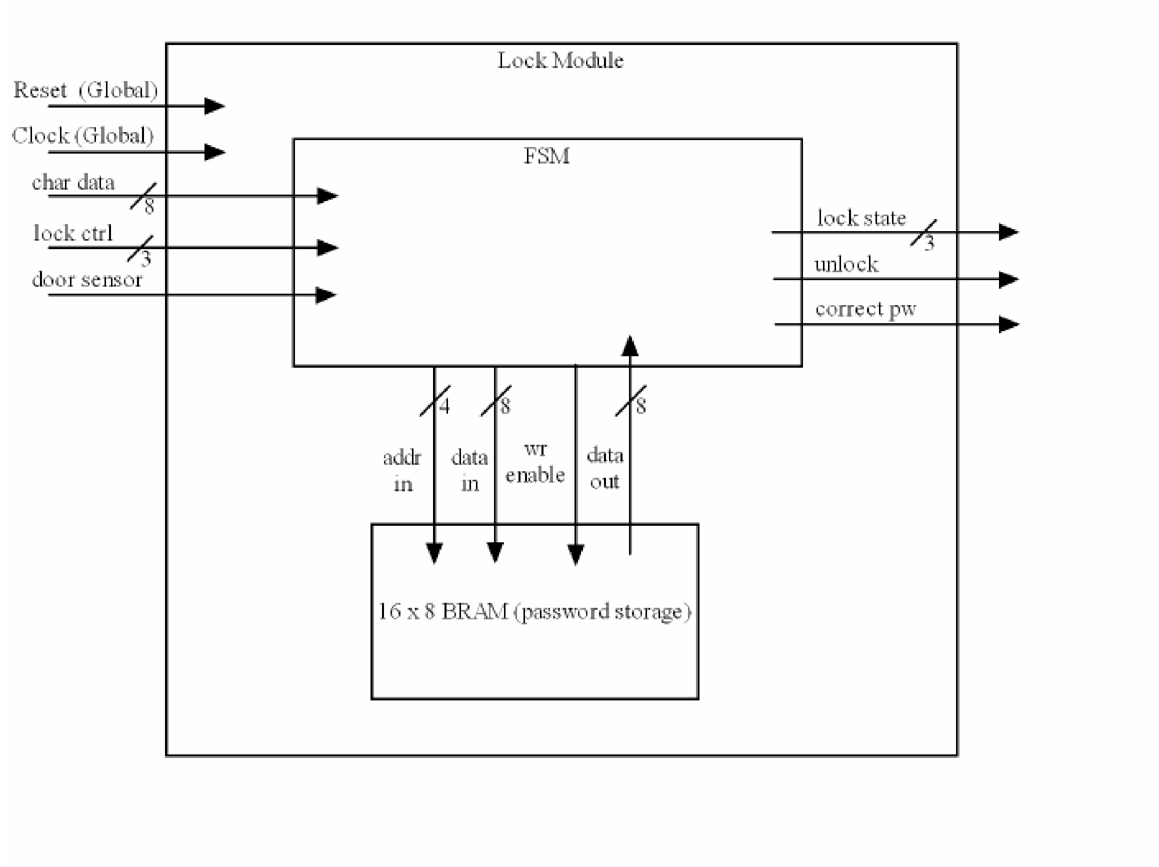

Figure 3: Lock Module Block Diagram

I created the 16 x 8 block RAM with the Xilinx Core Generation software. Each ASCII character in the password takes 8 bits to store, thus a user can have a password of up to sixteen characters. As I will explain, the FSM uses the data and control signals from the signal processing module and the door sensor to control both the reading and writing of the password storage RAM in addition to the unlock, correct password and lock state outputs. In using a magnetic door switch, with a digital system, I needed to design a way to hook up the magnetic switch to the lab kit. This turned out to be relatively simple. The 6.111 lab kit's user inputs accept voltage values of 3.3 V and 0 V as a logical "1" and "0". The magnetic sensor is simply a short circuit when the door is closed, and an open circuit when the door is open. Therefore, I hooked the lab kit's 3.3 V supply up to the user input across a 10 kΩ resistor, and hooked the magnetic switch from 0 V to the user input. This when the door is open, the lab kit sees 3.3 V producing a logical "1", and when the door is closed, the 3.3 V is dropped a cross the resistor, and the lab kit sees 0 V producing a logical 0. With the magnetic sensor digitized, I set out to design the FSM.

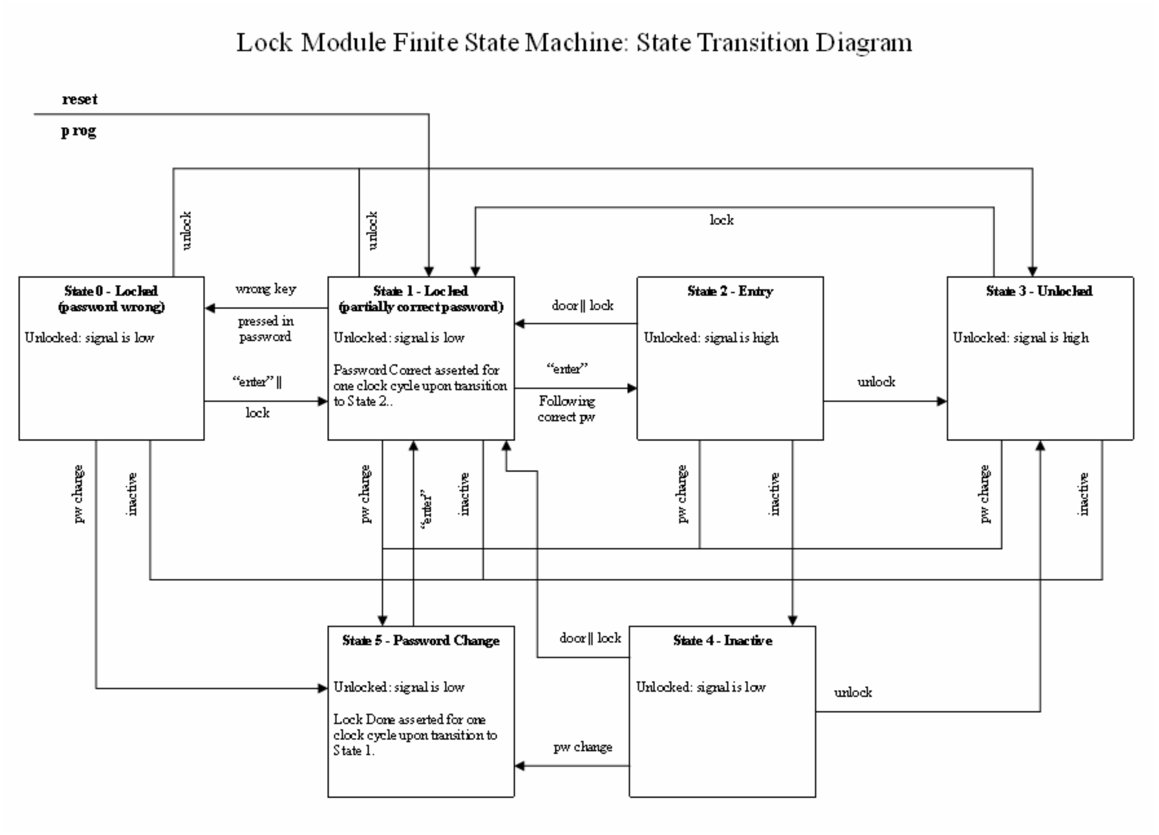
Lock Module Finite State Machine: State Transition Diagram
reset
prog
unlock
unlock
lock
State 0 - Locked (password wrong)
Unlocked: signal is low
wrong key pressed in password
"enter" || lock
State 1 - Locked (partially correct password)
Unlocked: signal is low
Password Correct asserted for one clock cycle upon transition to State 2..
door || lock
"enter" Following correct pw
State 2 - Entry
Unlocked: signal is high
unlock
State 3 - Unlocked
Unlocked: signal is high
pw change
inactive
pw change
"enter"
inactive
pw change
inactive
pw change
inactive
State 5 - Password Change
Unlocked: signal is low
Lock Done asserted for one clock cycle upon transition to State 1.
door || lock
State 4 - Inactive
Unlocked: signal is low
unlock
pw change

Figure 4: Lock Module FSM Diagram

As illustrated in Figure 4, the lock module has 6 states, but let us for the moment consider the four states "locked", "unlocked", "entry", and "inactive". When in any one of these states, the user can push "lock", "unlock" and "inactive" to transition to each of theses states. While locked, entering the correct code will transition the lock to the entry state. Once in entry state, the lock will remain unlocked until the door opens, at which point tit will re-lock itself. When inactive, the lock remains locked, but password entry is disabled – this mode would be useful to a user who has given out the code to a few close friends, but doesn't want them entering the room at a certain time. As a safeguard, if the door is opened while the lock is in inactive mode, the lock automatically returns to the locked state, preventing an accidental late-night lock-out after a visit to the bathroom.

The remaining two states deal with password entry and storage. When the lock receives a change password command, it allows the user to input a new password to be stored into the RAM. Since the lock module grabs the signal processors semaphore, it transmits a "lock done" signal to release the semaphore after the user pushes return to complete the password entry. The final state actually divides the "locked" state into two sub-states. As the user is entering a code for authentication purposes, each digit of the code is cross-referenced with the expected digit from the RAM. If the characters entered so far correctly correspond to the initial characters of the password, the FSM remains in "partially correct password" state. When the correct password has been entered, and the return key is pushed, the lock module outputs a "correct password" signal for one clock cycle before transitioning to the "entry" state. However, if the user makes a mistake at any point during password entry, the FSM transitions to “password wrong state” – where the FSM will remain until the user completes entry of the wrong password by pressing return. It that point, the FSM returns to the “locked, partially correct password” state.

As eluded to earlier, the lock sends an “unlock signal” to the electronic latch, a done signal to release the signal processor’s semaphore following a password change, a current state signal to the display, and a “correct password” signal to the burglar alarm module. Thus, the burglar alarm can be disarmed using the same password which unlocks the door. See the burglar alarm section for additional details.

## Burglar Alarm Module

--By David Nedzel

The burglar alarm module implements the primary means of security offered by the Dorm Room Genie. Its primary function is to trigger the “burglary” signal and sound an audible alarm when a burglary is detected. Like the lock module, the burglar alarm module receives control signals from the signal processing module, and users the magnetic door sensor. As shown in Figure 5, it also takes as input the current system time, the “correct password” signal from the lock module, and a “motion detected” signal from the video module. The burglar module outputs the burglary signal to many of the other modules in the system and sounds the alarm buzzer.

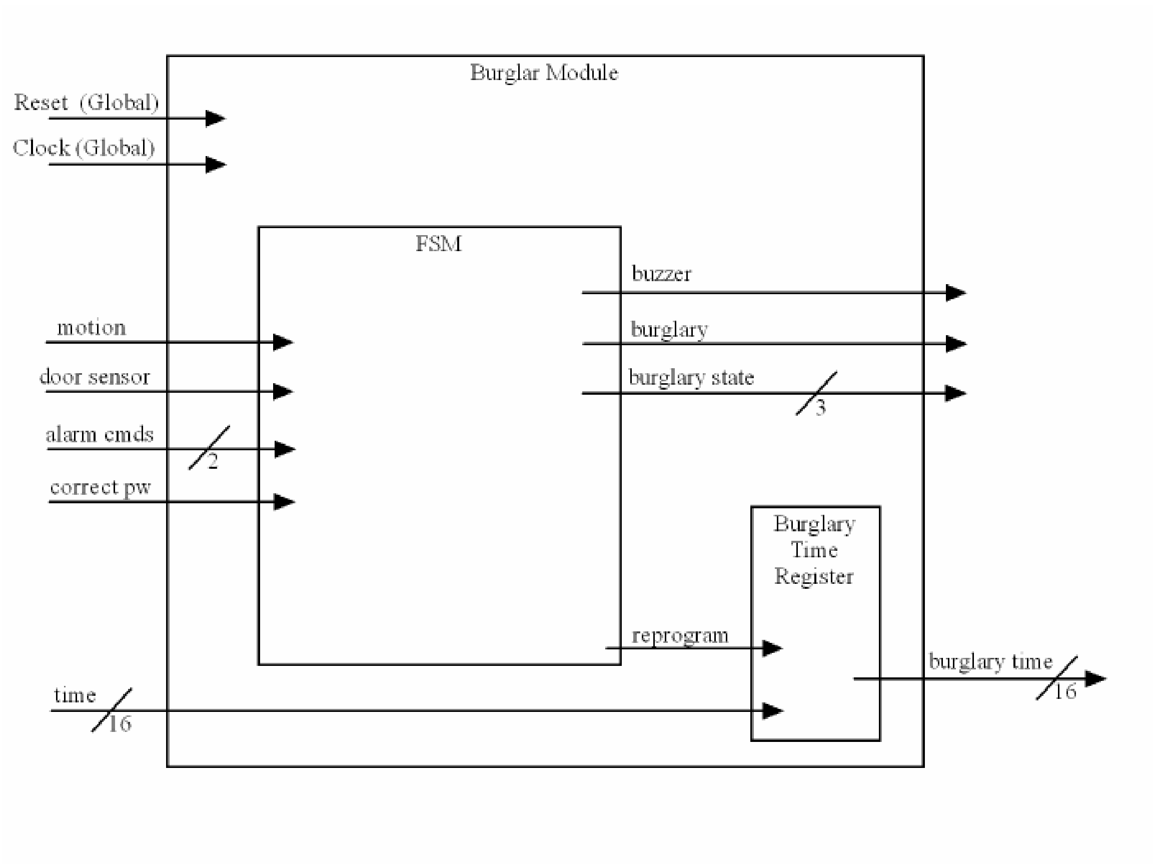

Figure 5: Burglar Alarm Module Block Diagram

At the heart of the burglary module is an FSM which controls when and how an intrusion is detected. As illustrated in Figure 6, the FSM has 5 states: unarmed, exit, two armed states, and the burglary state. When in any state other than "burglary" a user can always press the disarm button to return the alarm to the disarmed state. No matter what state it is in, the alarm will always return to disarmed when the password is entered – the "password correct" signal generated by the lock module will disarm the burglar alarm. Note that this is the only way to disarm the system during a burglary.

The user can enable the alarm in two different modes. Often times, it is useful to enable the alarm while at home, in order to alert you while you sleep of a break in. This could be useful to a student who has friends that might try to pull a prank in the middle of the night. Obviously, if one is in the room, it would not be very useful to have the motion detector enabled since the alarm would sound as soon as you moved. Therefore, if you arm the system while the door is closed, the system activates without motion detection – only opening the door will trigger an alarm. Of course when leaving the room, the user would likely wish to enable motion detection. By arming the system while the door is open, the system transitions to "exit" mode. As soon as the door closes, the system will arm itself – now, either motion or opening the door will trigger an alarm.

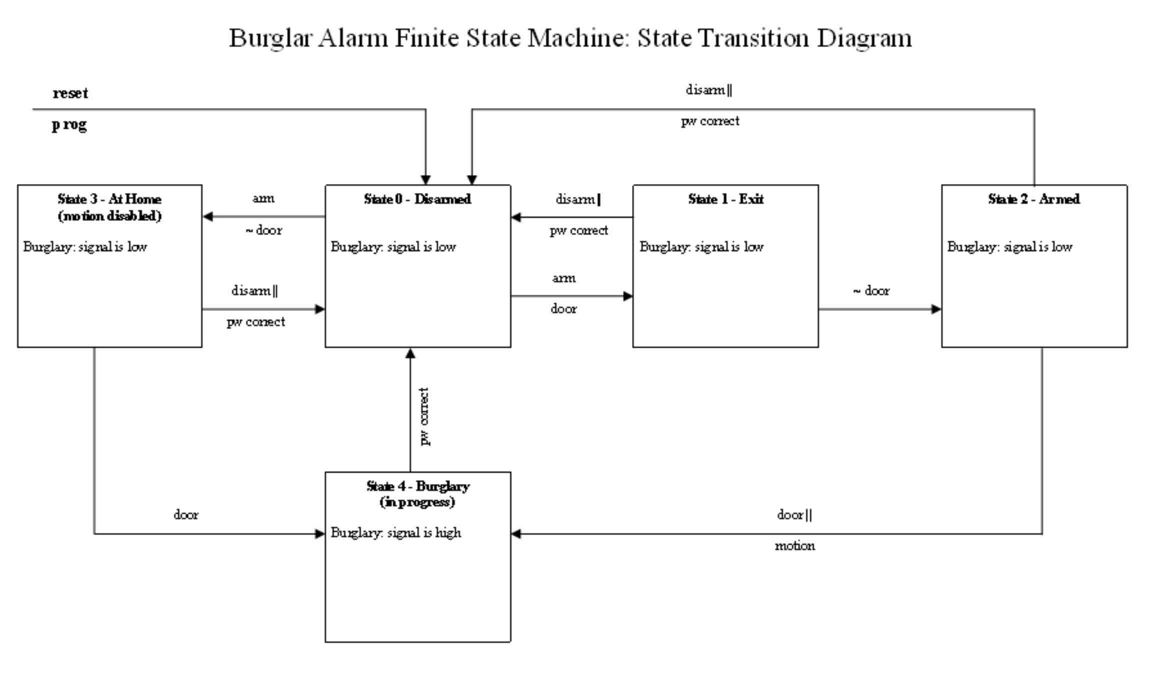

Figure 6: Burglar Alarm FSM Diagram

When an alarm is triggered, the burglar alarm module activates the burglary signal and sounds the alarm buzzer – only entering the password will disable the alarm. The burglary signal is transmitted to nearly all other modules of the system – see the other module descriptions for information on how they react to this signal. The FSM also outputs its current state to the display module.

The burglar alarm module also has the capability to store the time at which a burglary occurs. Using the same 16-bit time format as the clock, the burglar alarm module store the current system time into a register (shown in Figure 5) each time the FSM transitions to burglary. This time is subsequently output to the display module.

**Time Division Modules**
--By Aaron Stonely

The time division modules are responsible for creating the signals which drive the time-related elements of the Dorm Room Genie. The time division modules are comprised of the one hertz enable, one minute enable, and the one-sixtieth second enable modules. These time division modules are among the simplest of the modules used by the Controller Module. The time division modules take in as input a synchronized reset signal and the 27 megahertz system clock. The function of these modules is to take this signal and convert it into a signal of the appropriate length. For example, the one hertz enable module outputs an enable signal every 27 million clock cycles, which results in a second. Upon reaching the 27 millionth clock cycle, the counter resets to zero and the enable signal is disabled on the following clock cycle.

At the beginning of every clock cycle, the time division modules check to see if a reset has been asserted. If it has, the count is reset to zero, and the count begins incrementing again. If not, the time division modules continue by checking to see if the count has reached the specified value. If it has, the count is reset to zero, and the enable signal is asserted. If the count has not yet reached the specified value, the enable signal is held low, and the count is incremented. Table 9 contains a summary of the time division modules, their count, and their corresponding time value. The actual count value is one less than shown in the table, in order to account for the clock cycle delay for enable to be asserted high.

| Module | Count Value | Time Value |
|---|---|---|
| One_hz_enable | 27 million | 1 second |
| One_min_enable | 1.62 billion | 1 minute |
| Demo_enable | 500,000 | 1 sixtieth second |

Table 9

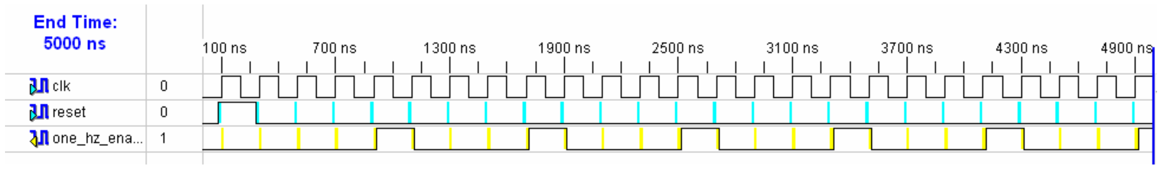

Figure 7

Figure 7 shows a timing diagram for a generic time division module. This particular timing diagram shows the behavior of a time division module that counts to three. In this case, the one_hz_enable count was temporarily set to three. This was done in order to demonstrate the behavior of the time division modules without having to simulate 27 million clock cycles in order to view a single enable signal. From the timing diagram, it is clear that the enable signal is asserted on the fourth clock cycle instead of the third. This is a significant delay when the time division module is counting to three, but when it counts to values used in this lab, the introduction of an extra clock cycle delay has a negligible effect. However, as noted, in order to account for the extra clock cycle delay, the modules count to a value that is one less than that specified in Table 9.

## Clock Module

--By Aaron Stonely

The clock module is responsible for keeping the system time. The time it outputs is used by several other modules throughout the system in order to implement functionality that depends on the current time. The clock module takes in as input the 27 megahertz system clock, the synchronized reset, a time control signal and numbers from the signal processor module, a time selector signal from the user, and a burglary signal from the burglary module. The clock module takes these signals and then outputs its current state, the system time, and a one minute enable signal to be used by other modules. Figure 8 shows the organization of the clock module.

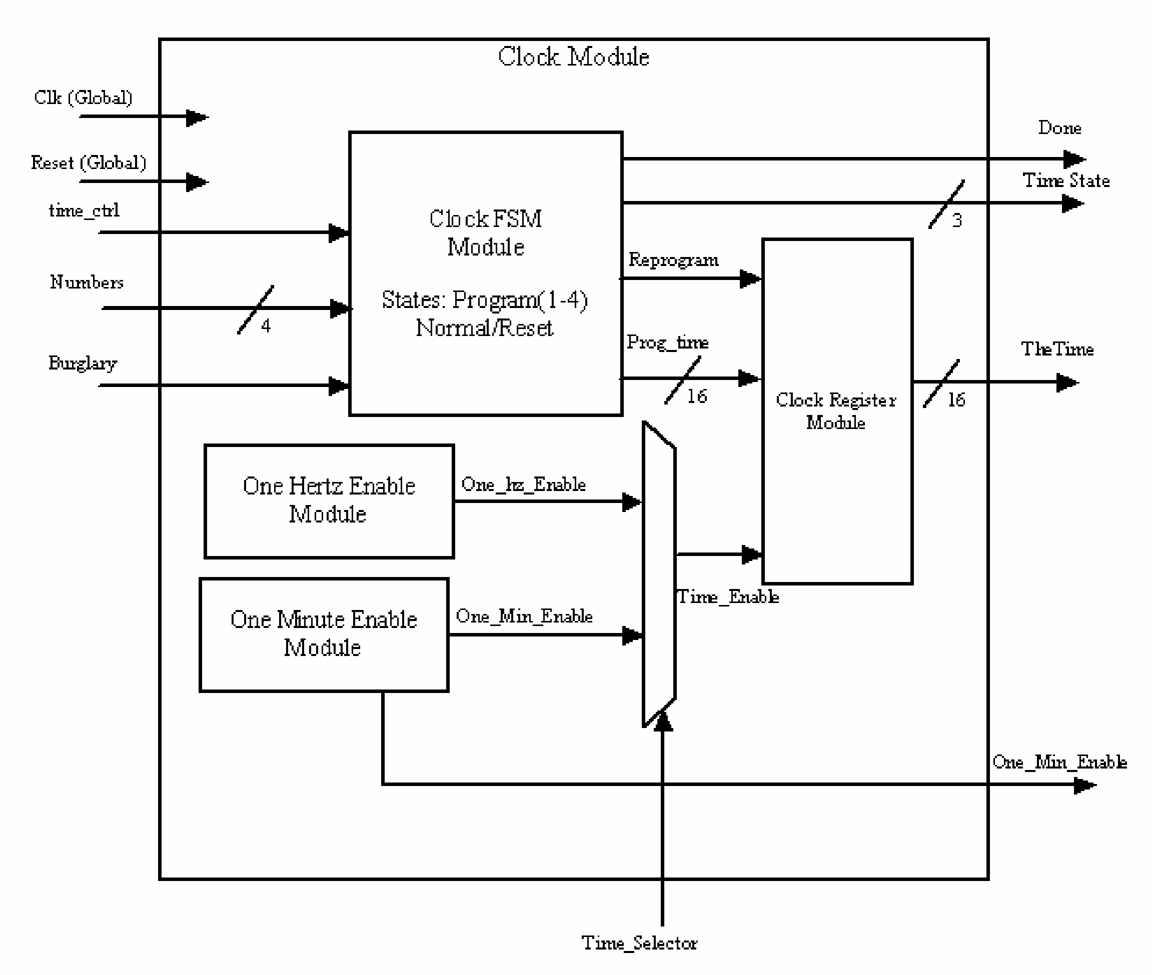

Figure 8

The clock module implements the basic functionality of a digital clock. Under normal operation, the clock module outputs the current time, and updates it when either one minute, or one second has elapsed as specified by the user. The format for the time used by the dorm room genie is twenty-four hour time, displayed as 4 hex numbers. At any time, the user can request to reprogram the current time. The user does this by first making a reprogram request, and then by specifying four valid numbers for the time to be programmed. The clock module is divided into three major sections which include the Clock Finite State Machine (FSM) section, the time keeper module section, and the time enable section.

The simplest of these sections is the time enable section. This section includes an instance of the one minute enable module and the one hertz enable module. The output of the one minute enable module is output from the clock module so that it may be used by other modules within the Controller module. The purpose of this section is to allow the user to choose whether the system keeps time in minutes or seconds. This function is primarily intended for demonstration purposes because it allows other functionality in the Dorm room genie to be demonstrated in less time than normal operation would require. For example, the wake-up lighting sequence takes place over fifteen minutes, and would be difficult to demonstrate in a short period of time unless this option were present. The time mode specified by the user is then output to the time keeper section as a time_enable signal.

The second section of the clock module comprises the clock register module. This module is basically a large register containing the system time, with some additional logic in order to increment the system time. The clock register takes in as input the global clock and reset, a reprogram signal from the clock finite state machine (FSM), a time enable signal as described previously, and the time to be programmed. The clock register uses these outputs to determine the current system time as well as the next time to be displayed. The largest portion of this module was dedicated to determining the next time to be displayed. This involved the conversion of the time enable signals and the current time into the system time which is displayed as 4 hexadecimal numbers. The system time is reprogrammed upon request from the clock FSM. If the clock register receives a reprogram signal from the clock FSM, the current reprogram time output from the FSM becomes the system time which is output from the clock module.

The third and final section in the clock module contains the clock finite state machine (FSM). The clock FSM takes in as input the global 27 megahertz clock, the synchronized reset, the time control and the 4-bit hex numbers from the signal processor module, and a burglary signal from the burglary module. The clock FSM directs the operation of the clock module. It controls the current state of the clock system as well as the programming sequence. The clock FSM has five states. The first state is the normal operating state. In this state, the FSM does not assert the reprogram signal, and the clock register module is allowed to keep the current system time as normal. Upon a reprogram request from the user, the FSM enters the program sequence. The programming sequence consists of the remaining states of the FSM, which are program 1-4. In the program 1 state, the clock FSM is waiting for the user to input the first number of the time to be programmed. In the program 2 state, the clock FSM waits for the user to input the second number of the time to be programmed, and the program 3-4 states continue this pattern. During a transition from the program four state to the normal operating state, which occurs after a successful reprogram, the clock FSM asserts a done signal, which is output from the clock module. This signal is used to let the signal processor module know that a programming sequence has been completed. One last thing of note is the function of the burglary input. During a burglary, the keyboard exits a reprogram sequence and enters a dedicated password entry sequence. If the user is in the middle of a reprogram sequence and burglary is asserted, the FSM immediately exits the reprogram sequence and returns to the normal operating state with the current time left unchanged.

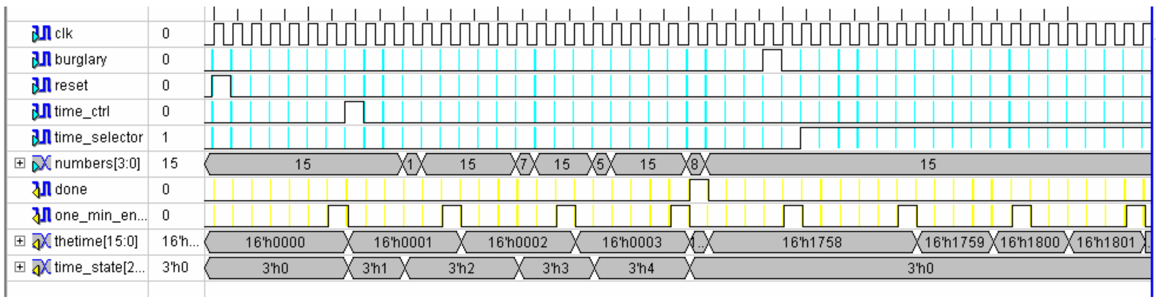

Figure 9

Figure 9 shows a timing diagram of the clock module. As shown in figure 9, the current time (labeled 'thetime') is displayed as a 4 digit hexadecimal number. Time_ctrl is the user input and when asserted high, represents a request to reprogram the current time. A reprogram sequence is also shown, in which the current time is reprogrammed to 17:58 versus the old time of 00:04. After the program sequence is completed, the done signal is asserted, and the current time becomes the programmed time. Finally, the function of the time selector is also demonstrated. Before the time selector was asserted, the time was being updated every six clock cycles. After the time selector was asserted, the time was updated every four clock cycles. This corresponds to the one_minute_enable count being reprogrammed to 5 clock cycles and the one_hz_enable count being reprogrammed to 3 clock cycles. The extra clock cycle delay is due to the clock cycle delay introduced by the clock register module updating the time.

## Wake-up Alarm Module

-- By Aaron Stonely

The wake-up alarm module will implement additional functionality to the clock module. The alarm clock module allows the user to set a wake-up time and to be gently awoken by light, even if their curtains are closed. If the user fails to acknowledge the wake-up sequence, the wake-up alarm will sound a buzzer in short bursts in order to guarantee that the student will wake up. The wake-up module takes in as input the 27 megahertz system clock, the synchronized reset, a wake control signal and hexadecimal number from the signal processor module, the current time, a one minute enable signal, a one hertz enable signal, a burglary signal, and a time selector signal from the user. The wake-up module outputs the current value of the wake-up time, a corresponding light time to be used by the lighting module, the current state of the wake-up module, a buzzer signal, a done signal, and an alarm signal.

The user interacts with the wake-up alarm module through the keyboard and the signal processor module. There are four user commands, and they include alarm on, alarm off, snooze, and set wake-up time. The user must first set the time they wish to wake-up at. After setting the wake-up time, the wake-up alarm is still inactive. The user can then choose to turn the alarm on or leave the alarm in its off state. If the user activates the alarm, the wake-up alarm module waits until the current time is within fifteen minutes of the programmed wake-up time. It is at this time that the wake-up sequence begins, and the lighting module is signaled to begin lighting through the alarm signal. As soon as the wake-up sequence begins, the user is able to snooze the wake-up alarm. This turns off the lights and gives the user another fifteen minutes of sleep. Once the current time reaches the wake-up time, the buzzer begins to sound for one second, every other second. This is a loud buzzer, and is intended to ensure the user will wake up and not sleep through an important deadline. The organization of the wake-up module is shown in figure 10.

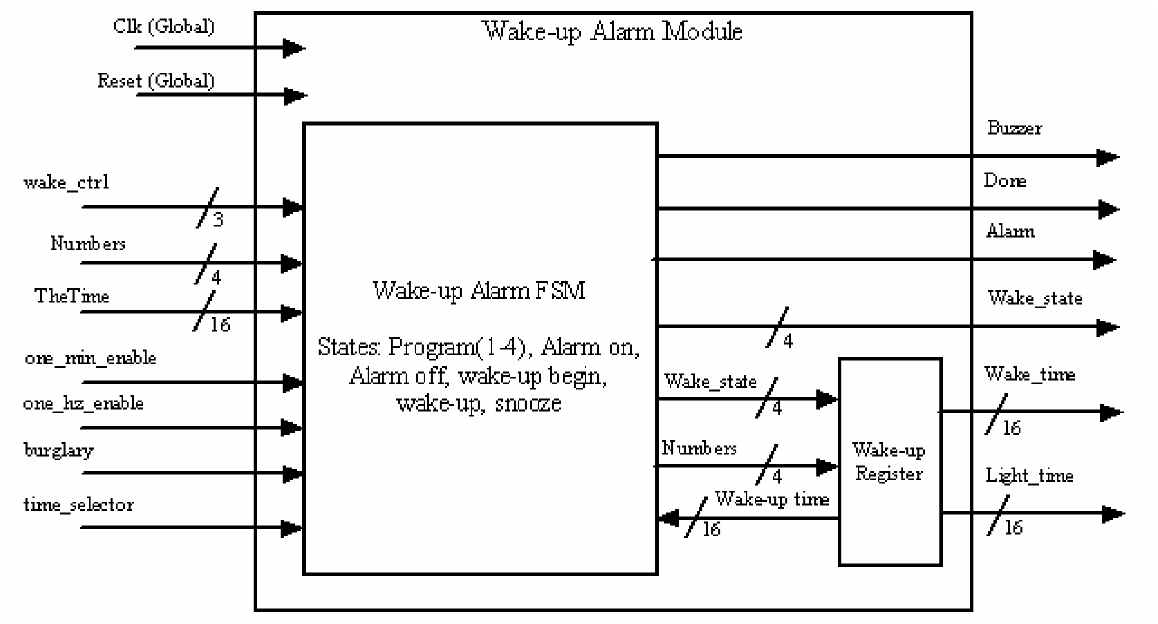

Figure 10

The wake-up alarm module is split into two major sections. These sections are the wake-up alarm finite state machine (FSM) and the wake-up alarm register. The wake-up alarm register stores the most recently programmed wake-up time. During a reprogram, the register stores the number input from the user into the appropriate slot in the wake-up time. The wake-up register then outputs a wake-up time which is used to calculate the wake-up time. Most of the work for this part consisted of creating the logic needed to calculate the value of the wake-up time minus fifteen minutes in twenty-four hour hexadecimal format used. Once calculated, this value is output as the light time.

The second and largest section of the wake-up alarm module is the wake-up alarm finite state machine (FSM). The FSM is responsible for implementing the various functions of the wake-up alarm. It takes uses the current state and the current time to determine whether the alarm should be going off. The wake-up alarm module has several states. These states include the normal operating or 'off' state, the program 1-4 states, the "alarm on" state, the "wake-up sequence begin" state, the "wake-up active state", and the "snooze" state. Transitions between these states are governed by the user inputs from the signal processor module, which include no input, the "alarm on", "alarm off" "snooze" and "set wake-time" commands.

During the normal or "alarm off" state, the alarm will not go off. Instead, the wake-up module simply stores the last wake-up time value programmed by the user. If the user inputs a "set wake-time" command from the signal processor module, the wake-up alarm finite state machine (FSM) transitions to the program 1 state. In this state, the FSM waits for the user to input a valid number for the first number in the wake-up time to be programmed. Program states 1-4 all follow this pattern, just as in the clock module. Once the last valid number is programmed in the wake-up time by the user, the FSM asserts the done signal so that the signal processor module knows that a successful reprogram sequence has terminated. Once again, if the burglar signal has been activated during a program sequence, then the FSM aborts the programming sequence and returns to the default or "alarm off" state.

If the user issues an "alarm on" command from the signal processor module, the wake-up finite state machine (FSM) then transitions to the "alarm on" state. In this state, the user can turn off the alarm or choose to reprogram the wake-up time, both of which eventually leave the FSM in the "alarm off" state. While in the "alarm on" state, two types of behavior may be observed. If the current time is less than fifteen minutes from the programmed wake-up time when the user turns the alarm on, then the interval is too short for a lighting wake-up, and instead only the buzzer will be used to wake-up the user.

If when the user turns the alarm on, the current time is more than fifteen minutes from the programmed wake-up time, then the user has the option to wake-up to a lighting sequence, which is selected by the user in the lighting module. In this case, once the time reaches the time for the lighting sequence to begin, the wake-up module finite state machine (FSM) transitions to the "wake-up sequence begin" state. Without input from the user, the FSM will remain in this state until the current time reaches the user-programmed wake-up time. Once the wake-up time is reached, the FSM enters the "alarm active" state. While in the "wake-up sequence begin state", the user has the option of turning the alarm off or pressing the snooze command. If the user turns the alarm off, the lights will turn off, and the alarm will be deactivated. If the user chooses to snooze, the alarm will give the user another fifteen minutes before the FSM transitions to the alarm active state. In addition to halting the lighting sequence, the lights are turned off since a snooze request during the wake-up sequence demonstrates that the user does not want the lights on. If the dorm room genie is in demonstration mode (shorter time intervals) then the snooze only lasts for duration of fifteen seconds.

When in the “alarm active” state, the wake-up module finite state machine (FSM) is sounding the buzzer. The buzzer is activated for duration of one second every other second in order to ensure the user wakes up and acknowledges the alarm. When in this state, the user may choose to turn the alarm off, or snooze the alarm. If the user commands the alarm off, the alarm will transition to the “alarm off” state, and the buzzer will stop sounding. If the user commands the alarm to snooze, then the lights will once again shut off, and the user will get another fifteen minutes to sleep before the alarm becomes active once again and the buzzer starts sounding. Once again, if the dorm room genie is in demonstration mode, the snooze will only last fifteen seconds.

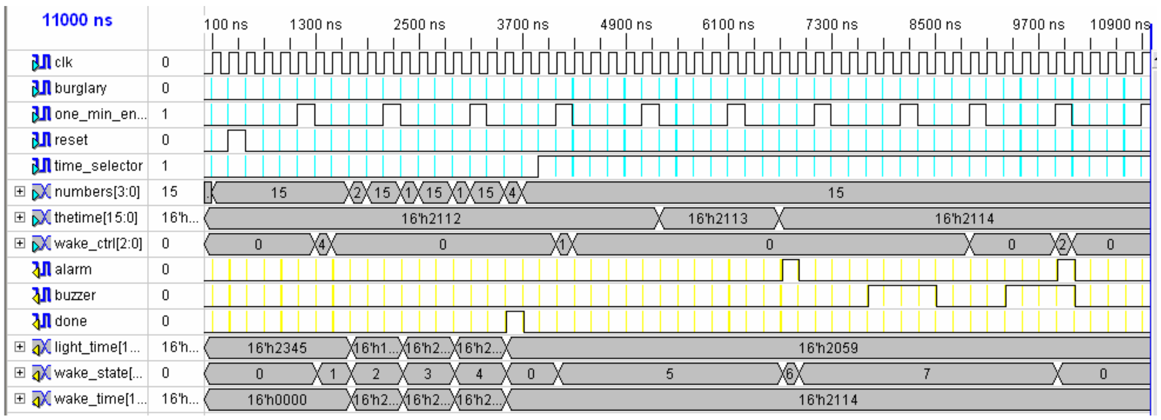

Figure 11

Figure 11 contains the timing diagram of the wake-up alarm module. Wake_ctrl is the user input from the signal processor module. A value of 4 represents a reprogram request, a value of 1 represents an “alarm on” request, a value of 2 represents an “alarm off” request, and a value of 3 represents a snooze request. As shown, the wake-up time is update as the user puts the numbers in. Since the time is static and does not change as the user inputs a new time to be programmed, there was no need to delay the reprogram, so the wake-up time is updated as the inputs are received. Note how alarm is asserted once the current time reaches the programmed wake-up time and again when the user turns the alarm off. This is used to tell the lighting module when a lighting sequence should begin, and when the lighting sequence should be terminated.

## Lighting Module

--By Aaron Stonely

The lighting control module is designed to give the user even greater control of their dorm room. The lighting module adds another level of functionality to the wake-up alarm, as well as three other modes of functionality. The lighting module takes in as input the 27 megahertz system clock, the synchronized reset, lighting control commands from the signal processor module, the alarm signal from the wake-up module, the burglary signal from the burglary module, the time selector input from the user, a one hertz enable signal, a demo enable signal, and photo-sensor data containing the ambient lighting value. The lighting module outputs the lighting module state, and the lighting value. The lighting value is an 8-bit number as is used to represent a lighting value when converted using a DAC0831 digital audio converter chip.

The lighting module has four operating modes which include “manual mode”, “automatic mode”, “sleep mode”, and “burglary mode”. In “manual” mode, the user can manually adjust the lighting level. This includes the ability to turn the lights on and off, as well as the ability to brighten or dim the lights. This implementation allows eight discrete lighting levels which the user can select between using the brighten and dim commands. In “automatic” mode, the light intensity is controlled by the level of ambient light as determined by a photo-sensor and the ADC0841 inputs. This lets the system adjust the lights in order to maintain a certain level of brightness. When in “sleep” mode, the lighting module works with the wake-up module in order to wake up the user with a lighting sequence. Finally, when a burglary occurs, the lighting module enter “burglary” mode which flashes the lights on and off in order to attract additional attention to the burglar with the goal of deterring theft. Figure 12 shows the organization of the lighting module.

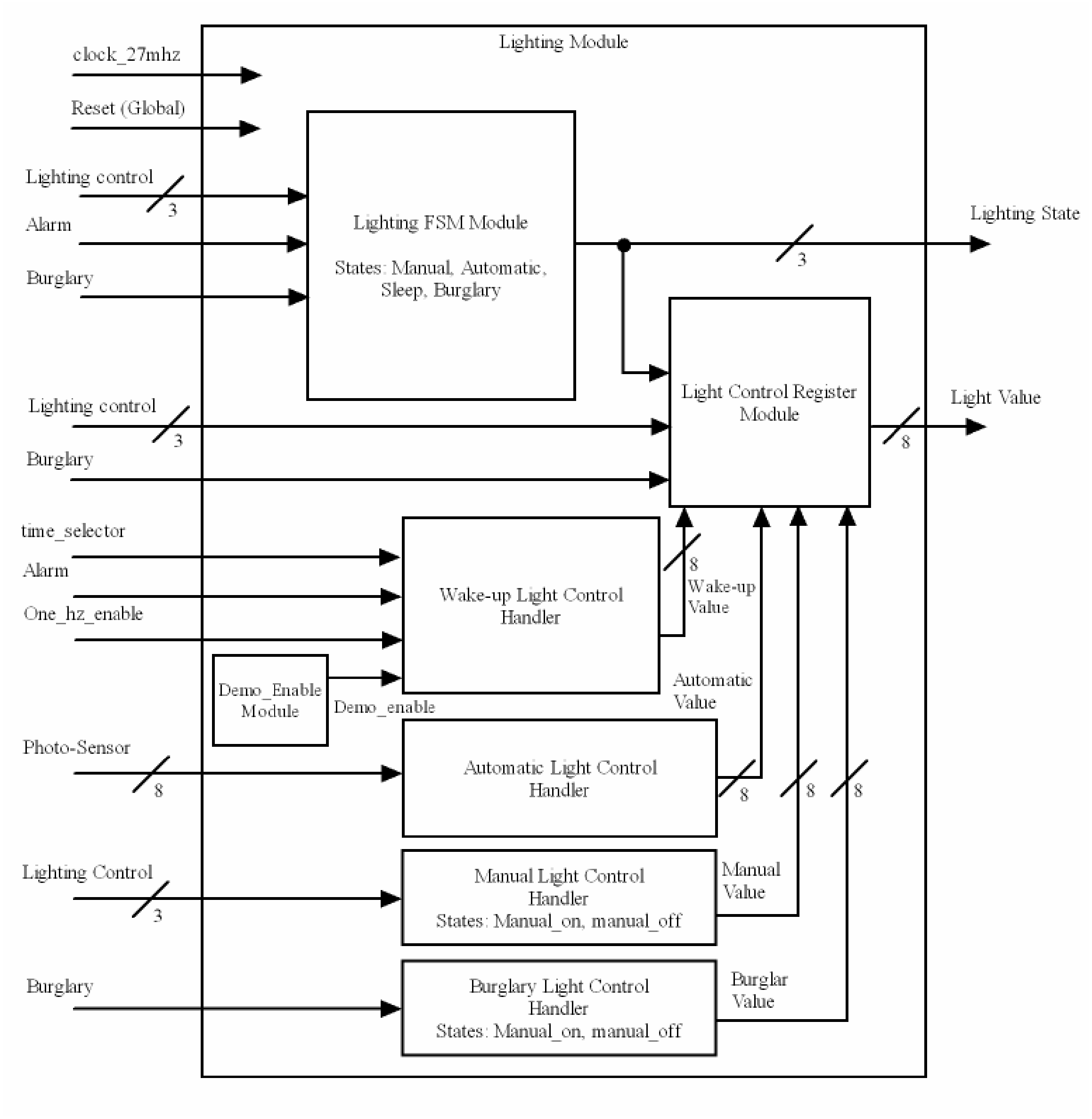

Figure 12

The lighting controller is divided into five major sections. These include the wake-up, automatic, manual, and burglary light control handlers, the light control register, and the lighting finite state machine (FSM). The user interacts with the lighting module through the keyboard control commands which come from the signal processing module. These commands include no input, ‘off’, ‘on’, ‘automatic’, ‘brighten’, ‘dim’, and ‘sleep’.

<u>Lighting Module Finite State Machine</u>

The lighting module finite state machine (FSM) interprets the user command inputs, and uses these to maintain the state of the lighting module, which in turn determines which lighting value is selected to be output to the rest of the system. The lighting module FSM has four states. These states include the ‘manual’, ‘automatic’, ‘sleep’, and the ‘burglary’ operating state. Transitions between these states are fairly straightforward. For example, if the FSM is in any other state than manual, if the user inputs a manual command such as ‘on’, ‘off’, ‘dim’, or ‘brighten’, the FSM transitions to the ‘manual’ state. Once in the ‘manual’ state, the FSM remains there until the user issues a non-manual command. The user can transition between the ‘manual’, ‘automatic’, and ‘sleep’ states (or modes), at any time with the corresponding user command. The FSM only enters the burglary state during an active burglary. Once the burglary is over, the FSM transitions to the reset or ‘manual’ operating state.

## Wake-up Light Control Handler

The wake-up light control handler is responsible for implementing the additional functionality associated with the wake-up alarm. Once the time has approached fifteen minutes before the programmed wake-up time, the wake-up module signals the light control module to begin the wake-up sequence. As described earlier in the wake-up module, the user has the option of making the sequence begin fifteen seconds before the programmed wake-up time. The start of the wake-up sequence causes the lighting control module to use the lighting value provided by the wake-up control handler.

At the beginning of the sequence, the lights are off. The lighting module then gradually increments the lighting value so that the lighting level is its maximum value as the time reaches the programmed wake-up time. This is done by incrementing the lighting value by one every three time units until the maximum value of 255 is achieved. The wake-up handler uses a sub-module called demo enable. The demo enable module takes in the system clock and the reset and outputs an enable signal once every sixtieth of a second. This module is needed in order to properly demonstrate the lighting sequence when in demonstration mode (minutes lasting one second). When not in demonstration mode, the lighting sequence lasts fifteen minutes instead of fifteen seconds. During a lighting sequence, the user may choose to snooze or shut off the wake-up alarm. If the user does so, the wake-up alarm module asserts the alarm signal, and the wake-up light control handler aborts the lighting sequence and shuts off the lights.

## Automatic Light Control Handler

The automatic lighting control functionality is designed to handle the lighting for the user without the user having to adjust it as the room gets lighter or darker. The automatic light control handler does this through the use of the light sensor which outputs a voltage that is converted to an 8-bit number through the use of the ADC0841. The automatic light control handler then interprets this lighting value and adjusts the lighting level accordingly. Due to the properties of the lighting sensor, a lighting threshold was set for about half of the 8-bit light range. This is because the user would likely want the lighting at full intensity even if it wasn’t pitch black in the room. When the light level falls below this threshold, the lights are set to full intensity. Otherwise, the lighting value is set to the maximum lighting value minus the level of ambient light in the room. This results in the behavior that when the room gets brighter, the lights dim, until eventually they aren't visible.

## Manual Light Control Handler

The manual light control handler is responsible for implementing the on, off, brighten, and darken command functionality. The manual light control handler is basically a register for the manual lighting value with additional logic. The manual control handler stores the current value of the lighting, and allows the user to turn the lights off to the minimum intensity, on to the maximum intensity, and uses brighten or dim to reach the six other lighting intensities in between. The on and off command implementations are fairly straightforward and simply adjust the lighting to the value 0 or 255, which correspond to the minimum and maximum lighting intensities respectively. The brighten and darken commands increase and decrease the lighting value by 32 so that there are 8 discrete lighting values. This is because many lighting values are virtually indistinguishable, so 8 discrete lighting values allows a good amount of user control while maintaining discernable lighting levels. When in manual mode, the resulting manual lighting value is output as the lighting value from the lighting module.

## Burglary Light Control Handler

The burglary control handler takes care of the behavior associated with a burglary in progress. When there is a suspected burglary in progress, the lighting control FSM transitions to burglary mode, and the lighting value output from the burglary light control handler is used as the output from the light controller module. During a burglary in progress, the lights flash on and off for one second duration every two seconds. This means that the lighting value will change between 0 and 255 when the lights flash off and on respectively.

## Light Control Register

The light control register determines which of the handler's lighting values is output as the lighting value of the system. The light control register takes in the current state of the lighting finite state machine, and selects the output that corresponds to the current state or mode of operation.

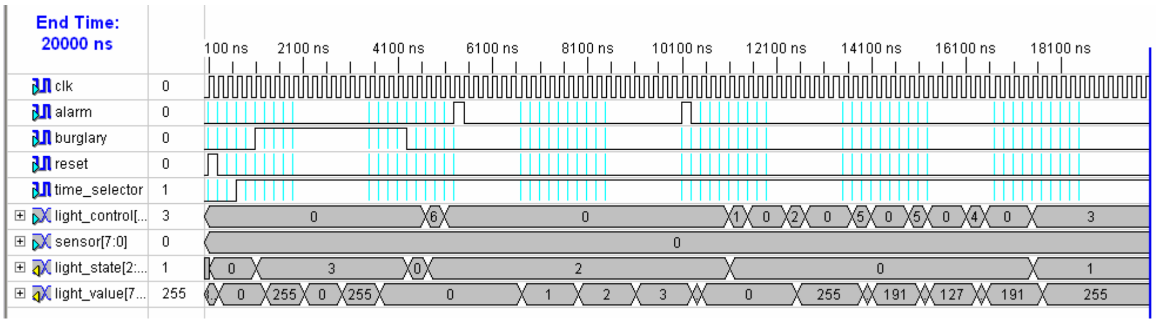

Figure 13

Figure 13 shows the timing diagram of the light control system. Displayed are the burglary, automatic, wake-up, and manual modes of operation. The result is shown as light_value which ranges from 0 to 255 corresponding to the minimum and maximum lighting intensities respectively.

**Display Module**
-- By Aaron Stonely

The lighting module is responsible for displaying the status of the entire controller system. This allows the user to see what settings are in place and what module they are currently sending input to. The display module takes in as the 57 megahertz system clock, the synchronized reset, the system time from the clock module, the burglary time from the burglary module, the wake-up time from the wake-up module, vcount and hcount from the video module, the light value from the lighting module, and the states of the clock, signal processor, burglary, lock, wake-up, and lighting modules. For all these inputs, the display module only outputs a 3-bit wide pixel value which corresponds to the values of the hcount and vcount inputs.

First, all the inputs to the display module are synchronized with the system clock using the synchronize module provided by the 6.111 course staff. This is necessary because the inputs are coming from the modules which are operating at 27 megahertz. One the inputs pass through the synchronizer module, they are now ready to be used by the logic operating at 50 megahertz. Next, the inputs are then used by logic in order to determine the display.

The display module makes heavy use of the character string display module provided by the 6.111 course staff. One instance of character string display module and its corresponding font rom was created for each line of text that was displayed. For each status line in the display module, logic was created in order to generate the string to be displayed according to the value of the input. Once the string was generated, an instance of the character string display module was created which would use this string to create a pixel value for the appropriate values of hcount and vcount. The generated pixel would have a 3-bit value of 111 when the hcount and vcount fell within a letter in the string, and a 3-bit value of 000 when hcount and vcount were outside of the display window. Each instance of the character display module had a specified coordinate on the display string, so that the information could be organized, and displayed without text overlapping. The character string display module provided by the 6.111 course staff was slightly altered so that it would support the pipelining of pixels which will be described further in the testing and debugging section. Once the pixel is output from the character display module, it again passes through a register in order to pipeline the logic associated with the pixel. The organization of the display module is in figure 14.

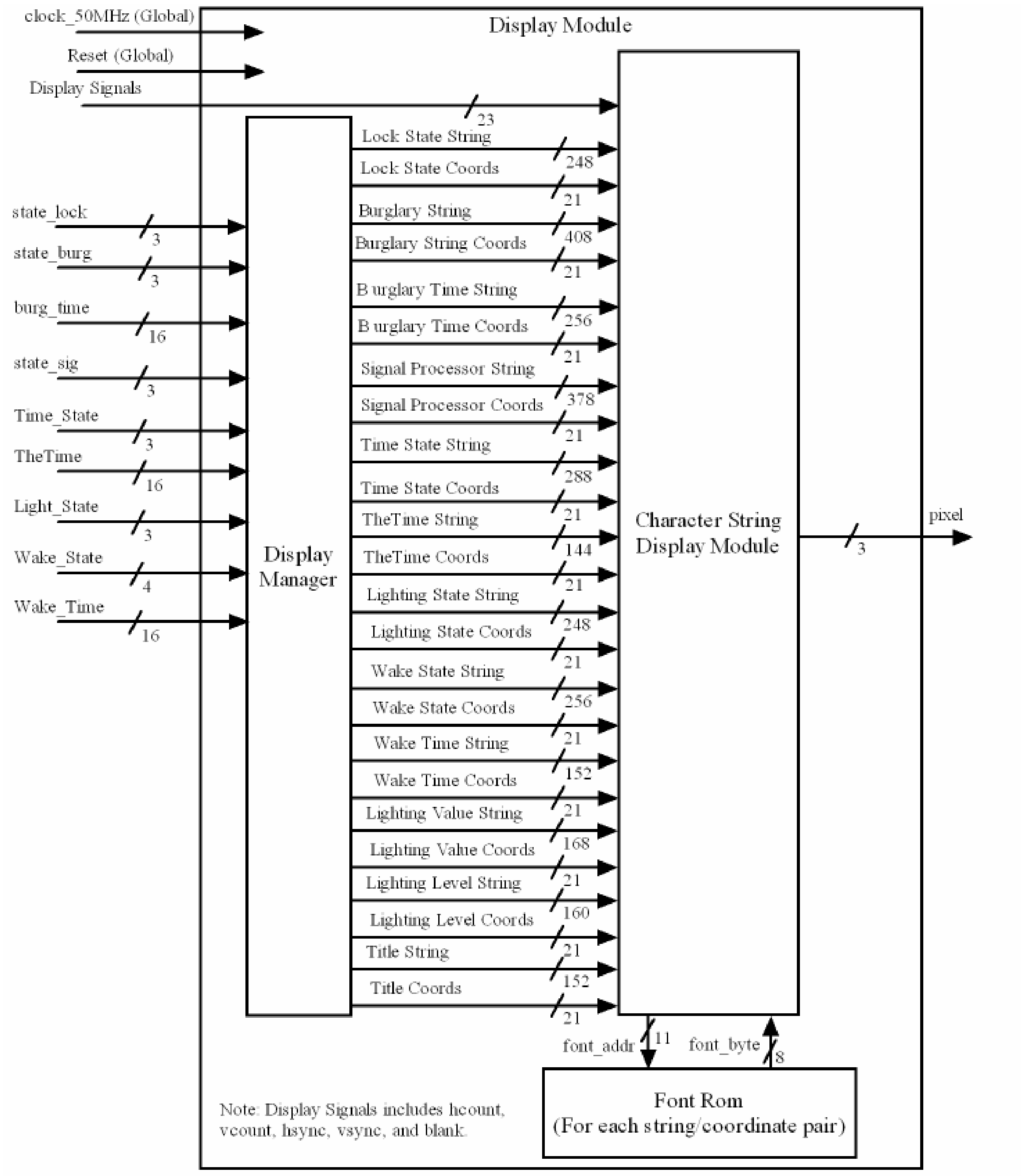

Figure 14

The number of bits in each string varies since it depends on the length of the string that is displayed for a given input. Each display string also has an associated set of coordinates which are used to determine the position of the string on the display screen. Tables 10 and 11 contain a list of all the display outputs for each corresponding input and their relative position on the screen. The display module is designed to display all the text on a screen resolution of 640 x 480.

| Display Item | Associated Strings |
|---|---|
| Title | “The Dorm Room Genie” |
| System Time | “System Time: (HH:MM)” |
| System Time State | “System clock is in operational mode.”<br>“Please program the current time.” |
| Wake-up Time | “Wake-up Time: (HH:MM)” |
| Wake-up State | “Wake-up Alarm is off.”<br>“Please program the wake-up time.”<br>“The wake-up alarm is on.”<br>“The wake-up sequence has begun.”<br>“Time to wake up!!”<br>“Alarm is in snooze mode.” |
| Signal Processor State | “System is in normal operating mode.”<br>“The clock module is waiting for input.”<br>“The wake-up module is waiting for input.”<br>“Burglary in progress! Enter password to disarm.” |
| Lock State | “The door is locked.”<br>“”Please enter.”<br>“The door is unlocked.”<br>“Keyless entry is inactive.”<br>“Please enter your new password.” |
| Burglary State | “Burglar alarm is disarmed.”<br>“Burglar alarm is in standby: Please exit now.”<br>“Burglar alarm is armed.”<br>“Burglar alarm is armed. (Motion detection disabled.)”<br>“Burglary in progress!!” |
| Burglary Time | “A burglary has not occurred.”<br>“A burglary occurred at: (HH:MM)” |
| Light State | “Lighting is in manual mode.”<br>“Lighting is in automatic mode.”<br>“Lighting is in sleep mode.”<br>“Intruder alert! Intruder alert!” |
| Light Value | “Light value: (8-bit binary number)” |
| Light Level | “Light Level: 7 (Max)”<br>“Light Level: (1-6)”<br>“Lights are off.” |

Table 10

| Display Item | Inputs | (Horizontal, Vertical) Coordinates |
|---|---|---|
| Title | N/A | (248, 20) |
| System Time | [15:0] TheTime | (100,260) |
| System Time State | [2:0] Time_state | (100,280) |
| Wake-up Time | [15:0] wake_time | (100,300) |
| Wake-up Time State | [3:0] Wake_state | (100,320) |
| Signal Processor State | [2:0] state_sig | (100,60) |
| Lock State | [2:0] state_lock | (100,100) |
| Burglary State | [2:0] state_burg | (100,160) |
| Burglary Time | [15:0] burg_time | (100,180) |
| Light State | [2:0] light_state | (100,400) |
| Light Value | [7:0] light_value | (100,420) |
| Light Level | [7:0] light_value | (100,444) |

Table 11

**Video/Motion Module**
--By David Nedzel

The most complicated of all the modules of the Dorm Room Genie, the video module is responsible for sensing motion in the room, capturing images of the burglary, and displaying them back upon request. As shown in Figure 15, the video module makes use of several labkit features. In explaining the functionality of the video module, I will follow the track of the video signal from camera input to display output.

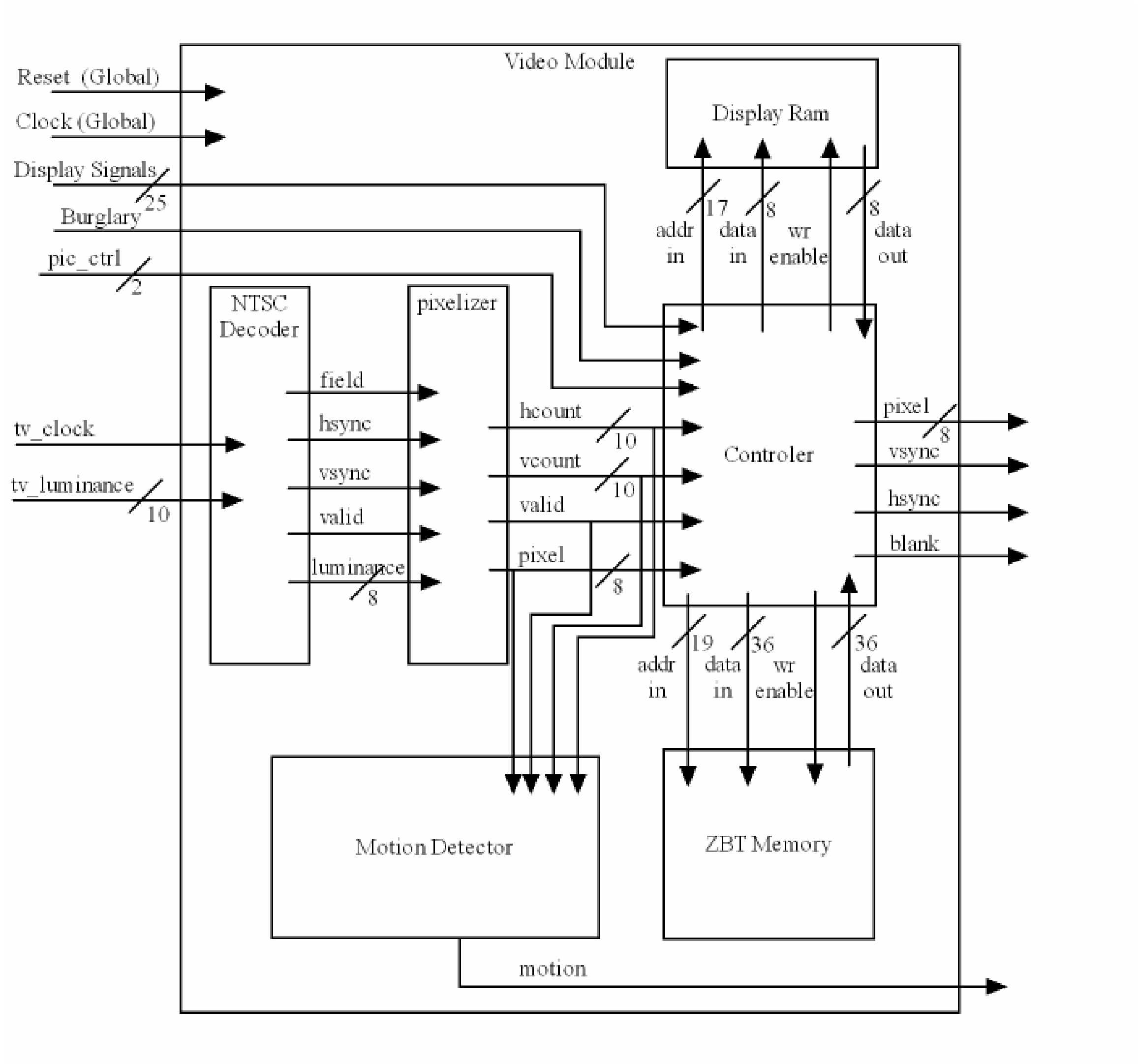

Figure 15: Video Module Block DiagRAM

When the tv clock and luminance data first enter the video module, the pass through the 6.111 provided NTSC Decoder and are transformed into field, horizontal and vertical sync signals, luminance values, and a valid signal. In order for these signals to be used effectively, they must be further transformed to indexed pixels. The pixilizer, a modified version of some 6.111 sample code performs this function. The pixilizer interprets the fvh and valid signals to select the upper left 320x240 pixels in the camera's field of view. I choosing the resolution of the snapshots, there were several considerations. First, they had to be sharp enough, to ensure that you would probably be able to identify the burglar. Second, they had to be small enough so that several could be stored in memory as a record of what happened during a specific time period. By using grayscale video and reducing the image resolution to 320x240 we can reasonably balance these two conflicting objectives. Once the pixels have been indexed, they are sent to both the motion detector unit, and the video capture and playback unit.

The purpose of the video capture and playback unit is to make a record of the burglary so that the user might learn who is responsible for missing items, or how the event happened. Having settled on 320x240 resolution, it was a simple matter to calculate how many images the ZBT memory could store. Having 2^19 indexes of 36 pixels each, there is enough room to store (2^19)/4 = 2^17 pixels, since each pixel has eight bits of luminance data. Each 320x240 image is 76,800 pixels, or 19200 index slots of the ZBT. (2^19) index slots ÷ 19,200 index slots per image ≈ 27 images. By capturing these images every two seconds, we can document roughly the first minute of the intrusion.

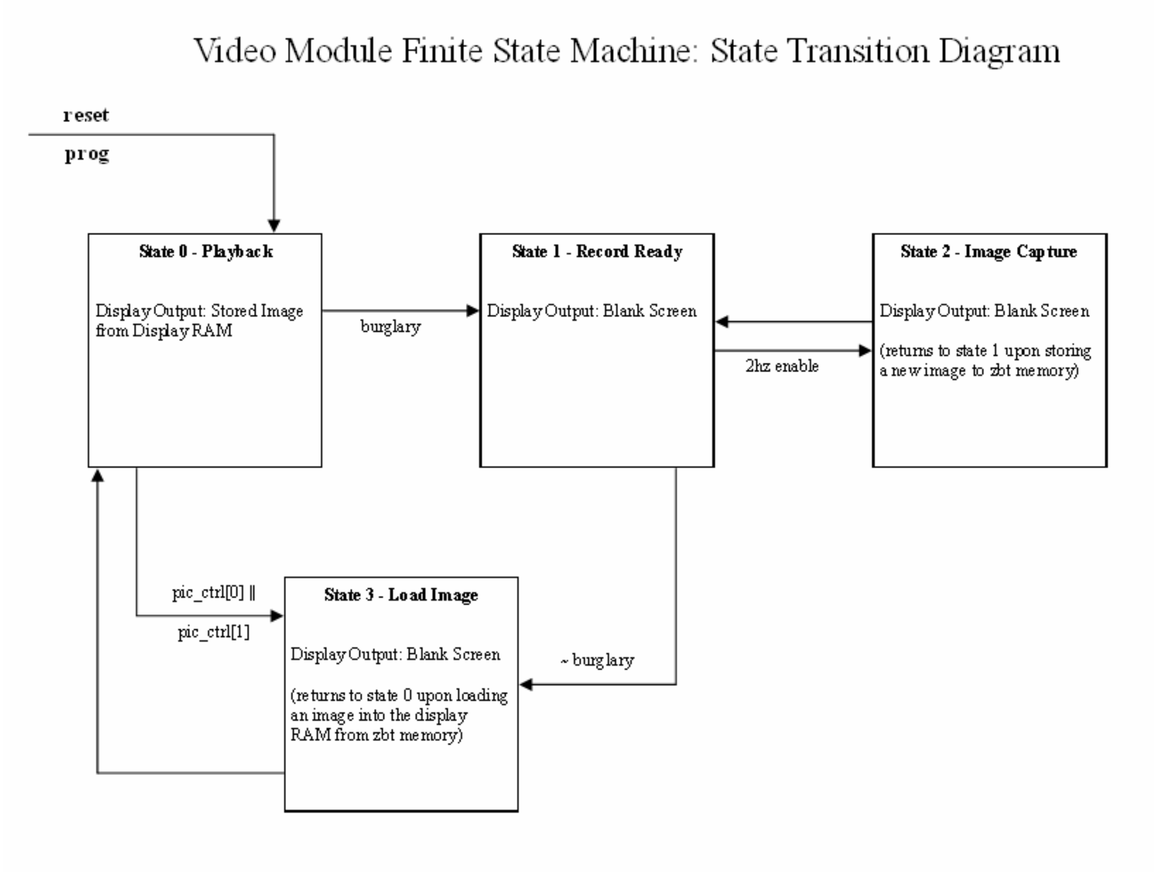
Video Module Finite State Machine: State Transition Diagram
reset
prog
State 0 - Playback
Display Output: Stored Image from Display RAM
burglary
State 1 - Record Ready
Display Output: Blank Screen
State 2 - Image Capture
Display Output: Blank Screen
(returns to state 1 upon storing a new image to zbt memory)
2hz enable
pic_ctrl[0] ||
pic_ctrl[1]
State 3 - Load Image
Display Output: Blank Screen
(returns to state 0 upon loading an image into the display RAM from zbt memory)
~ burglary

Figure 16: Video Module FSM DiagRAM

After the burglary ends, the controller then pulls an image out of the ZBT memory and loads it into the display RAM for display on the monitor. One of the complicated design elements of this module is that the components are clocked at 27MHz, except for the display RAM and video signals which are clocked at 50MHz. The still image is displayed on the monitor until the user requests the next image. Like many of the other modules, the video capture module uses an FSM to manage its actions. As illustrated in Figure 16, the FSM has four states: playback, record ready, image capture and load image. Driving the FSM to change states are the burglary signal from the burglar alarm module, the picture control signal from the signal-processing module, and 2 second enable signal created by a clock divider. While in playback mode, the user can request a new image by pushing next/previous picture. The FSM transitions to load to load the requested picture into the display RAM. Once the picture has been loaded, the FSM transitions back to playback mode. When a burglary occurs, the FSM transitions to “record ready.” Then, so long as there is sill memory space available, the FSM transitions to “image capture” every two seconds to store an image into the ZBT, and then immediately back to “record ready”. Once the burglary ends, the FSM transitions to load, and finally back to playback.

The final component to the video module is the motion detector. In principle, as an object moves from location to location in a picture, the average luminosity of pixels in a particular area should change as that object moves. The motion detector attempts to leverage that principle by averaging the luminosity of groups of pictures, and looking for deviations in those averages.

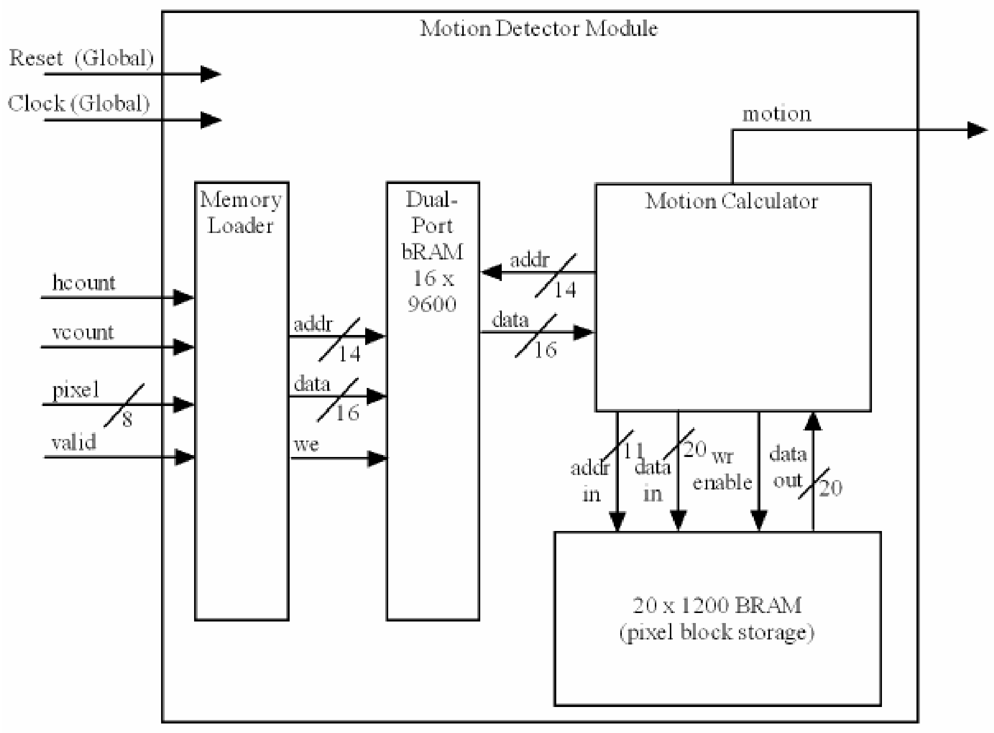

Figure 17: Motion Detector Block DiagRAM

As illustrated in Figure 17, the motion detector takes indexed pixels as input. The memory loader sums rows of eight pixels and stores them into the dual port RAM. Then the motion calculator sums adjacent rows, such that it then stores luminosity sum of an 8x8 pixel grid. Finally, as the motion calculator is updating pixels averages in the pixel-block RAM, it measures the amount of change. If that change is above a certain threshold, the module signals that motion has been detected. This threshold would be found experimentally after successfully implementing the pixel averaging functions.

**Labkit Module**
--By David Nedzel and Aaron Stonely

The labkit module is the top level file in the Dorm Room Genie. The labkit file contains both the entire controller module and the video modules that are used to display the status of the system, as well as capturing a snapshot of the burglar. The purpose of this module is to create an instance of each of the sub-modules described and wire them up together. The labkit module acts as a bridge between the 6.111 FPGA labkit hardware and the verilog implementations in the sub-modules. Here, all the user inputs are assigned to switches, buttons, and the user I/O's. Also, the outputs sent through the user I/O's and the LED displays are assigned. All of the user inputs are synchronized to the system clock and debounced so that they may be used by the modules. The labkit also contains the code provided which creates and assigns the video signals which are used by our displays. Figures 18 and 19 show the organization of the labkit.v file and all of the modules declared within it.

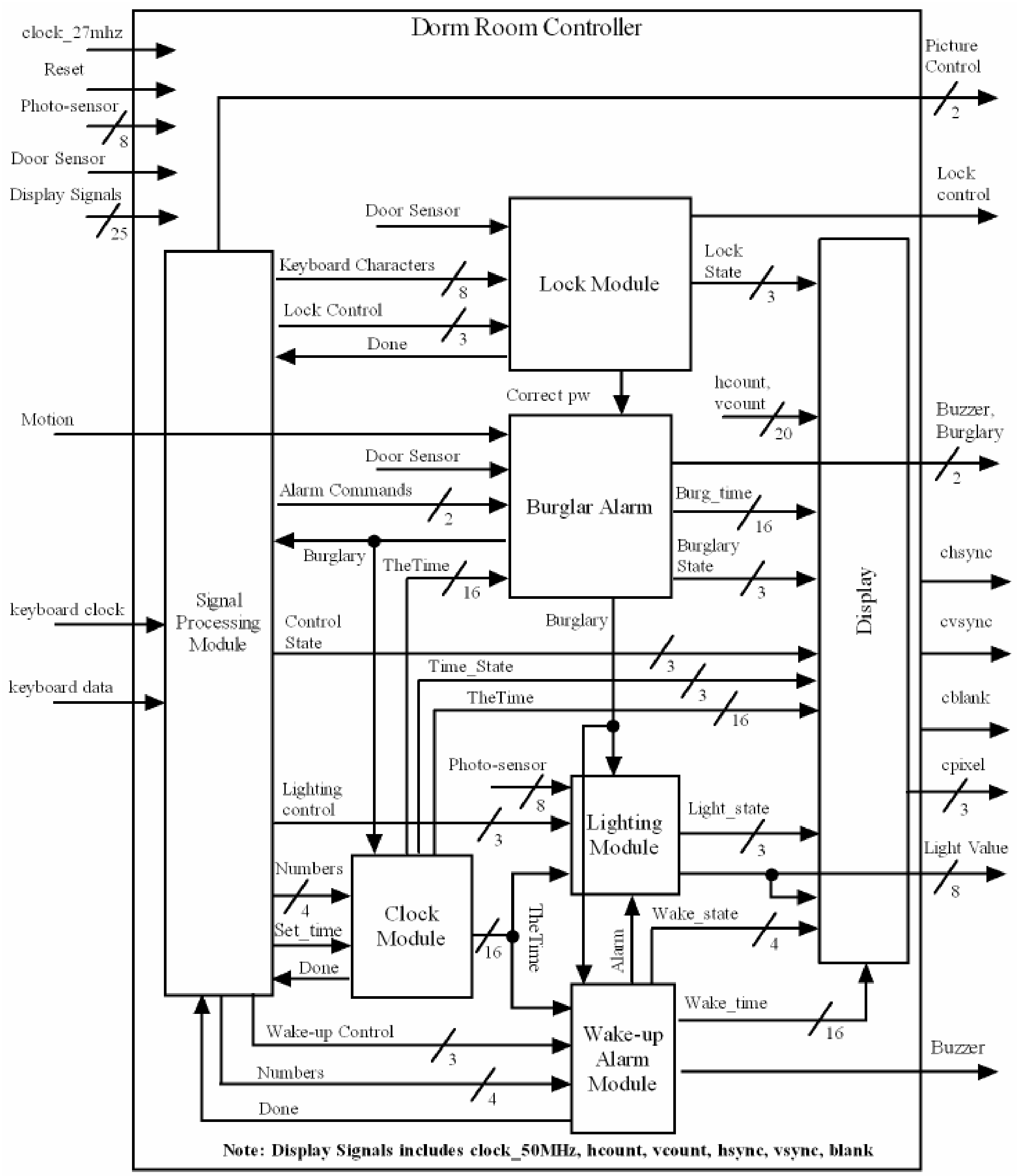
Dorm Room Controller
clock_27mhz
Reset
Photo-sensor
8
Door Sensor
Display Signals
25
Picture Control
2
Lock control
Door Sensor
Keyboard Characters
8
Lock Control
3
Done
Lock Module
Lock State
3
Correct pw
hcount, vcount
20
Motion
Door Sensor
Alarm Commands
2
Burglar Alarm
Buzzer, Burglary
2
Burg_time
16
Burglary
TheTime
16
Burglary State
3
chsync
cvsync
cblank
cpixel
3
Light Value
8
Buzzer
keyboard clock
keyboard data
Signal Processing Module
Control State
Burglary
Time_State
3
3
TheTime
16
Display
Photo-sensor
8
Lighting control
3
Lighting Module
Light_state
3
Numbers
4
Set_time
Done
Clock Module
16
TheTime
Alarm
Wake_state
4
Wake_time
16
Wake-up Control
3
Numbers
4
Done
Wake-up Alarm Module
Note: Display Signals includes clock_50MHz, hcount, vcount, hsync, vsync, blank

Figure 18

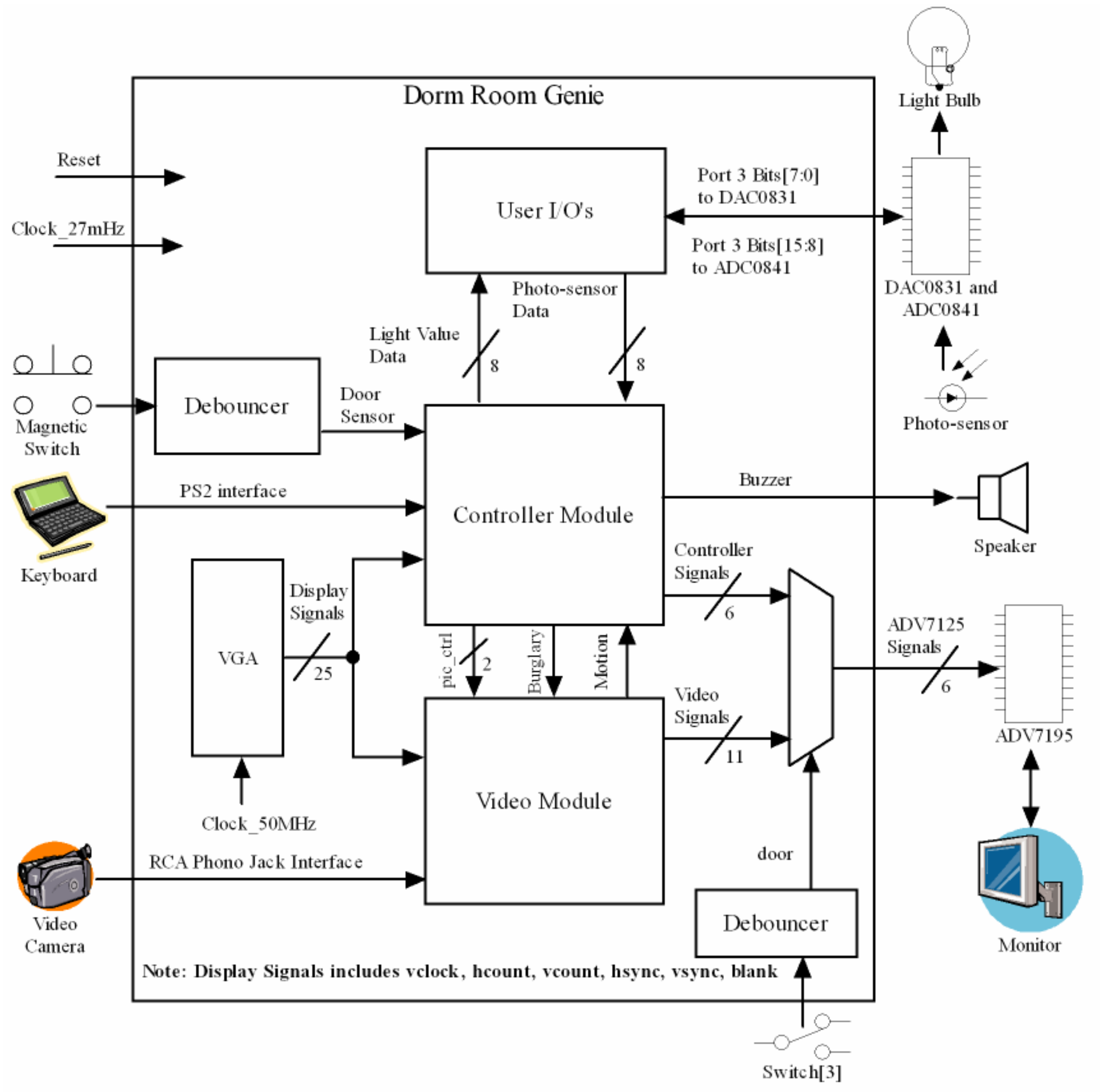

Figure 19

## Testing and Debugging

Section by Aaron Stonely

As is almost always the case, the implementation of my modules did not work perfectly the first time. In fact, testing and debugging of the modules both individually and together took a substantial amount of the total project time. The testing and debugging of the modules took place in several steps. First, each module was debugged for syntax errors. This was usually the simplest step because the Xilinx ISE debugger is very good at identifying typos and other common mistakes. Next, once the modules correctly compiled, I simulated each module using the Xilinx ISE/Modelsim waveform simulators. This step involved testing all of the possible cases for input in order to determine correct functionality. Once each individual module passed this step, I next proceeded to wire them up according to the layout presented in figures 9 and 10. After the modules were wired up and they passed the syntax and compilation steps, the next step would ordinarily have been to test the labkit file in the waveform simulators. Unfortunately however, since the labkit contains so many inputs and outputs, waveform simulations are often not helpful as simulation errors occur frequently, and simulation data becomes complex an intractable. The following step involved generating the program file. Once the modules passed this test, the next step was to program the modules onto the FPGA. Once the FPGA was programmed, the most difficult part of the debugging process began. The process of understanding what was happening on the outputs and why involved many stages of trial and error. Common debugging tools such as displaying a value on the LED's became invaluable in gaining insight into the operation of the system. Below, the testing and debugging of each of the modules is described in greater detail.

## Clock and Wake-up Alarm Modules

As expected, these modules did not work perfectly the first time; however, the debugging of these modules was relatively simple when compared to other modules. This was because all of the behavior of these modules could be easily simulated using the Xilinx ISE/Modelsim waveform simulators. Most of the issues I had with these two modules involved the calculations of the current time, the next time, and the light time. The format chosen for use in the Dorm Room Genie was twenty-four hour time and the format was 4 hexadecimal numbers. While this format was particularly simply to display and program from the keyboard, it made calculating times difficult. This was because each digit was to be treated individually in order to keep it in 4 hexadecimal number format. Thus, the logic to keep time, or subtract 15 minutes for the lighting time was not straightforward and involved careful thought. This also meant that I did not get the formulas right the first time. In order to guarantee their functionality, I had to careful test and simulate all fringe cases for the time in order to ensure that there were no bugs. After several adjustments to the calculations, and some debugging of state transitions, the modules passed the simulation. Since the simulation was very effective in catching the behavior of these modules, further testing during the integration stage was minimal and only involved a few minor adjustments to make my modules compatible with my partner's.

## Lighting Module

As with the clock and wake-up alarm modules, testing and debugging for the module itself was relatively straightforward. Use of the Xilinx ISE debugged and the Modelsim waveform simulators were helpful in determining what bugs were present, and the correction for the bugs found were relatively straightforward. As a result of the extensive simulation testing, the integration of the lighting module itself was relatively simple and only ran into a few minor compatibility issues which were quickly resolved.

The biggest problem I ran into with the lighting module involved the automatic wake-up feature. This is because it involved analog devices which were hooked up to the

FPGA user I/O pins. Both the light value to light voltage and ambient light voltage to ambient light value stages were particularly difficult, and as a result, not all of the functionality desired in the plan was completed.

The lighting module outputs an 8-bit number corresponding to a lighting value which was supposed to be converted into a voltage and displayed on the light bulb. After several hours, a couple fried chips, and two minor burns later, I was able to successfully hook up the output from the FPGA user I/O's to the DAC0831 which converted the binary light value into a current which was then converted into a voltage using a transimpedance amplifier. Once I obtained a working light voltage from the light values, I was pleased with the result, but not for long. This is because upon hooking up the light bulb to the output voltage of the op amp, the voltage at the output would disappear! After another couple hours spent trying to debug this, I finally went downstairs to the 6.002 lab and determined that the resistance of the light bulb was four ohms which effectively made it a short, which explained the loss of voltage at the output of the op amp as well as the fried op amp and my burnt finger. After trying several strategies, including a voltage buffer op amp, I was unable to find a solution for how to place a voltage of five volts across a four ohm resistor. The result was that while I was able to demonstrate the voltage on the oscilloscope, I was unable to demonstrate the lighting of the light bulb as desired.

The second aspect of the automatic lighting mode also presented considerable difficulty and was ultimately left for future work. This part involved obtaining a lighting value from a lighting sensor. For this part of the project, I purchased two sensors from DigiKey which seemed to be exactly what I wanted. I found all of the specs to be what I wanted, except for the minor detail that the sensors were a few millimeters large. My partner and I spent considerable time trying to solder the small sensors so that they could be used in our project, but we were ultimately unsuccessful when in the process of soldering, the contacts were ripped off the sensors. The next idea for sensing the lighting value involved the luminance data which could be calculated using the camera and functionality to be implemented by my partner. My partner will describe this debugging process in greater detail; however the ultimate result was that we were unable to obtain the luminance data from the camera as we had hoped to do. This resulted in our inability to sense the lighting level and thus, the automatic functionality was never fully implemented.

## Display Module

The display module comprised the largest amount of my testing and debugging experience in the project. This is because I had initially tried to display the system state on a 1024 by 768 resolution. This resolution requires the use of the 65 megahertz system clock, which only gives logic 15.4 nanoseconds to generate the pixel. As a result, when I first tested my display module, the displayed strings were full of glitches to the point that the text was virtually unreadable. Thus, a long process began in order to determine the causes of the glitches. Initially, I did not suspect that the timing was the issue. First I looked at the inputs that were used by the logic in order to generate the strings to be displayed. After synchronizing the input signals, and several careful test benches, I effectively eliminated the inputs as the cause. I then created several constant string instances to test if it was the logic itself, and instead found that the glitches persisted. After conferring with the TA's my next strategy was to look and debug the video display code and see if something was interfering with those signals. Finally, after an e-mail to Professor Terman, I learned that the glitch issue was likely due to the timing. The next strategy for solving the glitches was to delay the pixels through a couple of registers in order to give the logic more time to calculate the value of the pixel. This also involved delaying the hsync and vsync signals by the same amount. The pipelining of the pixels did noticeably improve the display when a few strings were display, however as the number of strings displayed increased, the glitches reappeared. After trying to continue with this strategy, a conference with Professors Terman and Chuang provided me with another strategy. This involved changing the display resolution to 640 by 480. This requires the system clock of 50 megahertz which allowed considerably more time for the logic to generate the pixel value. This strategy appeared to solve the problem instantly and once and for all. I was able to get about eight lines of text to display with no glitches at all. However, once additional strings were added to the display, the glitches started appearing again albeit much less than before. The final solution was to discard the unnecessary pixels. This was appropriate since we were displaying the text in black and white anyway, so discarding the pixels would have no effect on the visual presentation. With this final fix, I was finally able to complete the system status display with no visible glitches present.

## Labkit Module

Both my partner David Nedzel and I took part in the final integration in the labkit.v file. I was responsible for wiring up all the modules and getting them to work together, while my partner was responsible for integrating both of our video displays so that they could both be presented using the same VGA signals. Needless to say, integration did not work perfectly the first time. First, there were a couple of different assumptions my partner and I made in our implementations, but since we did the block diagrams and were in conference often, these compatibility issues were quickly resolved. The next part involved testing and bugging the communication between our separately written modules. Thankfully, this part of the integration process was relatively simple.

The largest portion of the testing and debugging process of integration involved identifying incorrectly wired together modules. This is because a typo or different word used in a module declaration is not always identified by verilog as a mistake. Often, verilog is perfectly content by calling a typo in a module declaration a new signal and assigning it the constant value of zero. One particular typo involved several hours of debugging as both my partner and I could not find the source of the problem and spent additional time debugging our modules. Once we confirmed that our modules should indeed be working, I took another look at the module declarations and found the typo. As a result of the extensive planning stage encouraged by the 6.111 course structure, we had few compatibility errors, so once the incorrect wirings were identified, our modules worked together very well.

## Section by David Nedzel

### Signal Processing Module

In designing the signal processing module, I ran into several problems along the way. I first had to expand the PS/2 to ASCII converter to accommodate the new function keys. This involved placing an extra output signal in that code so that the native keyboard code would be displayed on the LEDs. I was subsequently able to identify the relevant codes, and add them to the list of valid inputs. In doing this, I also found that the code given for backslash was incorrect, but a minor tweak (changing the native keyboard code from 8'h5C to 8'h5D) corrected the problem.

Another interesting problem I encountered concerned the interaction between the numeric keypad, which interacts with both the arrow keys, the "insert, delete, home, end, pg up, pg down" block of keys. Initially, instead of using the numeric keypad, I had been using the other ten keys, but problems getting the PS/2 module to recognize their input caused me to modify my design to use the numeric keypad instead. The problem has to do with the way in which those keys send multiple signals, and given additional time, it would be interesting to program those keys to work properly.

### Lock Module

The most challenging aspect of the lock module was ensuring that the timing was exactly correct for the password programming and authentication. After several unsuccessful attempts to read through my Verilog code and spot the errors, I used the Xilinx tools to create a test bench waveform. This allowed to carefully control each of the inputs, and to observe the way in which the output changed with each clock cycle. I was able to view some of the variables internal to the module by temporarily connecting them as outputs.

My lock module went through several design iterations. The original design had 2 RAMs – the first for storing the password, and the second for storing the user's attempt to enter the password. Instead of the two "password partially correct" and "password wrong" locked states, my original design had a "password entry attempt" state and a "compare password state." However, comparing the contents of two RAMs proved more difficult than I originally thought. As I was designing an algorithm to do this, I realized that it wasn't necessary to store the attempted password after all, but rather I could simply compare what was being entered to what was already stored. This proved to be a better design in the end, as the FSM has the same number of states, while it loses the need for a second memory storage element.

### Burglar Alarm Module

The Verilog code for the burglar alarm was more easily debugged than the code for many of the other modules. Before this module was integrated with the clock, I had no way to test its time-storage capability other than a test-bench wave-form – the timing capability appeared to work flawlessly. However, when the clock was hooked up, the time was never stored into the register. After pouring through the Verilog code, I was initially unable to identify a cause – naturally one first assumes that the problem most be either with the clock or the burglar alarm module. In the end, the problem was identified as an integration issue – the wire allegedly connecting the two modules was not only named incorrectly, but was only one bit wide. Needless to say, this project has been a lesson in communication. By ensuing that everyone on the project team discusses how various pieces will come together, such issue can be minimized.

The second design challenge presented by the alarm module was hooking up the alarm buzzer to the lab kit. It was impractical to hook the buzzer up to the logic output for two reasons: first, the logic is not designed to source much current to power the buzzer, and second it required 12 V where the lab kit logic runs at 3.3 V. To solve this problem, I hooked the buzzer in series with a MOSFET across the lab kit's 12 V source. I then planned to use the 3.3 V output to trigger the gate. I'd carefully read the specifications sheet for the 2N7000 MOSFET, and the voltages all seemed to be within the appropriate ranges. Nevertheless, when I hooked up the circuit, the buzzer was sounding even with the gate open. After an intense look back at the lab kit specification sheet, and replacing the MOSFET, I realized I had confused the source and gate pins on the MOSFET. Switching the wires solved the problem.

Video/Motion Module

Of all the modules I worked to implement, this one was by far the most challenging, and in the end I was unsuccessful in getting it to work completely. There were many debugging and design challenges I faced along the way. First and foremost, this module had three different clocks involved. The 27 MHz system clock, the roughly 27 MHz but non-synchronous video clock, and the 50 MHz display clock. This first proved troublesome when I was attempting to index the pixel data in my pixilize module. Following one of the models in the 6.111 sample code, I implemented several latches to synchronize the video data with the 27MHz system clock. The 50 MHz clocked proved less of a challenge, in that by clocking the display RAM at 50 MHz, it can still effectively load asynchronous data at 27 MHz. This was evidenced by my success in piping indexed video data directly to the display RAM, producing an image of reasonable quality, but of course, without the ability to store multiple images in the ZBT RAM.

In designing hardware to load picture data into the ZBT RAM, one had to think carefully about the timing, since there is a two clock-cycle delay between a read request an when the data becomes available. Despite my careful attempts to introduce delay registers, I was unsuccessful in storing and retrieving images from the ZBT. As an attempt to debug the problem, I tried storing constant color bars into the entire ZBT, and us inexplicably ***sometimes*** able to load these color bars back from the ZBT to the display RAM. I still suspect the problem to be an issue of timing, and wish I had had further time to attempt to resolve it.

Finally, despite my work to design it, there was insufficient time to implement the motion detection module. I suspect that while my design may seem good on paper, there would be several challenges implementing it as proposed, largely due to varying light levels, such as the sun passing behind a cloud. However, this very problem suggests a solution to the trouble we had finding luminosity data to drive the automatic light mode. Perhaps with additional time to work, we could have gotten the camera to serve three functions: video capture, motion detection and luminosity sensing.

## Conclusion

The objective of this project was to implement a working dorm room automation system with the goal of providing functionality that would make a student's life easier. This project started with that basic problem. How do we make the average student's life easier? From this problem, we worked together to brainstorm functionality that we would like to have were we to get a Dorm Room Genie for our very own dorm rooms. To solve the problem, we were given the push buttons, sensors, and switches provided by the 6.111 labkit as well as a budget to obtain the additional parts necessary in order to implement the desired functionality.

In order to accomplish our overall objective, we broke the dorm room automation problem into a series of smaller objectives. Each objective involved the implementation of a single function, such as keyboard input. A different module would be created to accomplish each smaller functionality objective. Once all the modules were completed, they would work together to accomplish our overall objective of making the student's life easier. For the dorm room automation system, we identified several functionality objectives and created a module to accomplish each. These objectives included the implementation of the user keyboard interface, a keyless entry system, a burglar alarm system including motion capture, a digital clock and alarm clock, a lighting control system, and a system status display showing the status of the entire system.

The first objective of a user keyboard interface was accomplished by the signal processing module. This required the ability to handle the PS/2 interface and convert the various keystrokes into the ASCII format so that it could be used by the rest of the system. The signal processor module then went further and divided all the keyboard signals into commands that could be used by the other modules in the system.

The second objective of a keyless entry system was accomplished by the lock module. This required the ability to store a programmed password, and then compare it to the password input by the user in order to gain entry into the dorm room. The lock module was used to control access to the dorm room while providing the convenience of a keyless entry system without the downfall of a lockout.

The third objective of a burglar alarm system was accomplished by the burglar alarm system. This system would interpret a correct password input from the lock module, and determine whether a burglary was taking place or not. The burglar alarm module was responsible for providing additional security to the student by sounding the alarm if it suspected an unauthorized entry was taking place. The burglar alarm module would also have the ability to sound the alarm if motion was detected while the user was away. This made the student's life easier by providing them the piece of mind of knowing that their valuables were safe.

The next objective of a digital alarm clock was implemented by the clock module and the wakeup alarm modules. This required the ability to keep time as well as have it programmed upon request. This also required the ability for the user to set the wake-up time, turn on the alarm, turn it off, and snooze the alarm upon request. This furthered the goal of making the student's life easier by covering the student's need to have a separate alarm clock and wake-up alarm. The provided alarm clock could be used in conjunction with another alarm clock in order to ensure the student does not miss an important deadline.

The objective of giving the user the ability to control the lights was accomplished by the lighting module. In order to accomplish this objective, the use of a lighting sensor, an audio/digital converter chip, and a digital/audio converter chip were necessary. The light control module implemented several modes of functionality including the manual mode, automatic mode, sleep mode, and the burglary mode. The manual mode gave the user additional control over the lighting level beyond the standard control of turning the lights on and off. The automatic mode would provide the user with a lighting level that would adjust itself automatically to changes in the level of ambient light. The sleep mode was used in conjunction with the alarm clock to provide the student with another wake-up option. Finally, the burglary mode augmented the functionality of the burglary alarm, but flashing the lights in order to deter a burglary and wake-up the user should an intrusion occur while the user was inside. These lighting module modes combined to provide the student with additional level of control and further simplify their life.

The last objective of providing a visual interface for the user to interact with the Dorm Room Genie was accomplished by the display module. This required the module to interpret the states of all of the other modules, and convert them into messages that would let the user know the current state of the system and programmed values entered. This furthered our objective by letting the user quickly determine the state of the automation system as well as the current time and programmed wake-up time.

The labkit module put all of these smaller modules together. The function of the labkit module was to integrate all the modules together into a single working system. Once the modules were integrated, they collectively accomplished the goal of providing a dorm room automation system that provided functionality designed to improve the lives of the average student.

As with most projects, the Dorm Room Genie did not work correctly the first time. Through a long and methodical process of testing and debugging, we slowly worked our way toward a working product. While we did not accomplish everything we set out to do, we certainly accomplished our larger objective of providing a dorm room automation system, while simultaneously leaving a good start on the functionality we were not able to fully complete.

While we were somewhat disappointed that the Dorm Room Genie did not fully accomplish its planned functionality, we are still very pleased with the overall project experience. The hard work and long hours involved with this project were well worth it and left us with a sense that we really accomplished something as well as learning a little along the way. Overall, we have found the Dorm Room Genie project to be a very rewarding experience.

## Appendix

```
///////////////////////////////////////////////////////////////////////////////
//
// Burglar Module
//
// File:   burglar.v
// Last Updated:   December 13, 2005
// Author: David Nedzel
//
// This module implements a burglar alarm taking input from the user, the lock,
// and multiple sensors to indicate a burglary.
//
//
///////////////////////////////////////////////////////////////////////////////


module burglar(clock, reset, burg_ctrl, door, motion, pw_correct, sys_time,
                      state, burglary, burg_time);

  input clock, reset; // system clock & synchronous reset
  input door; // door sensor '1' = open, '0' = closed
  input motion; // input from motion function of video module, active high signals motion
  input pw_correct; // lock signals that the correct password has been entered, active
high
  input [15:0] sys_time; // system time input from clock module
  input [1:0] burg_ctrl; // user input from signal-processing module
  output [2:0] state; // outputs the current state of the burglar alarm module to the
display
  output [15:0] burg_time; // outpus the time of the last burglary to the display
  output burglary; // outputs to many system modules that there is a burglary in
progress, active high

  wire burglary;

  reg [2:0] state; // 0 = disarmed, 1 = exit, 2 = armed, 3 = at-home armed, 4 = burglary
  reg [2:0] old_state;
  reg [15:0] burg_time;

  assign burglary = (state == 4) ? 1 : 0; // if in burglary state, signal a burglary

  wire arm, instant, disarm;
  assign arm = ((burg_ctrl == 1) && door); // if door is open when arming, enable motion
detection
  assign instant = ((burg_ctrl == 1) && (!door)); // if door is closed when arming,
disable motion detection
  assign disarm = (burg_ctrl == 2); // user-input disarm

  always @(posedge clock) begin
	if (reset) begin
		state <= 0;
		old_state <= 0;
		burg_time <= 16'b0; end
	else case (state)
		0: begin // disarmed
			old_state <= 0;
			state <= arm ? 1 : // if user signals arm while door is open,
transition to exit
					instant ? 3 : 0; end // if user signals arm while door
is closed, arm without motion detection enabled
		1: begin // exit
			old_state <= 1;
			state <= (disarm || pw_correct) ? 0 : // disarm if user signals
disarm, or if correct password is entered
					(!door) ? 2 : 1; end // arm with motion once user
closes the door
		2: begin // armed
			old_state <= 2;
			state <= (disarm || pw_correct) ? 0 : // disarm if user signals
disarm, or if correct password is entered
```

```
                                  (door || motion) ? 4 : 2; end // transition to burglary
if the door is opened, or motion is sensed
               3: begin // at-home armed (motion detection disabled)
                      old_state <= 3;
                      state <= (disarm || pw_correct) ? 0 : // disarm if user signals
disarm, or if correct password is entered
                                  door ? 4 : 3; end // transition to burglary if the door
is opened
               4: begin // burglary
                      old_state <= 4;
                      state <= pw_correct ? 0 : 4; // disarm once correct password is
entered
                      burg_time <= (old_state != 4) ? sys_time : burg_time; end // when
burglary begins, store the system time
               default: state <= 0;
        endcase
  end

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Video Capture Module
//
// File:   capture.v
// Last Updated:   December 13, 2005
// Author: David Nedzel
//
//
// This simplified video module takes as input video signals and outputs a
// still image to the display
//
//
////////////////////////////////////////////////////////////////////////////////


module video(reset, clock_27mhz, clock_50mhz, burglary, tv_in_line_clock1, tv_in_ycrcb,
                  vga_hcount, vga_vcount, vga_pixel);

   input reset, clock_27mhz; // system clock & synchronous reset
   input clock_50mhz; // clock for driving the display
   input burglary; // indicates a burlary, causing module to record
   input tv_in_line_clock1; // video in clock
   input [9:0] tv_in_ycrcb; // video in lummenance data
   input [9:0] vga_hcount, vga_vcount; // pixel counters from vga signals generator
   output [7:0] vga_pixel; // pixel data


   // VGA display (640x480 resolution)
   //wire vga_hsync, vga_vsync, vga_pix_clk, vga_blank;
   wire [7:0] dram_data_out, vga_pixel;
   //wire [9:0] vga_hcount, vga_vcount;
   assign vga_pixel = dram_data_out;

   // Decode Input Video
   wire [29:0] ycrcb;  // video data (luminance, chrominance)
   wire f, v, h, dv;   // field sync, vertical sync, horizontal sync, data valid
   ntsc_decode decode(tv_in_line_clock1, reset, tv_in_ycrcb, ycrcb, f, v, h, dv);

   // Convert decoded video data to pixilized format
   wire [7:0] pixel;
   wire [9:0] col, row;
   wire valid;
   ntsc_to_pixel pixels(clock_27mhz, tv_in_line_clock1, {f, v, h}, dv, ycrcb[29:22], col,
row, pixel, valid);

   // Store and retrieve image from display RAM
   wire [16:0] address;
   assign address = burglary ? (col + (row * 320)) : ((vga_hcount/2) + ((vga_vcount/2) *
320));
```

```
   // display ram implementation
   ram76800x8 dram(.clk(clock_50mhz), .we(valid && burglary), .addr(address),
                              .din(pixel), .dout(dram_data_out));

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// Clock Module
//
// File:   clock.v
// Last Updated:  December 13, 2005
// Author: Aaron Stonely
//
// This module implements the behavior of a digital alarm clock.
//
///////////////////////////////////////////////////////////////////////////////

module clock(clk,reset,time_ctrl,numbers,burglary,time_selector,
             done,time_state,thetime,one_min_enable);

input clk, reset, time_ctrl, burglary, time_selector;
input [3:0] numbers;
output done;
output [2:0] time_state;
output [15:0] thetime;
output one_min_enable;

wire [15:0] thetime;
wire one_min_enable, one_hz_enable;

reg [15:0] prog_time=0;
reg [2:0] time_state=0;
reg reprogram=0, done=0;

// Below, both a one second enable and one_hz_enable are initialized
// so that the user can chooses to advance time every minute or every
// which is done through the time_selector switch and time_enable below.
one_min_enable mins(clk,reset,one_min_enable);
one_hz_enable secs(clk,reset,one_hz_enable);

wire time_enable = time_selector ? one_hz_enable : one_min_enable;

// Below the clock register is initialized to be used to store the time.
clock_reg clkreg(clk,reset,reprogram,time_enable,
                 prog_time,thetime);

// Below is a statement used to determine if the keyboard has requested
// to set the time, or if the keyboard is sending a valid number to reprogram
// the system time.
wire set_time = ((time_ctrl |
                       ((time_state == S_001) && (numbers < 4'h3)) ||
                       ((time_state == S_010) && (numbers < 4'hA) && (prog_time[15:12] <
2)) ||
                       ((time_state == S_010) && (numbers < 4'h4) && (prog_time[15:12] ==
2)) ||
                       ((time_state == S_011) && (numbers < 4'h6)) ||
                       ((time_state == S_100) && (numbers < 4'hA))) &&
                       !burglary);

// Below are the states used by the Clock FSM

parameter S_000 = 0; // Reset/Normal operating state
parameter S_001 = 1; // Waiting for first number
parameter S_010 = 2; // Waiting for second number
parameter S_011 = 3; // Waiting for third number
parameter S_100 = 4; // Waiting for fourth number

always @ (posedge clk)
```

```
begin
    if(reset | burglary)
    begin
        reprogram <= 0;
        time_state <= 0;
        prog_time <= 0;
    end
    else
    case(time_state)
        S_000:        // Reset/Normal operating state
                        // Only transitions to S_001 upon a time
                        // reprogram request from the user.
           if(set_time)
               time_state <= S_001;
           else
           begin
               reprogram <= 0;
               prog_time <= 0;
                      done <= 0;
           end
                 // States S_001 through S_100 transition upon
                 // receipt of a valid number from the keyboard.
                 // S_100 transitions back to the normal operating
                 // state since the program sequence is complete.

        S_001:        // Waiting for first number
           if(set_time)
           begin
               time_state <= S_010;
               prog_time <= prog_time + {numbers,12'h0};
           end
        S_010:        // Waiting for second number
           if(set_time)
           begin
              time_state <= S_011;
              prog_time <= prog_time + {4'h0,numbers,8'h00};
          end
        S_011:        // Waiting for third number
           if(set_time)
           begin
               time_state <= S_100;
                      prog_time <= prog_time + {8'h00,numbers,4'h0};
           end
        S_100:        // Waiting for fourth number
           if(set_time)
           begin
               time_state <= S_000;
                      prog_time <= prog_time + {12'h000,numbers};
               reprogram <= 1;
                      done <= 1;
           end
        default:
           begin
                reprogram <= 0;
                time_state <= 0;
                prog_time <= 0;
                       done <= 0;
            end
    endcase
end

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// One_hz_enable Module
//
// File:   Clock.v
// Last Updated:   December 13, 2005
// Author: Aaron Stonely
```

```
//
// Has a counter that counts to 1.62 billion then sets the one_min_enable high
// and rolls over. This makes one_min_enable be high for 1 clock cycle every
// 1 minute.
//
// Assumes 27Mhz clock
//
////////////////////////////////////////////////////////////////////////////////

module one_min_enable(clk,reset,one_min_enable);

input clk, reset;
output one_min_enable;

// In order to count to 1.62 billion, 31 registers for count are needed.
reg [30:0] count=0;
reg one_min_enable=0;

always @ (posedge clk)
begin
    if(reset)
    begin
        one_min_enable <= 0;
        count <= 0;
    end
    else
    begin
        if(count == 1619999999)
        begin
            count <= 0;
            one_min_enable <= 1;
        end
        else
        begin
            one_min_enable <= 0;
            count <= count + 1;
        end
    end
end
endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Clock Register Module
//
// File:   Clock.v
// Last Updated:   December 13, 2005
// Author: Aaron Stonely
//
// This module stores the current time, and reprograms the current time upon
// request from the clock module.
//
// This module also increments the time every one minute or one second as
//      selected by the user.
//
////////////////////////////////////////////////////////////////////////////////

module clock_reg(clk,reset,reprogram,time_enable,
                 prog_time,thetime);

input clk, reset, reprogram, time_enable;
input [15:0] prog_time;
output [15:0] thetime;

reg [15:0] nextime=0, thetime=0;


// Below are several statements used to calculate the current time
// as 4 hex numbers. The format is 24-hour time.
wire [3:0] min1 = ((nextime << 12) >> 12);
wire [3:0] min2 = ((nextime << 8) >> 12);
wire [3:0] hour1 = ((nextime << 4) >> 12);
wire [3:0] hour2 = (nextime >> 12);
wire [3:0] next_hour2 = (hour2 + 1);
wire [3:0] next_hour1 = (hour1 + 1);
wire [3:0] next_min2 = (min2 + 1);

// Below, a reprogram, and time-keeping behavior are implemented
// The time is changed to the user inputted program time upon request,
// and the time is incremented once every time_enable.
always @ (posedge clk)
begin
thetime <= reset ? 0 :
           reprogram ? prog_time :
           time_enable ? nextime :
           thetime;
nextime <= reset ? 1 :
           reprogram ? prog_time :
            time_enable ?
                                      (min1 == 9) ?
                                      (min2 == 5) ?
                                      (hour2 == 2) ?
                                      (hour1 == 3) ? 0 :
                                      {hour2,next_hour1,8'h00} :
                                      (hour1 == 9) ?
                                      {next_hour2,12'h000} :
                                      {hour2,next_hour1,8'h00} :
                                      {hour2,hour1,next_min2,4'h0} :
                                      (nextime + 1) :
                                      nextime;
end
endmodule


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

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

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

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

   input vclock;      // 50mHz clock
   input [9:0] hcount; // horizontal index of current pixel (0..1023)5;
   input [9:0]        vcount; // vertical index of current pixel (0..767)
   output [2:0] pixel; // char display's pixel
   input [NCHAR*8-1:0] cstring;      // character string to display
   input [10:0] cx;
   input [9:0]        cy;

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

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

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

   reg dispflag;
   always @ (posedge vclock) begin
       xchar <= char;
       dispflag <= ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*8)
                   & (vcount < cy + 12));
  end

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

   // generate character pixel if we're in the right h,v area
   wire [2:0] cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0;
   wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Debounce Module
//
// File:   debounce.v
// Last Updated:   December 13, 2005
//
// ****** NOTE ******
// This module contains code provided by the 6.111 course staff.
//
// This particular implementation has a modified time parameter designed to
// handle the bounce of the magnetic switch.
//
////////////////////////////////////////////////////////////////////////////////

// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce (reset, clock, noisy, clean);
   parameter DELAY = 2700000;   // .1 sec with a 27Mhz clock
   input reset, clock, noisy;
   output clean;

   reg [21:0] count;
   reg new, clean;

   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


///////////////////////////////////////////////////////////////////////////////
//
// DelayN Module
//
//
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
//
///////////////////////////////////////////////////////////////////////////////

module delayN(clk,in,out);

input clk,in;
output out;

parameter NDELAY = 2;

reg[NDELAY-1:0] shiftreg;

wire out = shiftreg[NDELAY-1];

always @ (posedge clk)
       shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// Delaycount Module
//
//
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
//
// ***** Note ******
// This file contains code written by Professor I. Chuang. It is a modified
// version of the delayN module shown above. It has been changed in order to
// implement a two clock-cycle delay of the hcount and vcount signals which
// are 10 bits wide.
//
///////////////////////////////////////////////////////////////////////////////

module delaycount(clk,in,out);
   input clk;
   input [9:0] in;
   output [9:0] out;

   reg [19:0] shiftreg;
   wire out = shiftreg[19:10];

   always @(posedge clk)
     shiftreg <= {shiftreg[9:0], in};

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// System Display Module
//
// File:   display.v
// Last Updated:   December 13, 2005
// Author: Aaron Stonely
//
// This module is used to create the display that will show the status of the
// entire system on a VGA screen of resolution 640 x 480.
//
// ****** NOTE ******
//     This module uses the character display module written by Professors
// I. Chaung, and C. Terman.
//
///////////////////////////////////////////////////////////////////////////////

module system_display(clk,reset,thetime,time_state,wake_time,wake_state,state_lock,

        state_sig,state_burg,burg_time,light_state,light_value,pixor,hcount,vcount);

    input [15:0] thetime,burg_time,wake_time;
    input [9:0] hcount,vcount;
    input [7:0] light_value;
    input [3:0] wake_state;
    input [2:0] time_state,state_sig,state_burg,state_lock,light_state;
    input clk,reset;
    output [2:0] pixor;

// The system display is comprised of 12 instances of the character display module.
// Each instance outputs a pixel associated with that instance. The pixel is either
// all 1's or all 0's corresponding to a black or white pixel. Whether a pixel is
// black or white is determined by hcount and vcount, the position of the instance,
// and the font_rom.

// Note: Every instance that involves a signal from the controller is synchronized
// to the 50mhz clock since the signals themselves were generated by modules
// operating at 27mhz.
//
// Also, the pixel from every instance is delayed in order to allow more time for
// the logic to generate the pixel which will cut down on visible glitches.

// System Time Display
// Format: "System Time: (HH:MM)"

   wire [15:0] sync_time;
   synchronize sync_t[15:0](clk,thetime,sync_time);

// Below the time is converted to ASCII
   wire [7:0] min1_disp = (sync_time[3:0] + 48);
   wire [7:0] min2_disp = (sync_time[7:4] + 48);
   wire [7:0] hour1_disp = (sync_time[11:8] + 48);
   wire [7:0] hour2_disp = (sync_time[15:12] + 48);

   wire [143:0] timestring_a = {"System Time:
",hour2_disp,hour1_disp,":",min2_disp,min1_disp};
   wire [2:0]  timepixel_a;
   char_string_display timecd_a(clk,hcount,vcount,
                     timepixel_a,timestring_a,11'd100,10'd260);
   defparam timecd_a.NCHAR = 18;
   defparam timecd_a.NCHAR_BITS = 5;

   reg [2:0] timepixdelay_a;
   always @ (posedge clk) timepixdelay_a <= timepixel_a;

// System Time State Display

   wire [2:0] time_state_sync;
   synchronize sync_ts[2:0](clk,time_state,time_state_sync);

   wire [287:0] timestring_b = (time_state_sync == 0) ?
                                                    {"System clock is in operational
mode."} :
                                                           {"Please program the current
time.    "};
   wire [2:0] timepixel_b;
   char_string_display timecd_b(clk,hcount,vcount,
                     timepixel_b,timestring_b,11'd100,10'd280);
   defparam timecd_b.NCHAR = 36;
   defparam timecd_b.NCHAR_BITS = 6;

   reg [2:0] timepixdelay_b;
   always @ (posedge clk) timepixdelay_b <= timepixel_b;

// Wake-up Time display
// Format: "Wake-up time: (HH:MM)"

   wire [15:0] sync_wake_time;
   synchronize sync_w[15:0](clk,wake_time,sync_wake_time);

// Below the time is converted to ASCII
   wire [7:0] wake_min1_disp = (sync_wake_time[3:0] + 48);
   wire [7:0] wake_min2_disp = (sync_wake_time[7:4] + 48);
   wire [7:0] wake_hour1_disp = (sync_wake_time[11:8] + 48);
   wire [7:0] wake_hour2_disp = (sync_wake_time[15:12] + 48);

   wire [151:0] wakestring_a = {"Wake-up Time: ",wake_hour2_disp,wake_hour1_disp,":",
                                             wake_min2_disp,wake_min1_disp};
   wire [2:0] wakepixel_a;
   char_string_display wakecd_a(clk,hcount,vcount,wakepixel_a,
                                    wakestring_a,11'd100,10'd300);
   defparam wakecd_a.NCHAR = 19;
   defparam wakecd_a.NCHAR_BITS = 5;

   reg [2:0] wakepixdelay_a;
   always @ (posedge clk) wakepixdelay_a <= wakepixel_a;

// Wake-up State Display

   wire [3:0] sync_wake_state;
   synchronize sync_ws[3:0](clk,wake_state,sync_wake_state);

   wire [255:0] wakestring_b = (sync_wake_state == 0) ?
                                                    {"Wake-up Alarm is off.           "}
:
                                               ((sync_wake_state > 0) && (sync_wake_state
< 5)) ?
                                                    {"Please program the wake-up time."}
:
                                               (sync_wake_state == 5) ?
```

```
                                                      {"The wake-up alarm is on.        "}
:
                                                 (sync_wake_state == 6) ?
                                                      {"The wake-up sequence has begun. "}
:
                                                 (sync_wake_state == 7) ?
                                                      {"Time to wake up!!               "}
:
                                                 (sync_wake_state == 8) ?
                                                      {"Alarm is in snooze mode.        "}
:
                                                      {"System error! Contact help.     "};
   wire [2:0] wakepixel_b;
   char_string_display wakecd_b(clk,hcount,vcount,wakepixel_b,
                                    wakestring_b,11'd100,10'd320);
   defparam wakecd_b.NCHAR = 32;
   defparam wakecd_b.NCHAR_BITS = 6;

   reg [2:0] wakepixdelay_b;
   always @ (posedge clk) wakepixdelay_b <= wakepixel_b;

// System State Display

   wire [2:0] sync_state_sig;
   synchronize sync_ss[2:0](clk,state_sig,sync_state_sig);

   wire [375:0] sigstring = (sync_state_sig == 0) ?
                                                      {"System is in normal operating mode.
"} :
                                          (sync_state_sig == 1) ?
                                                      {"The lock module is waiting for
input.          "} :
                                          (sync_state_sig == 2) ?
                                                      {"The clock module is waiting for
input.         "} :
                                          (sync_state_sig == 3) ?
                                                      {"The wake-up module is waiting for
input.       "} :
                                          (sync_state_sig == 4) ?
                                                      {"Burglary in progress! Enter
password to disarm."} :
                                          {"Fatal error has occured!
"};
   wire [2:0] sig_pixel;
   char_string_display sigcd(clk,hcount,vcount,sig_pixel,
                                    sigstring,11'd100,10'd60);
   defparam sigcd.NCHAR = 47;
   defparam sigcd.NCHAR_BITS = 6;

   reg [2:0] sigpixdelay;
   always @ (posedge clk) sigpixdelay <= sig_pixel;

// Lock State Display

   wire [2:0] sync_state_lock;
   synchronize sync_sl[2:0](clk,state_lock,sync_state_lock);

   wire [247:0] lockstring = (sync_state_lock < 2) ?
                                             {"The door is locked.            "} :
                                             (sync_state_lock == 2) ?
                                             {"Please enter.                  "} :
                                             (sync_state_lock == 3) ?
                                             {"The door is unlocked.          "} :
                                             (sync_state_lock == 4) ?
                                             {"Keyless entry is inactive.     "} :
                                             (sync_state_lock == 5) ?
                                             {"Please enter your new password."} :
                                             {"Fatal error has occured!       "};
   wire [2:0] lock_pixel;
   char_string_display lockcd(clk,hcount,vcount,lock_pixel,
                                    lockstring,11'd100,10'd100);
   defparam lockcd.NCHAR = 31;
   defparam lockcd.NCHAR_BITS = 5;

   reg [2:0] lockpixdelay;
   always @ (posedge clk) lockpixdelay <= lock_pixel;

// Burglary State Display

   wire [2:0] sync_state_burg;
   synchronize sync_sb[2:0](clk,state_burg,sync_state_burg);

   wire [407:0] burgstring_a = (sync_state_burg == 0) ?
                                                  {"Burglar alarm is disarmed.
"} :
                                             (sync_state_burg == 1) ?
                                                  {"Burglar alarm is in standby: Please
exit now.      "} :
                                             (sync_state_burg == 2) ?
                                                  {"Burglar alarm is armed.
"} :
                                             (sync_state_burg == 3) ?
                                                  {"Burglar alarm is armed. (Motion
detection disabled)"} :
                                             (sync_state_burg == 4) ?
                                                  {"Burglary in progress!!
"} :
                                                  {"Fatal error has occured!
"};

   wire [2:0] burg_pixel_a;
   char_string_display burgcd_a(clk,hcount,vcount,burg_pixel_a,
                                    burgstring_a,11'd100,10'd160);
   defparam burgcd_a.NCHAR = 51;
   defparam burgcd_a.NCHAR_BITS = 6;

   reg [2:0] burgpixdelay_a;
   always @ (posedge clk) burgpixdelay_a <= burg_pixel_a;


// Burglary Time Display

// Used to determine if a burglary has in fact occured.
   reg burg_check=0;
   always @ (posedge clk)
   begin
   if(reset) burg_check <= 0;
   else
   burg_check <= (sync_state_burg == 4) ? 1 : burg_check;
   end

   wire [15:0] sync_burg_time;
   synchronize sync_bt[15:0](clk,burg_time,sync_burg_time);

// Below the time is converted to ASCII
   wire [7:0] burg_min1_disp = (sync_burg_time[3:0] + 48);
   wire [7:0] burg_min2_disp = (sync_burg_time[7:4] + 48);
   wire [7:0] burg_hour1_disp = (sync_burg_time[11:8] + 48);
   wire [7:0] burg_hour2_disp = (sync_burg_time[15:12] + 48);

   wire [255:0] burgstring_b = (burg_check == 0) ?
                                  {"A burglary has not occured.     "} :
                                  {"A burglary has occured at: ",
                                  burg_hour2_disp,burg_hour1_disp,":",
                                  burg_min2_disp,burg_min1_disp};

   wire [2:0] burgpixel_b;
   char_string_display burgcd_b(clk,hcount,vcount,burgpixel_b,
                                    burgstring_b,11'd100,10'd180);
   defparam burgcd_b.NCHAR = 32;
   defparam burgcd_b.NCHAR_BITS = 6;
   reg [2:0] burgpixdelay_b;
   always @ (posedge clk) burgpixdelay_b <= burgpixel_b;

// Light State Display

   wire [2:0] sync_light_state;
   synchronize sync_ls[2:0](clk,light_state,sync_light_state);

   wire [247:0] lightstring = (sync_light_state == 0) ?
                                          {"Lighting is in manual mode.    "} :
                                           (sync_light_state == 1) ?
                                          {"Lighting is in automatic mode. "} :
                                           (sync_light_state == 2) ?
                                          {"Lighting is in sleep mode.     "} :
                                           (sync_light_state == 3) ?
                                          {"Intruder alert! Intruder Alert!"} :
                                          {"Fatal error occured!           "};

   wire [2:0] light_pixel;
   char_string_display lightcd(clk,hcount,vcount,light_pixel,
                                    lightstring,11'd100,10'd400);
   defparam lightcd.NCHAR = 31;
   defparam lightcd.NCHAR_BITS = 5;

   reg [2:0] lightpixdelay;
   always @ (posedge clk) lightpixdelay <= light_pixel;

// Light Value Display

   wire [7:0] sync_light_val;
   synchronize sync_lv[7:0](clk,light_value,sync_light_val);

// Below the 8-bit light value is converted to ASCII
   wire [7:0] light_val1_disp = (sync_light_val[0] + 48);
   wire [7:0] light_val2_disp = (sync_light_val[1] + 48);
   wire [7:0] light_val3_disp = (sync_light_val[2] + 48);
   wire [7:0] light_val4_disp = (sync_light_val[3] + 48);
   wire [7:0] light_val5_disp = (sync_light_val[4] + 48);
   wire [7:0] light_val6_disp = (sync_light_val[5] + 48);
   wire [7:0] light_val7_disp = (sync_light_val[6] + 48);
   wire [7:0] light_val8_disp = (sync_light_val[7] + 48);

   wire [167:0] lightvalstring = {"Light value: ",
                                                 light_val8_disp,light_val7_disp,
                                                 light_val6_disp,light_val5_disp,
                                                 light_val4_disp,light_val3_disp,
                                                 light_val2_disp,light_val1_disp};

   wire [2:0] lightval_pixel;
   char_string_display lightvalcd(clk,hcount,vcount,lightval_pixel,
                                          lightvalstring,11'd100,10'd420);
   defparam lightvalcd.NCHAR = 21;
   defparam lightvalcd.NCHAR_BITS = 5;

   reg [2:0] lightvalpixdelay;
   always @ (posedge clk) lightvalpixdelay <= lightval_pixel;

// Light Level Display

// Below, the light values are converted into levels which
// are accessible to manual adjustment.

   wire [159:0] lightlevelstring = (sync_light_val > 250) ?
                                                        {"Light Level: 7 (Max)"} :
                                                  (sync_light_val > 220) ?
                                                        {"Light Level: 6      "} :
                                                  (sync_light_val > 185) ?
                                                        {"Light Level: 5      "} :
                                                  (sync_light_val > 155) ?
                                                        {"Light Level: 4      "} :
                                                  (sync_light_val > 120) ?
```

```
                                                {"Light Level: 3      "} :
                                          (sync_light_val > 90) ?
                                                {"Light Level: 2      "} :
                                          (sync_light_val > 55) ?
                                                {"Light Level: 1      "} :
                                                {"Lights are off.     "};
   wire [2:0] lightlevel_pixel;
   char_string_display lightlevelcd(clk,hcount,vcount,lightlevel_pixel,
                                         lightlevelstring,11'd100,10'd444);
   defparam lightlevelcd.NCHAR = 20;
   defparam lightlevelcd.NCHAR_BITS = 5;

   reg [2:0] lightlevelpixdelay;
   always @ (posedge clk) lightlevelpixdelay <= lightlevel_pixel;


// Title Display
// "The Dorm Room Genie" !

   wire [151:0] titlestring = {"The Dorm Room Genie"};
   wire [2:0] title_pixel;
   char_string_display titlecd(clk,hcount,vcount,title_pixel,
                                  titlestring,11'd248,10'd20);
   defparam titlecd.NCHAR = 19;
   defparam titlecd.NCHAR_BITS = 5;

   reg [2:0] titlepixdelay;
   always @ (posedge clk) titlepixdelay <= title_pixel;

// Below all the display pixels are OR'd together in order to create
// a single output pixel.
   wire [2:0] pixor_a = (timepixdelay_a | timepixdelay_b | wakepixdelay_a |
                                 wakepixdelay_b | sigpixdelay | lockpixdelay |
burgpixdelay_a |
                                 burgpixdelay_b | lightpixdelay | lightvalpixdelay |
                                 lightlevelpixdelay | titlepixdelay);

// Before being output, the pixel is reduced by the OR operator, and is
// repeated for all the pixel values. This ensures that only a pixel value of
// 111 or 000 (white or black) will be output to this display. This further
// helps to cut down on visible glitches.

   wire [2:0] pixor = {|pixor_a,|pixor_a,|pixor_a};

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// Divider Module
//
// File:   divider.v
// Last Updated:   December 13, 2005
// Author: David Nedzel
//
// This module takes as input a 27MHz clock, and produces a 2 hertz clock
// signal.
//
///////////////////////////////////////////////////////////////////////////////

module divider (clock, reset, hz_enable);

   input clock, reset;
   output hz_enable;

   reg [31:0] count;

   always @(posedge clock)
   count <= (reset || hz_enable) ? 1 : count+1;

assign hz_enable = (count == 13500000) ? 1 : 0;
```

```
endmodule

/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2004 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file font_rom.v when simulating
// the core, font_rom. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module font_rom(
       addr,
       clk,
       dout);


input [10 : 0] addr;
input clk;
output [7 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
               11,     // c_addr_width
               "0",    // c_default_data
               1536,   // c_depth
               0,      // c_enable_rlocs
               0,      // c_has_default_data
               0,      // c_has_din
               0,      // c_has_en
               0,      // c_has_limit_data_pitch
               0,      // c_has_nd
               0,      // c_has_rdy
               0,      // c_has_rfd
               0,      // c_has_sinit
               0,      // c_has_we
               18,     // c_limit_data_pitch
               "font_rom.mif",        // c_mem_init_file
               0,      // c_pipe_stages
               0,      // c_reg_inputs
               "0",    // c_sinit_value
               8,      // c_width
               0,      // c_write_mode
               "0",    // c_ybottom_addr
               1,      // c_yclk_is_rising
               1,      // c_yen_is_high
               "hierarchy1",  // c_yhierarchy
               0,      // c_ymake_bmm
               "16kx1",       // c_yprimitive_type
               1,      // c_ysinit_is_high
               "1024", // c_ytop_addr
               0,      // c_yuse_single_primitive
               1,      // c_ywe_is_high
               1)      // c_yydisable_warnings
       inst (
               .ADDR(addr),
               .CLK(clk),
               .DOUT(dout),
               .DIN(),
               .EN(),
               .ND(),
               .RFD(),
               .RDY(),
               .SINIT(),
               .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of font_rom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of font_rom is "black_box"

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
// ***** NOTE *****
// Contains by David Nedzel and Aaron Stonely as noted below
// ****************
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
```

```
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

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

               clock_feedback_out, clock_feedback_in,

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

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

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

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

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

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

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

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

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

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

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

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

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

    input  clock_feedback_in;
    output clock_feedback_out;

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

    output rs232_txd, rs232_rts;
    input  rs232_rxd, rs232_cts;

    input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

    input  clock_27mhz, clock1, clock2;

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

    input  button0, button1, button2, button3, button_enter, button_right,
          button_left, button_down, button_up;
```

```
input  [7:0] switch;
output [7:0] led;

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

inout [43:0] daughtercard;

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

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

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

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

// VGA Output
// assign vga_out_red = 10'h0;
// assign vga_out_green = 10'h0;
// assign vga_out_blue = 10'h0;
// assign vga_out_sync_b = 1'b1;
// assign vga_out_blank_b = 1'b1;
// assign vga_out_pixel_clock = 1'b0;
// assign vga_out_hsync = 1'b0;
// assign vga_out_vsync = 1'b0;

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

// Video Input
//assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
//assign tv_in_reset_b = 1'b0;
assign tv_in_clock = clock_27mhz;
//assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

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

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

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

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

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

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

// Daughtercard Connectors
assign daughtercard = 44'hZ;

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

// Logic Analyzer
assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

// Below continues code provided by the 6.111 Staff
// This code is used by our video modules

// use FPGA's digital clock manager to produce a 50 Mhz clock from 27 Mhz
// actual frequency: 49.85 MHz
```

```
wire clock_50mhz_unbuf,clock_50MHz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_50mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 13
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_50MHz),.I(clock_50mhz_unbuf));

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_50MHz), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce2 db5(power_on_reset, clock_50MHz, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;

// 640x480 VGA display
wire [7:0] pixel;
wire       blank;
wire       pix_clk;
wire [9:0] hcount, vcount;

vga_sync vga1(clock_50MHz,hsync,vsync,hcount,vcount,pix_clk,blank);

// adv7185 initialization module
adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                   .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                   .tv_in_i2c_clock(tv_in_i2c_clock),
                   .tv_in_i2c_data(tv_in_i2c_data));

// ********************************************************
// The following contains code written by:
// David Nedzel and Aaron Stonely.
// ********************************************************


// The wires for connecting all the modules together are
// declared here.
wire [15:0] sys_time,burg_time,thetime,light_time,wake_time;
wire [7:0] ascii_data,light_value,sensor,char_data,vga_pixel;
wire [3:0] num_data,wake_state;
wire [2:0] state_sig,state_burg,state_lock,light_state,pixor;
wire [2:0] lock_ctrl,lght_ctrl,wake_ctrl,time_state;
wire [1:0] burg_ctrl,pic_ctrl,vid_test;
wire burglary,door,ascii_ready,time_ctrl,pw_correct,motion;
wire wake_buzzer,alarm,take_pic,disp_pic,unlock,lock_done;
wire time_selector,wake_done,one_hz_enable,write,read;
wire one_min_enable,hsdelay,vsdelay;

// The buzzer output is assigned so that it may be used as a
// wakeup alarm buzzer and a burglar alarm. This signal is
// then sent out through the I/O's and wired to a buzzer.
assign buzzer = (burglary | wake_buzzer);

// User I/Os
// Used for the buzzer, the door sensor, and the light
// digital audio converter.
assign user1 = {31'hZ, buzzer};
assign user2 = 32'hZ;
//assign user3 = 32'hZ;
assign user4 = {24'hZ, light_value};

// Below, the switches and buttons used during operation are
// debounced using a module provided by the 6.111 course staff.
debounce door_sensor(reset, clock_27mhz, user3[0], door);
debounce demo_selector1(reset, clock_27mhz, switch[7], time_selector);
debounce demo_selector2(reset, clock_27mhz, switch[5], take_pic);
debounce demo_selector3(reset, clock_27mhz, switch[3], disp_pic);
```

```
   debounce demo_selector4(reset, clock_27mhz, switch[1], vid_test[1]);
   debounce demo_selector5(reset, clock_27mhz, switch[0], vid_test[0]);
   debounce demo_selector6(reset, clock_27mhz, ~button1, motion);

   // Below the LED's are assigned to indicate the status of the door and
   // the presence of motion.
   assign led[6:0] = unlock ? 6'b111111 : 6'b000000;
   assign led[8:7] = motion ? 2'b11 : 2'b00;

   // Below, all the modules are declared and wired up together.

   ps2_ascii_input keyboard(clock_27mhz, reset, keyboard_clock, keyboard_data,
ascii_data, ascii_ready);

   signal_processor my_sig(clock_27mhz, reset, ascii_data, ascii_ready, lock_done,
time_done, wake_done, burglary,
                                       state_sig, char_data, num_data, lock_ctrl,
burg_ctrl, lght_ctrl, time_ctrl, wake_ctrl, pic_ctrl);

   burglar my_burglar(clock_27mhz, reset, burg_ctrl, door, motion, pw_correct, thetime,
                state_burg, burglary, burg_time);

   lock my_lock(clock_27mhz, reset, char_data, lock_ctrl, door, state_lock, unlock,
pw_correct, lock_done);

   one_hz_enable one_second(clock_27mhz,reset,one_hz_enable);

   adc_sampler adc_signals(clock_27mhz,reset,one_hz_enable,write,read);

   clock theclock(clock_27mhz,reset,time_ctrl,num_data,burglary,time_selector,
                         time_done,time_state,thetime,one_min_enable);

   lighting
thelights(clock_27mhz,reset,lght_ctrl,alarm,burglary,sensor,light_value,light_state,time_
selector);

   wakeup
wake_module(clock_27mhz,reset,wake_ctrl,num_data,thetime,one_min_enable,burglary,time_sel
ector,

wake_buzzer,wake_done,alarm,wake_state,wake_time,light_time);

   system_display
sys_disp(clock_50MHz,reset,thetime,time_state,wake_time,wake_state,state_lock,

        state_sig,state_burg,burg_time,light_state,light_value,pixor,hcount,vcount);

   video capture(reset, clock_27mhz, clock_50MHz, take_pic, tv_in_line_clock1,
tv_in_ycrcb[19:10],
                         hcount, vcount, vga_pixel);

   // **The following contains code written by the 6.111 course staff.**
   // It has been altered as needed.

   //  vid_test[1:0] selects which video generator to use:
   //  00: System Status Display
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars

   reg [2:0] rgb;
   reg b,hs,vs;
   always @(posedge clock_50MHz) begin
      hs <= hsync;
      vs <= vsync;
      b <= blank;
      if (vid_test[1:0] == 2'b01) begin
         // 1 pixel outline of visible area (white)
         rgb <= (hcount==0 | hcount==639 | vcount==0 | vcount==479) ? 7 : 0;
      end else if (vid_test[1:0] == 2'b10) begin
         // color bars
         rgb <= hcount[8:6];
      end else begin
      // System Status Display
        rgb <= pixor;
      end
   end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it pix_clk from the providedvga_sync module.

   // The vga out rgb values have been assigned the same bit in order
   // to get a black and white text output when the system display is
   // visible.
   assign vga_out_red = disp_pic ? vga_pixel :{8{rgb[0]}};
   assign vga_out_green = disp_pic ? vga_pixel : {8{rgb[0]}};
   assign vga_out_blue = disp_pic ? vga_pixel :{8{rgb[0]}};
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = disp_pic ? ~blank : ~b;
   assign vga_out_pixel_clock = pix_clk;

   // Delay for horizontal and vertical sync
   // This is needed due to pipelining of pixel data present in the
   // System Display and Character Display modules
   delayN delay_hs(clock_50MHz,hs,hsdelay);
   delayN delay_vs(clock_50MHz,vs,vsdelay);
   assign vga_out_hsync = disp_pic ? hsync : hsdelay;
   assign vga_out_vsync = disp_pic ? vsync : vsdelay;

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//      ****** NOTE *******
//
// This module was provided by the 6.111 course staff in order to be used in
// conjunction with the provided video modules.
//
// It has been retitled to 'debounce2' since a different debounce module needs
// to be used for the magnetic door sensor.
//
////////////////////////////////////////////////////////////////////////////////

module debounce2 (reset, clock_50MHz, noisy, clean);
   input reset, clock_50MHz, noisy;
   output clean;

   reg [19:0] count;
   reg new, clean;

   always @(posedge clock_50MHz)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Lighting Module
//
// File:   lighting.v
// Last Updated:   December 13, 2005
// Author: Aaron Stonely
//
// This module controls the lighting level based upon user inputs and the
// ambient light value given by a photo sensor.
//
////////////////////////////////////////////////////////////////////////////////
```

```
module lighting(clk,reset,light_control,alarm,burglary,
                       sensor,light_value,light_state,time_selector);

input clk, reset, alarm, burglary,time_selector;
input [2:0] light_control;
input [7:0] sensor;
output [2:0] light_state;
output [7:0] light_value;

wire one_hz_enable, demo_enable;

reg [2:0] light_state;
reg [7:0] light_value;

one_hz_enable seconds(clk,reset,one_hz_enable);
demo_enable miniseconds(clk,reset,demo_enable);

wire wake_time_enable = time_selector ? demo_enable : one_hz_enable;

// Below the states for the lighting FSM are parameterized
parameter S_000 = 0; // Reset/Manual operating state
parameter S_001 = 1; // Automatic state
parameter S_010 = 2; // Sleep state
parameter S_011 = 3; // Burglary state

// Below, the keyboard control commands are parameterized.
parameter no_signal = 3'h0;
parameter off = 3'h1;
parameter on = 3'h2;
parameter auto = 3'h3;
parameter bright = 3'h4;
parameter dim = 3'h5;
parameter sleep = 3'h6;

// Below are several boolean tests to see if a specific
// command has been asserted by the keyboard.
wire no_command = (light_control == no_signal);
wire off_command = (light_control == off);
wire on_command = (light_control == on);
wire auto_command = (light_control == auto);
wire bright_command = (light_control == bright);
wire dim_command = (light_control == dim);
wire sleep_command = (light_control == sleep);
wire manual_command = (off_command | on_command |
                                 bright_command | dim_command);

// Lighting Finite State Machine (FSM)
// The below statement controls the transitions between
// lighting states.

always @ (posedge clk)
begin
    if(reset) light_state <= S_000;
    else
    case(light_state)
        S_000:         // Reset/Manual Operating State
              light_state <= burglary ? S_011 :
                                      auto_command ? S_001 :
                                      sleep_command ? S_010 :
                                      light_state;
        S_001:        // Automatic State
                light_state <= burglary ? S_011 :
                                      manual_command ? S_000 :
                                      sleep_command ? S_010 :
                                      light_state;
        S_010:        // Sleep State
                light_state <= burglary ? S_011 :
                                      manual_command ? S_000 :
                                      auto_command ? S_001 :
```

```
                                      light_state;
        S_011:        // Burglary State
                light_state <= burglary ? light_state : S_000;
        default:
                light_state <= S_000;
    endcase
end

// Manual Lighting Handler
//
// The following code is used to handle the manual behavior
// of the lighting module. It outputs lighting values based on
// user commands.

reg [7:0] manual_value;
reg [1:0] manual_state;

// Below two states are declared. Manual on and manual off,
// used to differentiate between operation when the lights
// are off and when the lights are on.
parameter manual_off = 0;
parameter manual_on = 1;

always @ (posedge clk)
begin
       manual_state <= reset ? manual_off :
                              off_command ? manual_off :
                              on_command ? manual_on :
                              bright_command ?  manual_on :
                              dim_command ? manual_on :
                              manual_state;
       manual_value <= reset ? 8'h0 :
                              on_command ? 8'hFF :
                               (manual_state == manual_on) ?
                                bright_command ?
                                (manual_value >= 220) ? 8'hFF :
                                (manual_value + 32) :
                                dim_command ?
                                (manual_value <= 32) ? 8'h0 :
                                (manual_value - 32) :
                                manual_value :
                                8'h0;

end

// Burglary Lighting Handler
//
// The following code is used to handle the burglary behavior
// of the lighting module. During a burglary, the lights flash
// on and off every 2 seconds.

reg [7:0] burglary_value;

always @ (posedge clk)
begin
       burglary_value <= reset ? 0 :
                                one_hz_enable ?
                                 (burglary_value == 0) ? 8'hFF : 0 :
                                 burglary_value;
end

// Wake-up Lighting Handler
//
// The following code is used to handle the wake-up behavior
// of the lighting module. Upon assertion of the alarm input
// this module begins turning on the lights from their off
// value to their maximum on value over a 15-minute sequence.

reg [7:0] wake_up_value;
reg wake_up_state;
reg [1:0] wake_update;
```

```
parameter inactive_wake = 0;
parameter active_wake = 1;

always @ (posedge clk)
begin
       wake_up_state <= reset ? inactive_wake :
                                alarm ?
                                (wake_up_state == inactive_wake) ?
                                     active_wake : inactive_wake :
                                wake_up_state;
       wake_up_value <= reset ? 8'h0 :
                                (wake_up_state == inactive_wake) ? 8'h0 :
                                (wake_up_value == 8'hFF) ? wake_up_value :
                                (wake_update == 3) ? (wake_up_value + 1) :
                                wake_up_value;
       wake_update <= reset ? 0 :
                           (wake_update == 3) ? 0 :
                              wake_time_enable ?
                              (wake_update + 1) :
                              wake_update;
end

// Automatic Lighting Handler
//
// The following code is used to handle the automatic behavior
// of the lighting module. It outputs lighting values based on
// the value of the ambient light received by the light sensor.

reg [7:0] auto_value;

// The lighting threshold for the sensor is half of
// the 8-bit light range.
//
// When the 8-bit lighting value is less than 8'h80,
// the lighting is at full value.
//
// Otherwise, lighting is set to full value
// minus the current value of the ambient light.

always @ (posedge clk)
begin
       auto_value <= reset ? 8'h0 :
                              one_hz_enable ?
                                     (sensor <= 8'h80) ? 8'hFF :
                                     (8'hFF - sensor) :
                                     auto_value;
end

// Lighting Control Register
// Chooses the value to output as the lighting value

always @ (posedge clk)
begin
       light_value <= reset ? manual_value :
                              (light_state == S_011) ? burglary_value :
                              (light_state == S_000) ? manual_value :
                              (light_state == S_001) ? auto_value :
                              (light_state == S_010) ? wake_up_value :
                              manual_value;
end

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// One_hz_enable Module
//
// File:   Lighting.v
// Last Updated:   December 13, 2005
// Author: Aaron Stonely
//
// Has a counter that counts to 27,000,000 then sets the 1hz_enable high and
// rolls over. This makes 1hz_enable be high for 1 clock cycle every 1 second.
//
// Assumes 27Mhz clock
//
////////////////////////////////////////////////////////////////////////////////

module one_hz_enable(clk,reset,one_hz_enable);

input clk, reset;
output one_hz_enable;

// In order to count to 27,000,000, 25 registers for count are needed.
reg [24:0] count=0;
reg one_hz_enable=0;

always @ (posedge clk)
begin
    if(reset)
    begin
        one_hz_enable <= 0;
        count <= 0;
    end
    else
    begin
        if(count == 26999999) // 27million - 1 for the clock delay
        begin
            count <= 0;
            one_hz_enable <= 1;
        end
        else
        begin
            one_hz_enable <= 0;
            count <= count + 1;
        end
    end
end

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Demo Enable Module
//
// File:   Lighting.v
// Last Updated:   December 13, 2005
// Author: Aaron Stonely
//
// Has a counter that counts to 500000, then sets the 1hz_enable high and
// rolls over. This makes demo_enable be high for 1 clock cycle every
// 1/60th second. This is used for demonstration purposes.
//
// Assumes 27Mhz clock
//
////////////////////////////////////////////////////////////////////////////////

module demo_enable(clk,reset,demo_enable);

input clk, reset;
output demo_enable;

// In order to count to 500,000, 19 registers for count are needed.
reg [18:0] count=0;
reg demo_enable=0;

always @ (posedge clk)
begin
    if(reset)
    begin
        demo_enable <= 0;
        count <= 0;
    end
    else
    begin
        if(count == 449999) // 500,000 - 1 for clock cycle delay.
        begin
            count <= 0;
            demo_enable <= 1;
        end
        else
        begin
            demo_enable <= 0;
            count <= count + 1;
        end
    end
end

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// ADC_Sampler Module
//
// File:   lighting.v
// Last Updated:   December 13, 2005
// Author: Aaron Stonely
//
// This module was to be used in conjunction with the lighting module in
// order to sample the light sensor voltages from the ADC0841
//
////////////////////////////////////////////////////////////////////////////////

module adc_sampler(clk,reset,one_hz_enable,write,read);

input clk, reset, one_hz_enable;
output write, read;

reg [1:0] count;
reg write, read;

// Below, the write and read are asserted low on every other
// clock cycle. By using the one_hz_enabl signal, a new light
// value voltage is read in every 4 seconds.

always @ (posedge clk)
begin
       count <= reset ? 0 :
                      one_hz_enable ?
                              (count == 3) ? 0 :
                              (count + 1) :
                      count;
       write <= reset ? 1 :
                   (count == 0) ? 0 : 1;
       read <= reset ? 1:
                   (count == 2) ? 0 : 1;
end

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Lock Module
//
// File:   lock.v
// Last Updated:   December 13, 2005
// Author: David Nedzel
//
```

```
//
// This module implements the lock feature of the dorm room genie taking user
// input to store and authenticate a password and control a lock
//
////////////////////////////////////////////////////////////////////////////////



module lock(clock, reset, char_data, lock_ctrl, door, state, unlock, pw_correct,
lock_done);

  input clock, reset; // system clock and synchronus reset
  input door; // door sensor '1' = open, '0' = closed
  input [7:0] char_data; // character data input for password storage
  input [2:0] lock_ctrl; // lock control signals -- based upon user input
  output [2:0] state; // outputs state of the lock to the display module
  output unlock; // '1' when the lock is unlocked, '0' when locked
  output pw_correct; // high for one clock cycle when the correct password has been
entered
  output lock_done; // high for one clock cycle to release the signal-processing
semaphore after setting the password.

  wire unlock;
  wire lock_done;
  reg pw_correct;

  reg [2:0] state; // 0 = locked, 1 = password partially correct (locked), 2 = entery, 3
= unlocked, 4 = inactive, 5 = password change
  reg [2:0] old_state; // stores previous state
  reg [3:0] memory_max; // stores password length
  reg pw_zero; // '1' if no password is set
  reg pw_end; // '1' if correct password has been input but the user has not pushed
"retun/enter"
  reg pw_max; // '1' if the password is of maximum length (16 characters)

  // passwrod RAM access & control signals
  wire [7:0] stored_data;
  reg [7:0] char_write;
  reg [3:0] memory_addr;
  reg memory_we;


  assign unlock = ((state == 2) || (state == 3)) ? 1 : 0; // lock is unlocked when module
is in 'entry' or 'unlocked' states
  assign lock_done = ((old_state == 5) && (state != 5)) ? 1 : 0; // lock module releases
signal-processing semaphore after exiting password change mode

  wire char_correct, char_wrong, stored_end, new_char;
  assign char_correct = (char_data == 8'b0) ? 0 : (char_data == stored_data); // correct
key has been entered in password sequence
  assign char_wrong = (char_data == 8'b0) ? 0 : !(char_data == stored_data); // incorrect
key has been entered in password sequence
  assign stored_end = pw_max ? (memory_addr == memory_max) : (memory_addr == memory_max -
1);
  assign new_char = ((char_data != 8'b0) && (char_data != 8'h0D)); // a new character
rather (not a return/enter signal) has been input

    always @(posedge clock) begin
       if (reset) begin
               state <= 1;
               old_state <= 1;
               memory_addr <= 0;
               memory_max <= 0;
               memory_we <= 0;
               char_write <= 0;
               pw_zero <= 1;
               pw_end <= 0;
               pw_max <= 0;
               pw_correct <= 0; end
       else case (state)
               0: begin // locked
```

```
                        old_state <= 0;
                        pw_correct <= 0;
                        memory_we <= 0;
                        state <= (lock_ctrl == 2) ? 3 : // unlock if user signals
                                    (lock_ctrl == 3) ? 4 : // inactivate if user signals
                                    (lock_ctrl == 5) ? 5 : // change password if user
signals
                                    (char_data == 8'h0D) ? 1 : 0; // transition to
partially correct password after user finishes entering an incorrect password
                  end
               1: begin // password partially correct
                        old_state <= 1;
                        memory_we <= 0;
                        pw_end <= (old_state != 1) ? 0 :
                                        (stored_end && char_correct) ? 1 : pw_end;

                        pw_correct <= (old_state != 1) ? 0 :
                                            ((pw_end || pw_zero) && (char_data == 8'h0D));

                        memory_addr <= (old_state != 1) ? 0 :
                                                ((!stored_end) && char_correct) ?
memory_addr + 1 : memory_addr;
                        state <= (lock_ctrl == 2) ? 3 : // unlock if user signals
                                    (lock_ctrl == 3) ? 4 : // inactivate if user signals
                                    (lock_ctrl == 5) ? 5 : // change password if user
signals
                                    ((pw_end || pw_zero) && (char_data == 8'h0D)) ? 2 : //
transition to entry if correct password is entered
                                    ((pw_end || pw_zero) && !(char_data == 8'b0)) ? 0 : //
transition to "locked" if incorrect key is pressed
                                    char_wrong ? 0 : 1;
                  end
               2: begin // entry
                        old_state <= 2;
                        pw_correct <= 0;
                        memory_we <= 0;
                        state <= (lock_ctrl == 1) ? 1 : // lock if user signals
                                    (lock_ctrl == 2) ? 3 : // unlock if user signals
                                    (lock_ctrl == 3) ? 4 : // inactivate if user signals
                                    (lock_ctrl == 5) ? 5 : // change password if user
signals
                                    door ? 1 : 2; // transition to partially correct
password after user enters
                  end
               3: begin // unlocked
                        old_state <= 3;
                        pw_correct <= 0;
                        memory_we <= 0;
                        state <= (lock_ctrl == 1) ? 1 : // lock if user signals
                                    (lock_ctrl == 3) ? 4 : // inactivate if user signals
                                    (lock_ctrl == 5) ? 5 : 3; // change password if user
signals
                  end
               4: begin // inactive
                        old_state <= 4;
                        pw_correct <= 0;
                        memory_we <= 0;
                        state <= (lock_ctrl == 1) ? 1 : // re-activate if user signals
                                    (lock_ctrl == 2) ? 3 : // unlock if user signals
                                    (lock_ctrl == 5) ? 5 : // change password if user
signals
                                    door ? 1 : 4; // re-activate if user opens the door
                  end
               5: begin // password change
                        old_state <= 5;
                        pw_correct <= 0;
                        state <= (char_data == 8'h0D) ? 1 : 5; // when enter is pushed
passord setting is complete
                        memory_max <= (old_state != 5) ? 0 :
                                            ((memory_max < 15) && new_char) ? (memory_max +
1) : memory_max;
```

```
                      memory_addr <= (old_state != 5) ? 0 :
                                         ((memory_addr < 15) && memory_we) ? (memory_addr
+ 1) : memory_addr;
                      memory_we <= (!pw_max && new_char);
                      char_write <= char_data;
                      pw_zero <= (memory_max == 0);
                      pw_max <= (old_state != 5) ? 0 :
                                     (memory_we && (memory_addr == 15)) ? 1 : pw_max;
                      end
              default: state <= 1;
       endcase
  end

  // implement password RAM
  ram16x8 password(.clk(clock), .we(memory_we), .addr(memory_addr),
                              .din(char_write), .dout(stored_data));

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// NTSC to Pixel Module
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang ichuang@mit.edu
//
// ******* NOTE ******
// This contains code written by Professor I. Chuang. It has been modified by
// David Nedzel. This module takes data from Javier's NTSC decoder and converts
// it to a pixel associated with hcount & vcount. Also the output has been scaled
// back from the original 1024x768 image size to the 320x240 image size used
// by the Dorm Room Genie.
//
//
////////////////////////////////////////////////////////////////////////////////


module ntsc_to_pixel(clk, vclk, fvh, dv, din, hcount, vcount, pixel, valid);

   input clk; // system clock (27MHz)
   input vclk; // video clock from camera
   input [2:0] fvh; // field, vertical and horizontal synchronization signals
   input dv; // data valid signal
   input [7:0] din; // input from ntsc decoder
   output [9:0] hcount, vcount;
   output [7:0] pixel;
   output valid;

   parameter    COL_START = 10'b0;
   parameter    ROW_START = 10'b0;

   // pixilize the luminance data from the ntsc decoder into the ram

   reg [9:0]    col = 0;
   reg [9:0]    row = 0;
   reg [7:0]    vdata = 0;
   reg          vwe;
   reg          old_dv;
   reg          old_frame;    // frames are even / odd interlaced
   reg          even_odd;     // decode interlaced frame to this wire

   wire         frame = fvh[2];
   wire         frame_edge = frame & ~old_frame;

   always @ (posedge vclk)
     begin
        old_dv <= dv;
        vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
        old_frame <= frame;
        even_odd = frame_edge ? ~even_odd : even_odd;

        if (!fvh[2])
          begin
             col <= fvh[0] ? COL_START :
                    (!fvh[2] && !fvh[1] && dv && (col < 320)) ? col + 1 : col;
             row <= fvh[1] ? ROW_START :
                    (!fvh[2] && fvh[0] && (row < 240)) ? row + 1 : row;
             vdata <= (dv && !fvh[2]) ? din : vdata;
          end
     end

   // synchronize with the system clock
   reg [9:0] x[1:0],y[1:0];
   reg [7:0] data[1:0];
   reg       we[1:0];

   always @(posedge clk)
     begin
        {x[1],x[0]} <= {x[0],col};
        {y[1],y[0]} <= {y[0],row};
        {data[1],data[0]} <= {data[0],vdata};
        {we[1],we[0]} <= {we[0],vwe};
     end

        wire [9:0] hcount, vcount;
        wire [7:0] pixel;
        wire valid;

        assign hcount = x[1];
        assign vcount = y[1];
        assign pixel = data[1];
        assign valid = we[1];

endmodule


//
// File:   ps2_kbd.v
// Date:   24-Oct-05
// Author: C. Terman / I. Chuang
//
// PS2 keyboard input for 6.111 labkit
//
// INPUTS:
//
//   clock_27mhz   - master clock
//   reset         - active high
//   clock         - ps2 interface clock
//   data          - ps2 interface data
//
// OUTPUTS:
//
//   ascii         - 8 bit ascii code for current character
//   ascii_ready   - one clock cycle pulse indicating new char received
//
//
// ******** NOTE ********
// This code has been modified by David Nedzel. An error in the handling of
// a backslash has been corrected and support for the function keys and the
// numeric keypad has been added.
//
///////////////////////////////////////////////////////////////////////////////

module ps2_ascii_input(clock_27mhz, reset, clock, data, ascii, ascii_ready);

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

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

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

   // get keycodes

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

   ps2 myps2(reset, clock_27mhz, clock, data, fifo_rd, fifo_data,
            fifo_empty,fifo_overflow);

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

   always @(posedge clock_27mhz)
     begin

        // get key if ready

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

        // raise ascii_ready for last key which was read

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

     end

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

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

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

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

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

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

         8'h05: ascii_val = 8'h81;           // F1 (lock)
         8'h06: ascii_val = 8'h82;           // F2 (unlock)
         8'h04: ascii_val = 8'h83;           // F3 (inactive)
         8'h0C: ascii_val = 8'h84;           // F4
         8'h03: ascii_val = 8'h85;           // F5 (arm)
         8'h0B: ascii_val = 8'h86;           // F6 (disarm)
         8'h83: ascii_val = 8'h87;           // F7
         8'h0A: ascii_val = 8'h88;           // F8
         8'h01: ascii_val = 8'h89;           // F9 (set wake time)
         8'h09: ascii_val = 8'h8A;           // F10 (set clock)
         8'h78: ascii_val = 8'h8B;           // F11 (set password)
         8'h07: ascii_val = 8'h8C;           // F12

         8'h69: ascii_val = 8'h91;           // num1 (alarm clock on)
         8'h72: ascii_val = 8'h92;           // num2 (alarm clock off)
         8'h7A: ascii_val = 8'h93;           // num3
         8'h6B: ascii_val = 8'h94;           // num4 (lights off)
         8'h73: ascii_val = 8'h95;           // num5 (lights sleep)
         8'h74: ascii_val = 8'h96;           // num6 (lights dim)
         8'h6C: ascii_val = 8'h97;           // num7 (lights on)
         8'h75: ascii_val = 8'h98;           // num8 (lights auto)
         8'h7D: ascii_val = 8'h99;           // num9 (lights bright)
         8'h70: ascii_val = 8'h9A;           // num0 (alarm clock snooze)
         8'h71: ascii_val = 8'h9B;           // num.


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

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

module ps2(reset, clock_27mhz, ps2c, ps2d, fifo_rd, fifo_data,
           fifo_empty,fifo_overflow);

   input clock_27mhz,reset;
   input ps2c;                // ps2 clock
   input ps2d;                // ps2 data
   input fifo_rd;             // fifo read request (active high)
   output [7:0] fifo_data;    // fifo data output
   output      fifo_empty;    // fifo empty (active high)
   output      fifo_overflow; // fifo overflow - too much kbd input
```

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

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

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

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

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0] ps2c_sync;
  always @ (posedge clock_27mhz) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire sample = ps2c_sync[2] & ~ps2c_sync[1];

  always @ (posedge clock_27mhz) begin
    if (reset) begin
      count <= 0;
      wptr <= 0;
      rptr <= 0;
      fifo_overflow <= 0;
    end
    else if (sample) begin
           // order of arrival: 0,8 bits of data (LSB first),odd parity,1
           if (count==10) begin
              // just received what should be the stop bit
              if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
                fifo[wptr] <= shift[8:1];
                wptr <= wptr_inc;
                fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
              end
              count <= 0;
          end else begin
              shift <= {ps2d,shift[9:1]};
              count <= count + 1;
          end
        end
    // bump read pointer if we're done with current value.
    // Read also resets the overflow indicator
    if (fifo_rd && !fifo_empty) begin
      rptr <= rptr + 1;
      fifo_overflow <= 0;
    end
  end

endmodule


/******************************************************************************
*     This file is owned and controlled by Xilinx and must be used            *
*     solely for design, simulation, implementation and creation of           *
*     design files limited to Xilinx devices or technologies. Use             *
*     with non-Xilinx devices or technologies is expressly prohibited         *
*     and immediately terminates your license.                                *
*                                                                             *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"           *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                 *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION         *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION             *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS               *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                 *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE        *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR          *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF         *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS         *
*     FOR A PARTICULAR PURPOSE.                                               *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2004 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file ram16x8.v when simulating
// the core, ram16x8. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Guide".

module ram16x8 (
	addr,
	clk,
	din,
	dout,
	we);    // synthesis black_box

input [3 : 0] addr;
input clk;
input [7 : 0] din;
output [7 : 0] dout;
input we;

// synopsys translate_off

	BLKMEMSP_V6_1 #(
		4,	// c_addr_width
		"0",	// c_default_data
		16,	// c_depth
		0,	// c_enable_rlocs
		1,	// c_has_default_data
		1,	// c_has_din
		0,	// c_has_en
		0,	// c_has_limit_data_pitch
		0,	// c_has_nd
		0,	// c_has_rdy
		0,	// c_has_rfd
		0,	// c_has_sinit
		1,	// c_has_we
		18,	// c_limit_data_pitch
		"mif_file_16_1",	// c_mem_init_file
		0,	// c_pipe_stages
		0,	// c_reg_inputs
		"0",	// c_sinit_value
		8,	// c_width
		0,	// c_write_mode
		"0",	// c_ybottom_addr
		1,	// c_yclk_is_rising
		1,	// c_yen_is_high
		"hierarchy1",	// c_yhierarchy
		0,	// c_ymake_bmm
		"16kx1",	// c_yprimitive_type
		1,	// c_ysinit_is_high
		"1024",	// c_ytop_addr
		0,	// c_yuse_single_primitive
		1,	// c_ywe_is_high
		1)	// c_yydisable_warnings
	inst (
		.ADDR(addr),
		.CLK(clk),
		.DIN(din),
		.DOUT(dout),
		.WE(we),
		.EN(),
               .ND(),
               .RFD(),
               .RDY(),
               .SINIT());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of ram16x8 is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of ram16x8 is "black_box"

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Signal Processor Module
//
// File:   signal_processor.v
// Last Updated:   December 13, 2005
// Author: David Nedzel
//
// This module takes keyboard data produced from a modified 6.111 staff
// ps2_ascii_input module and translates this user input into control signals
// and data which it then outputs to other modules.
//
////////////////////////////////////////////////////////////////////////////////

module signal_processor(clock, reset, ascii_data, ascii_ready, lock_done, time_done,
wake_done, burglary,
                              state, char_data, num_data, lock_ctrl, burg_ctrl,
lght_ctrl, time_ctrl, wake_ctrl, pic_ctrl);

  input clock, reset; // system clock & sychronus reset
  input ascii_ready; // signal indicating new keyboard data is available
  input lock_done, time_done, wake_done; // signals to release signal processor semaphore
  input burglary; // indicates a burglary in progress
  input [7:0] ascii_data; // keystroke data from keyboard
  output [2:0] state; // outputs current state to the display
  output [7:0] char_data; // character data passed to lock module (8-bit ascii format)
  output [3:0] num_data; // numeric data passed to time and wake modules (4-bit hex
format)
  output [2:0] lock_ctrl; // control signal to the lock module
  output [1:0] burg_ctrl; // control signal to the burglary module
  output [2:0] lght_ctrl; // control signal to the lighting module
  output time_ctrl; // control signal to the clock module
  output [2:0] wake_ctrl; // control signal to the alarm clock module
  output [1:0] pic_ctrl; // control signal for the picture stoarage feature of the video
module

  wire [7:0] char_data;
  wire [3:0] num_data;
  wire [2:0] lock_ctrl;
  wire [1:0] burg_ctrl;
  wire [2:0] lght_ctrl;
  wire time_ctrl;
  wire [2:0] wake_ctrl;

  reg [2:0] state; // 0: normal, 1: lock, 2: clock, 3: wake, 4: burglary


  // In "normal" mode, send control signals
  wire send_signal;
  assign send_signal = ((state == 0) && ascii_ready);

  // maps keystrokes to lock control signals
  assign lock_ctrl = (!send_signal) ? 3'h0 : // no signal
                            (ascii_data == 8'h81) ? 3'h1 : // lock
                              (ascii_data == 8'h85) ? 3'h1 : // lock (when alarm is set
to arm)
                            (ascii_data == 8'h82) ? 3'h2 : // unlock
                            (ascii_data == 8'h83) ? 3'h3 : // inactive
                            //(ascii_data == ) ? 3'h4 :
                            (ascii_data == 8'h8B) ? 3'h5 : // change password
                            //(ascii_data == ) ? 3'h6 : //
                            //(ascii_data == ) ? 3'h7 : //
                            3'h0; // default (no signal)

  // maps keystrokes to burglar alarm control signals
  assign burg_ctrl = (!send_signal) ? 2'h0 : // no signal
                            (ascii_data == 8'h85) ? 2'h1 : // arm
                            (ascii_data == 8'h86) ? 2'h2 : // disarm
                            (ascii_data == 8'h8B) ? 2'h2 : // disarm (when going into
password change mode)
                            //(ascii_data == ) ? 2'h3 : //
                              (state == 1) ? 2'h2 : // if in password set, disarm.
                            2'h0; // default (no signal)

  // maps keystrokes to lighting contol signals
  assign lght_ctrl = (!send_signal) ? 3'h0 : // no signal
                            (ascii_data == 8'h94) ? 3'h1 : // off
                            (ascii_data == 8'h97) ? 3'h2 : // on
                            (ascii_data == 8'h98) ? 3'h3 : // auto
                            (ascii_data == 8'h99) ? 3'h4 : // bright
                            (ascii_data == 8'h96) ? 3'h5 : // dim
                            (ascii_data == 8'h95) ? 3'h6 : // sleep
                            //(ascii_data == ) ? 3'h7 : //
                            3'h0; // default (no signal)

  // maps keystrokes to clock control signals
  assign time_ctrl = (!send_signal) ? 1'b0 : // no signal
                            (ascii_data == 8'h8A) ? 1'b1 : // set clock time
                            1'b0; // default (no signal)

  // maps keystorkes to alarm clock control signals
  assign wake_ctrl = (!send_signal) ? 3'h0 : // no signal
                            (ascii_data == 8'h91) ? 3'h1 : // alarm on
                            (ascii_data == 8'h92) ? 3'h2 : // alarm off
                            (ascii_data == 8'h9A) ? 3'h3 : // snooze
                            (ascii_data == 8'h89) ? 3'h4 : // set wake-up time
                            //(ascii_data == ) ? 3'h5 : //
                            //(ascii_data == ) ? 3'h6 : //
                            //(ascii_data == ) ? 3'h7 : //
                            3'h0; // default (no signal)

  // maps keystrokes to picture control signals
  assign pic_ctrl = (!send_signal) ? 2'b00 : // no signal
                            (ascii_data == 8'h88) ? 2'b01 : // next pic
                            (ascii_data == 8'h87) ? 2'b10 : // previous pic
                            //(ascii_data == ) ? 2'b11 : //

                            2'b00; // default       (no signal)

  // The following statments map keyboard characters to char_data and num_data

  wire valid_alphanum; // indicates that the keyboard data is a valid character for
password use
  assign valid_alphanum = ascii_ready &&
                                      ((ascii_data == 8'h0D) || // "enter" used as an
acknowlege signal, not a character
                                       (ascii_data == 8'h20) || // space
                                       (ascii_data == 8'h27) || // '
                                       (ascii_data == 8'h3B) || // ;
                                       (ascii_data == 8'h3D) || // =
                                       (ascii_data == 8'h60) || // `
                                       ((ascii_data >= 8'h2C) &&
                                             (ascii_data <= 8'h39)) || // ,-./ and
numbers 0-9
                                        ((ascii_data >= 8'h41) &&
                                             (ascii_data <= 8'h5D))); // []\ and 26
letters

  assign char_data = valid_alphanum ? ascii_data : 8'b0;

  wire valid_num; // indicates that the keyboard data is a valid number (0-9)
  assign valid_num = ascii_ready &&
                               ((ascii_data >= 8'h30) &&
                                     (ascii_data <= 8'h39));

  // converts number from ascii to hex format
  assign num_data = valid_num ? ascii_data[3:0] : 4'hF;

  // Implements an FSM, esentially a semaphore which allows the lock, time, wake and
alarm modules to interrupt the signal processors normal function.

  always @(posedge clock) begin
       if (reset) state <= 0;
       else case (state)
              0: state <= (ascii_ready && (ascii_data == 8'h8B)) ? 1 : // if "password
change"
                                (ascii_ready && (ascii_data == 8'h8A)) ? 2 : // if "set
clock"
                                (ascii_ready && (ascii_data == 8'h89)) ? 3 : // if "set
alarm time"
                                burglary ? 4 : 0;
              1: state <= lock_done ? 0 : 1;
              2: state <= time_done ? 0 :
                                burglary ? 4 : 2;
              3: state <= wake_done ? 0 :
                                burglary ? 4 : 3;
              4: state <= burglary ? 4 : 0;
              default: state <= 0;
       endcase
  end

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Synchronizer Module
//
// File:   synchonize.v
//
// ****** NOTE ******
// This module consists entirely of code provided by the 6.111 course staff.
//
////////////////////////////////////////////////////////////////////////////////



// pulse synchronizer
module synchronize(clk,in,out);
  parameter NSYNC = 4;  // number of sync flops.  must be >= 2
  input clk;
  input in;
  output out;

  reg [NSYNC-2:0] sync;
  reg out;

  always @ (posedge clk)
  begin
    {out,sync} <= {sync[NSYNC-2:0],in};
  end
endmodule
```

```
//
// File:   vga_sync.v
// Date:   04-Nov-05
// Author: C. Terman / I. Chuang
//
// MIT 6.111 Fall 2005
//
// Verilog code to produce VGA sync signals (and blanking) for 640x480 screen
//

module vga_sync(clk,hsync,vsync,hcount,vcount,pix_clk,blank);

   input clk;     // 50Mhz
   output hsync;
   output vsync;
   output [9:0] hcount, vcount;
   output      pix_clk;
   output      blank;

   // pixel clock: 25Mhz = 40ns (clk/2)
   reg         pcount;          // used to generate pixel clock
   wire        en = (pcount == 0);
   always @ (posedge clk) pcount <= ~pcount;
   wire        pix_clk = ~en;

   //****************************************************************
   //****************************************************************
   //***
   //***  Sync and Blanking Signals
   //***
   //****************************************************************
   //****************************************************************

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

   // horizontal: 794 pixels = 31.76us
   // display 640 pixels per line
   wire        hsyncon,hsyncoff,hreset,hblankon;
   assign      hblankon = en & (hcount == 639);
   assign      hsyncon = en & (hcount == 652);
   assign      hsyncoff = en & (hcount == 746);
   assign      hreset = en & (hcount == 793);

   wire        blank =  (vblank | (hblank & ~hreset));    // blanking => black
   //wire      blank = vblank | hblank;

   // vertical: 528 lines = 16.77us
   // display 480 lines
   wire        vsyncon,vsyncoff,vreset,vblankon;
   assign      vblankon = hreset & (vcount == 479);
   assign      vsyncon = hreset & (vcount == 492);
   assign      vsyncoff = hreset & (vcount == 494);
   assign      vreset = hreset & (vcount == 527);

   // sync and blanking
   always @(posedge clk) begin
      hcount <= en ? (hreset ? 0 : hcount + 1) : hcount;
      hblank <= hreset ? 0 : hblankon ? 1 : hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;   // hsync is active low

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

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
// Video Module
//
// File:   video.v
// Last Updated:   December 13, 2005
// Author: David Nedzel
//
// This module takes in camara input and stores images captured during a
// burglary. The module will also sense motion, and outputs both a "motion
// detected signal" to the burglar alarm and the stored images to the display.
//
// **** NOTE ****
// Those module has not yet been completed, and was not used in our project
// demonstration.
//
////////////////////////////////////////////////////////////////////////////////

module video(reset, clock_27mhz, clock_50mhz, burglary, pic_ctrl, tv_in_line_clock1,
tv_in_ycrcb,
                  vram_we, vram_addr, vram_write_data, vram_read_data,
                  vga_pixel, vga_hsync, vga_vsync, vga_pix_clk, vga_blank, motion,
ledtest, switch);

   input reset, clock_27mhz; // system clock & synchronous reset
   input clock_50mhz; // clock for driving the display
   input burglary; // indicates a burlary, causing module to record
   input tv_in_line_clock1; // video in clock
   input [9:0] tv_in_ycrcb; // video in lummenance data
   input [1:0] pic_ctrl; // user control of picture display
   input [35:0] vram_read_data; // read line of zbt RAM
   output [7:0] vga_pixel; // display pixel data
   output vga_hsync, vga_vsync, vga_pix_clk, vga_blank; // display signals
   output motion; // indicates motion, active high
   output vram_we; // write enable for zbt RAM
   output [18:0] vram_addr; // address line for zbt RAM
   output [35:0] vram_write_data; // write data for zbt RAM

   input [7:0] switch; // input for debuggin purposes
   output [7:0] ledtest; // output for debuggin purposes

   reg [4:0] num_pics; // number of pictures stored
   reg [4:0] disp_pic; // picture currently being displayed
   reg [1:0] state; // state of video module
   reg [1:0] old_state;
   reg capture; // '1' while waiting for a new frame to capture an image
   reg [35:0] vram_write_data;
   reg vram_we;
   reg [20:0] zbt_rtaddr; // stores the location where a pixel will be read from zbt RAM
   reg [20:0] voffset, pic_addroffset; // track the index of a pixel to avoid large
multiplications
   reg [9:0] hcount, vcount;
   reg frame_done; // tracks the end of a frame of input video
   reg dram_we;
   reg [7:0] dram_data_in;
   reg [16:0] dram_addr;


   // Create 1/2 second & 2 second enables
   // 1/2 second enable used to trigger a frame capture for motion detection
   // 2 second enable used to trigger a frame capture for zbt storage
   wire half_sec, two_sec;
   reg [1:0] twosec; // will be advanced each time half_sec is high
   divider halfsec(clock_27mhz, reset, half_sec);
   assign two_sec = (half_sec && (twosec == 0)) ? 1 : 0;

   // VGA display (640x480 resolution)
   wire vga_hsync, vga_vsync, vga_pix_clk, vga_blank;
   wire [7:0] dram_data_out, vga_pixel;
   wire [9:0] vga_hcount, vga_vcount;
   assign vga_pixel = (state == 0) ? dram_data_out : 8'hFF;
   vga_sync vig_signals(clock_50mhz, vga_hsync, vga_vsync, vga_hcount, vga_vcount,
vga_pix_clk, vga_blank);

   // Decode Input Video
   wire [29:0] ycrcb;  // video data (luminance, chrominance)
   wire f, v, h, dv;   // field sync, vertical sync, horizontal sync, data valid

   ntsc_decode decode(tv_in_line_clock1, reset, tv_in_ycrcb, ycrcb, f, v, h, dv);

   // Convert NTSC decoded data to indexed pixel format
   wire [7:0] pixel;
   wire [8:0] col, row;
   wire valid;

   ntsc_to_pixel pixels(clock_27mhz, tv_in_line_clock1, {f, v, h}, dv, ycrcb[29:22], col,
row, pixel, valid);

   // Prepare Data for zbt storage & retreval
   wire [18:0] vram_addr;
   wire [20:0] zbt_staddr;
   assign zbt_staddr = (pic_addroffset + (row * 320) + col); // pixel index of a pixel to
be written into zbt RAM
   assign vram_addr = (state == 3) ? zbt_rtaddr[20:2] : zbt_staddr[20:2]; // divide by 4
for address (4 pixels per location)

   // The following prepares a 36 bit word (4 pixels) to be written into zbt RAM
   always @(posedge clock_27mhz) begin
        if (reset) begin
               vram_write_data <= 0;
               vram_we <= 0; end
        vram_write_data <= ((col[1:0] == 2'b00) && valid) ?
                                             {4'b0, pixel, vram_write_data[23:0]} :
                                  ((col[1:0] == 2'b01) && valid) ?
                                             {4'b0, vram_write_data[31:24], pixel,
vram_write_data[15:0]} :
                                  ((col[1:0] == 2'b10) && valid) ?
                                             {4'b0, vram_write_data[31:16], pixel,
vram_write_data[7:0]} :
                                  ((col[1:0] == 2'b11) && valid) ?
                                             {4'b0, vram_write_data[31:8], pixel} :
vram_write_data;
        vram_we <= ((col[1:0] == 2'b11) && valid) ? 1 : 0;
   end

   // Prepare the display ram to recieve from zbt & output to display
   wire [16:0] dram_addr;
   wire [9:0] hcount_delay, vcount_delay;

   // The zbt RAM introduces a 2 clock-cycle delay, therefore it is necessary to delay
these signals by 2 clock cycles as well.
   delay2count delayhcount(clock_27mhz, hcount, hcount_delay);
   delay2count delayvcount(clock_27mhz, vcount, vcount_delay);
   delayN delaydram_we(clock_27mhz, dram_we, dram_we_delay);


   // If storing data from zbt to display RAM, assign address based upon the retrieved
data's zbt location
   // If reading data from the display RAM, assign the address based upon the vga display
signals
   assign dram_addr = (state == 3) ? (hcount_delay + (320 * vcount_delay)) :
((vga_hcount/2) + ((vga_vcount/2) * 320));
   wire [7:0] dram_data_in;
   assign dram_data_in = (vcount[1:0] == 2'b00) ? vram_read_data[31:24] :
                                   (vcount[1:0] == 2'b01) ? vram_read_data[23:16] :
                                   (vcount[1:0] == 2'b10) ? vram_read_data[15:8] :
                                   (vcount[1:0] == 2'b11) ? vram_read_data[7:0] :
8'h00;

   // synchronize 27mhz dram signals with with the system clock
   // Modeled after 6.111 staff provided "ntsc2zbt.v"
   reg [9:0] dram50_hcount[1:0], dram50_vcount[1:0];
   reg [7:0] dram50_data_in[1:0];
   reg       dram50_we[1:0];

   always @(posedge clock_50mhz)
     begin
        {dram50_hcount[1], dram50_hcount[0]} <= {dram50_hcount[0], hcount_delay};
        {dram50_vcount[1], dram50_vcount[0]} <= {dram50_vcount[0], vcount_delay};
        {dram50_data_in[1], dram50_data_in[0]} <= {dram50_data_in[0], dram_data_in};
        {dram50_we[1], dram50_we[0]} <= {dram50_we[0], dram_we_delay};
        //dram_addr <= (state == 3) ? (dram50_hcount[1] + (320 * dram50_vcount[1])) :
(vga_hcount + (vga_vcount * 320));
        dram_addr <= (state == 3) ? (dram50_hcount[1] + (320 * dram50_vcount[1])) :
((vga_hcount/2) + ((vga_vcount/2) * 320));
     end

   // Implementation of display RAM
   ram76800x8 dram(.clk(clock_50mhz), .we(dram_we), .addr(dram_addr),
                                .din(dram_data_in), .dout(dram_data_out));

/* Alternate display RAM implementation for debuggin purposes

   wire [16:0] address;
   assign address = burglary ? (col + (row * 320)) : ((vga_hcount/2) + ((vga_vcount/2) *
320));

   ram76800x8 dram(.clk(clock_50mhz), .we(valid && burglary), .addr(address),
                                .din(pixel), .dout(dram_data_out));
*/

   always @(posedge clock_27mhz) begin
        if (reset) begin
                num_pics <= 0;
                disp_pic <= 0;
                state <= 0;
                old_state <= 0;
                capture <= 0;
                hcount <= 0;
                vcount <= 0;
                voffset <= 0;
                frame_done <= 0;
                zbt_rtaddr <= 0;
                pic_addroffset <= 0;
                twosec <= 0; end
        twosec <= (half_sec) ? (twosec + 1) : twosec;
        case (state)
                0: begin // playback mode
                        old_state <= state;
                        disp_pic <= ((old_state == 1) && (num_pics > 0)) ? 0 :
                                        (pic_ctrl[0] && (num_pics < 27)) ? (disp_pic + 1)
:
                                        (pic_ctrl[1] && (num_pics > 0)) ? (disp_pic - 1) :
disp_pic; // choose which stored image to display
                        pic_addroffset <= ((old_state == 1) && (num_pics > 0)) ? 0 :
                                                (pic_ctrl[0] && (num_pics < 27)) ?
(pic_addroffset + 76800) :
                                                (pic_ctrl[1] && (num_pics > 0)) ?
(pic_addroffset - 76800) : pic_addroffset; // update address offset based upon picture
chosen
                        state <= (burglary) ? 1 : // if there is a burglary, transition to
record
                                    (pic_ctrl != 0) ? 3 : // load a picture as the user
requests
                                    (old_state == 1) ? 3 : // load a picture after
recording is complete
                                    0; end // otherwise, stay in playback mode
                1: begin // record ready mode
                        old_state <= state;
                        num_pics <= ((old_state == 0) || (old_state == 3)) ? 0 : num_pics;
// if a new burglary, delete previously stored pictures
```

```
                      pic_addroffset <= ((old_state == 0) || (old_state == 3)) ? 0 :
pic_addroffset;
                      capture <= (old_state != state) ? 0 :
                                      (two_sec && (num_pics < 27)) ? 1 : capture;
                      state <= (!burglary) ? 0 : // if the burglary is over, transition
to playback
                                 ((row == 239) && (col == 319) && capture && (old_state
== state)) ? 2 : // if it's time, transition to image capture when the next frame begins
                                 1; end // otherwise, stay in record ready
               2: begin // image capture mode
                      old_state <= state;
                      frame_done <= ((row == 239) && (col == 318)) ? 1 : 0;
                      num_pics <= ((row == 239) && (col == 319) && frame_done) ?
(num_pics + 1) : num_pics; // advance after a write
                      pic_addroffset <= ((row == 239) && (col == 319) && frame_done) ?
(pic_addroffset + 76800) : pic_addroffset;
                      state <= ((row == 239) && (col == 319) && frame_done) ? 1 : 2; end
               3: begin // load image mode
                      old_state <= state;
                      hcount <= (old_state != state) ? 0 :
                                     (hcount == 319) ? 0 : (hcount + 1);
                      vcount <=      (old_state != state) ? 0 :
                                     (hcount == 319) ? (vcount + 1) : vcount;
                      voffset <= (old_state != state) ? 0 :
                                      (hcount == 319) ? (voffset + 320) : voffset;
                      zbt_rtaddr <= (pic_addroffset + voffset + hcount); // calculate
address to read from zbt
                      state <= (old_state != state) ? 3 :
                                 ((hcount == 319) && (vcount == 239)) ? 0 : 3; // if
images has been retrieved, return to playback mode
                      dram_we <= (old_state != state) ? 1 :
                                      ((hcount == 319) && (vcount == 239)) ? 0 : 1; end
               default: state <= 0;

       endcase
   end

/* The following are signals used for debugging:
   reg hhh5 = 0, hhh2 = 0, hhp = 0;

//   assign ledtest[7:6] = ~state;
//   assign ledtest[5] = hhp;
//   assign ledtest[4:0] = ~num_pics;
//      assign ledtest = (state == 2) ? ~row[7:0] : 8'hFF;
       assign ledtest = ~vram_write_data[15:8];
   always @(posedge half_sec) hhh5 <= hhh5 + 1;
   always @(posedge two_sec) hhh2 <= hhh2 + 1;
   always @(posedge state[1]) hhp <= hhp + 1;


endmodule


//
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
//
// This file contains the ntsc_decode and adv7185init modules
//
// These modules are used to grab input NTSC video data from the RCA
// phono jack on the right hand side of the 6.111 labkit (connect
// the camera to the LOWER jack).
//

/////////////////////////////////////////////////////////////////////////////
//
// NTSC decode - 16-bit CCIR656 decoder
// By Javier Castro
// This module takes a stream of LLC data from the adv7185
// NTSC/PAL video decoder and generates the corresponding pixels,
// that are encoded within the stream, in YCrCb format.

// Make sure that the adv7185 is set to run in 16-bit LLC2 mode.

module ntsc_decode(clk, reset, tv_in_ycrcb, ycrcb, f, v, h, data_valid);

   // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
   // reset - system reset
   // tv_in_ycrcb - 10-bit input from chip. should map to pins [19:10]
   // ycrcb - 24 bit luminance and chrominance (8 bits each)
   // f - field: 1 indicates an even field, 0 an odd field
   // v - vertical sync: 1 means vertical sync
   // h - horizontal sync: 1 means horizontal sync

   input clk;
   input reset;
   input [9:0] tv_in_ycrcb; // modified for 10 bit input - should be P[19:10]
   output [29:0] ycrcb;
   output      f;
   output      v;
   output      h;
   output      data_valid;
   // output [4:0] state;

   parameter   SYNC_1 = 0;
   parameter   SYNC_2 = 1;
   parameter   SYNC_3 = 2;
   parameter   SAV_f1_cb0 = 3;
   parameter   SAV_f1_y0 = 4;
   parameter   SAV_f1_cr1 = 5;
   parameter   SAV_f1_y1 = 6;
   parameter   EAV_f1 = 7;
   parameter   SAV_VBI_f1 = 8;
   parameter   EAV_VBI_f1 = 9;
   parameter   SAV_f2_cb0 = 10;
   parameter   SAV_f2_y0 = 11;
   parameter   SAV_f2_cr1 = 12;
   parameter   SAV_f2_y1 = 13;
   parameter   EAV_f2 = 14;
   parameter   SAV_VBI_f2 = 15;
   parameter   EAV_VBI_f2 = 16;




   // In the start state, the module doesn't know where
   // in the sequence of pixels, it is looking.

   // Once we determine where to start, the FSM goes through a normal
   // sequence of SAV process_YCrCb EAV... repeat

   // The data stream looks as follows
   // SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | EAV
sequence
   // There are two things we need to do:
   //   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
   //   2. Decode the subsequent data

   reg [4:0]   current_state = 5'h00;
   reg [9:0]   y = 10'h000;  // luminance
   reg [9:0]   cr = 10'h000; // chrominance
   reg [9:0]   cb = 10'h000; // more chrominance

   assign      state = current_state;

   always @ (posedge clk)
     begin
       if (reset)
         begin

         end
```

```
        else
          begin
             // these states don't do much except allow us to know where we are in the
stream.
             // whenever the synchronization code is seen, go back to the sync_state
before
             // transitioning to the new state
             case (current_state)
               SYNC_1: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_2 : SYNC_1;
               SYNC_2: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_3 : SYNC_1;
               SYNC_3: current_state <= (tv_in_ycrcb == 10'h200) ? SAV_f1_cb0 :
                                       (tv_in_ycrcb == 10'h274) ? EAV_f1 :
                                       (tv_in_ycrcb == 10'h2ac) ? SAV_VBI_f1 :
                                       (tv_in_ycrcb == 10'h2d8) ? EAV_VBI_f1 :
                                       (tv_in_ycrcb == 10'h31c) ? SAV_f2_cb0 :
                                       (tv_in_ycrcb == 10'h368) ? EAV_f2 :
                                       (tv_in_ycrcb == 10'h3b0) ? SAV_VBI_f2 :
                                       (tv_in_ycrcb == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;

               SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y0;
               SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cr1;
               SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y1;
               SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cb0;

               SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y0;
               SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cr1;
               SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y1;
               SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cb0;

               // These states are here in the event that we want to cover these signals
               // in the future. For now, they just send the state machine back to SYNC_1
               EAV_f1: current_state <= SYNC_1;
               SAV_VBI_f1: current_state <= SYNC_1;
               EAV_VBI_f1: current_state <= SYNC_1;
               EAV_f2: current_state <= SYNC_1;
               SAV_VBI_f2: current_state <= SYNC_1;
               EAV_VBI_f2: current_state <= SYNC_1;

             endcase
          end
      end // always @ (posedge clk)

    // implement our decoding mechanism

    wire y_enable;
    wire cr_enable;
    wire cb_enable;

    // if y is coming in, enable the register
    // likewise for cr and cb
    assign y_enable = (current_state == SAV_f1_y0) ||
                      (current_state == SAV_f1_y1) ||
                      (current_state == SAV_f2_y0) ||
                      (current_state == SAV_f2_y1);
    assign cr_enable = (current_state == SAV_f1_cr1) ||
                       (current_state == SAV_f2_cr1);
    assign cb_enable = (current_state == SAV_f1_cb0) ||
                       (current_state == SAV_f2_cb0);

    // f, v, and h only go high when active
    assign {v,h} = (current_state == SYNC_3) ? tv_in_ycrcb[7:6] : 2'b00;

    // data is valid when we have all three values: y, cr, cb
    assign data_valid = y_enable;
    assign ycrcb = {y,cr,cb};

    reg    f = 0;

    always @ (posedge clk)
      begin
        y <= y_enable ? tv_in_ycrcb : y;
        cr <= cr_enable ? tv_in_ycrcb : cr;
        cb <= cb_enable ? tv_in_ycrcb : cb;
        f <= (current_state == SYNC_3) ? tv_in_ycrcb[8] : f;
     end

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ADV7185 Video Decoder Configuration Init
//
// Created:
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
// Register 0
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SELECT                           4'h0
  // 0: CVBS on AIN1 (composite video in)
  // 7: Y on AIN2, C on AIN5 (s-video in)
  // (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE                             4'h0
  // 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
  // 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
  // 2: Autodetect: NTSC or PAL (N), w/o pedestal
  // 3: Autodetect: NTSC or PAL (N), w/pedestal
  // 4: NTSC w/o pedestal
  // 5: NTSC w/pedestal
  // 6: NTSC 4.43 w/o pedestal
  // 7: NTSC 4.43 w/pedestal
  // 8: PAL BGHID w/o pedestal
  // 9: PAL N w/pedestal
  // A: PAL M w/o pedestal
  // B: PAL M w/pedestal
  // C: PAL combination N
  // D: PAL combination N w/pedestal
  // E-F: [Not valid]

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 1
///////////////////////////////////////////////////////////////////////////////

`define VIDEO_QUALITY                          2'h0
  // 0: Broadcast quality
  // 1: TV quality
  // 2: VCR quality
  // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE                   1'b0
  // 0: Normal mode
  // 1: Square pixel mode
`define DIFFERENTIAL_INPUT                     1'b0
  // 0: Single-ended inputs
  // 1: Differential inputs
`define FOUR_TIMES_SAMPLING                    1'b0
  // 0: Standard sampling rate
  // 1: 4x sampling rate (NTSC only)
`define BETACAM                                1'b0
  // 0: Standard video input
  // 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE               1'b1
  // 0: Change of input triggers reacquire
  // 1: Change of input does not trigger reacquire

`define ADV7185_REGISTER_1 {`AUTOMATIC_STARTUP_ENABLE, 1'b0, `BETACAM,
`FOUR_TIMES_SAMPLING, `DIFFERENTIAL_INPUT, `SQUARE_PIXEL_IN_MODE, `VIDEO_QUALITY}
```

```
///////////////////////////////////////////////////////////////////////////////
// Register 2
///////////////////////////////////////////////////////////////////////////////

`define Y_PEAKING_FILTER                         3'h4
  // 0: Composite =  4.5dB,  s-video =  9.25dB
  // 1: Composite =  4.5dB,  s-video =  9.25dB
  // 2: Composite =  4.5dB,  s-video =  5.75dB
  // 3: Composite =  1.25dB, s-video =  3.3dB
  // 4: Composite =  0.0dB,  s-video =  0.0dB
  // 5: Composite = -1.25dB, s-video = -3.0dB
  // 6: Composite = -1.75dB, s-video = -8.0dB
  // 7: Composite = -3.0dB,  s-video = -8.0dB
`define CORING                                   2'h0
  // 0: No coring
  // 1: Truncate if Y < black+8
  // 2: Truncate if Y < black+16
  // 3: Truncate if Y < black+32

`define ADV7185_REGISTER_2 {3'b000, `CORING, `Y_PEAKING_FILTER}

///////////////////////////////////////////////////////////////////////////////
// Register 3
///////////////////////////////////////////////////////////////////////////////

`define INTERFACE_SELECT                         2'h0
  // 0: Philips-compatible
  // 1: Broktree API A-compatible
  // 2: Broktree API B-compatible
  // 3: [Not valid]
`define OUTPUT_FORMAT                            4'h0
  // 0: 10-bit @ LLC, 4:2:2 CCIR656
  // 1: 20-bit @ LLC, 4:2:2 CCIR656
  // 2: 16-bit @ LLC, 4:2:2 CCIR656
  // 3: 8-bit @ LLC, 4:2:2 CCIR656
  // 4: 12-bit @ LLC, 4:1:1
  // 5-F: [Not valid]
  // (Note that the 6.111 labkit hardware provides only a 10-bit interface to
  // the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS                  1'b0
  // 0: Drivers tristated when ~OE is high
  // 1: Drivers always tristated
`define VBI_ENABLE                               1'b0
  // 0: Decode lines during vertical blanking interval
  // 1: Decode only active video regions

`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS, `OUTPUT_FORMAT,
`INTERFACE_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 4
///////////////////////////////////////////////////////////////////////////////

`define OUTPUT_DATA_RANGE                        1'b0
  // 0: Output values restricted to CCIR-compliant range
  // 1: Use full output range
`define BT656_TYPE                               1'b0
  // 0: BT656-3-compatible
  // 1: BT656-4-compatible

`define ADV7185_REGISTER_4 {`BT656_TYPE, 3'b000, 3'b110, `OUTPUT_DATA_RANGE}

///////////////////////////////////////////////////////////////////////////////
// Register 5
///////////////////////////////////////////////////////////////////////////////


`define GENERAL_PURPOSE_OUTPUTS                  4'b0000
`define GPO_0_1_ENABLE                           1'b0
  // 0: General purpose outputs 0 and 1 tristated
  // 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE                      1'b0
  // 0: General purpose outputs 2 and 3 tristated
  // 1: General purpose outputs 2 and 3 enabled
`define BLANK_CHROMA_IN_VBI                 1'b1
  // 0: Chroma decoded and output during vertical blanking
  // 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE                        1'b0
  // 0: GPO 0 is a general purpose output
  // 1: GPO 0 shows HLOCK status

`define ADV7185_REGISTER_5 {`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI, `GPO_2_3_ENABLE,
`GPO_0_1_ENABLE, `GENERAL_PURPOSE_OUTPUTS}

///////////////////////////////////////////////////////////////////////////////
// Register 7
///////////////////////////////////////////////////////////////////////////////

`define FIFO_FLAG_MARGIN                    5'h10
  // Sets the locations where FIFO almost-full and almost-empty flags are set
`define FIFO_RESET                          1'b0
  // 0: Normal operation
  // 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET                1'b0
  // 0: No automatic reset
  // 1: FIFO is autmatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME                 1'b1
  // 0: FIFO flags are synchronized to CLKIN
  // 1: FIFO flags are synchronized to internal 27MHz clock

`define ADV7185_REGISTER_7 {`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET, `FIFO_RESET,
`FIFO_FLAG_MARGIN}

///////////////////////////////////////////////////////////////////////////////
// Register 8
///////////////////////////////////////////////////////////////////////////////

`define INPUT_CONTRAST_ADJUST                     8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register 9
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SATURATION_ADJUST                   8'h8C

`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register A
///////////////////////////////////////////////////////////////////////////////

`define INPUT_BRIGHTNESS_ADJUST                   8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register B
///////////////////////////////////////////////////////////////////////////////

`define INPUT_HUE_ADJUST                          8'h00

`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register C
///////////////////////////////////////////////////////////////////////////////

`define DEFAULT_VALUE_ENABLE                1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values
`define DEFAULT_VALUE_AUTOMATIC_ENABLE          1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values if lock is lost
`define DEFAULT_Y_VALUE                         6'h0C
  // Default Y value

`define ADV7185_REGISTER_C {`DEFAULT_Y_VALUE, `DEFAULT_VALUE_AUTOMATIC_ENABLE,
`DEFAULT_VALUE_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// Register D
///////////////////////////////////////////////////////////////////////////////

`define DEFAULT_CR_VALUE                        4'h8
  // Most-significant four bits of default Cr value
`define DEFAULT_CB_VALUE                        4'h8
  // Most-significant four bits of default Cb value

`define ADV7185_REGISTER_D {`DEFAULT_CB_VALUE, `DEFAULT_CR_VALUE}

///////////////////////////////////////////////////////////////////////////////
// Register E
///////////////////////////////////////////////////////////////////////////////

`define TEMPORAL_DECIMATION_ENABLE              1'b0
  // 0: Disable
  // 1: Enable
`define TEMPORAL_DECIMATION_CONTROL             2'h0
  // 0: Supress frames, start with even field
  // 1: Supress frames, start with odd field
  // 2: Supress even fields only
  // 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE                4'h0
  // 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE,
`TEMPORAL_DECIMATION_CONTROL, `TEMPORAL_DECIMATION_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// Register F
///////////////////////////////////////////////////////////////////////////////

`define POWER_SAVE_CONTROL                      2'h0
  // 0: Full operation
  // 1: CVBS only
  // 2: Digital only
  // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY              1'b0
  // 0: Power-down pin has priority
  // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                    1'b0
  // 0: Reference is functional
  // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR                1'b0
  // 0: LLC generator is functional
  // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                         1'b0
  // 0: Chip is functional
  // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                        1'b0
  // 0: Normal operation
  // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP                              1'b0
  // 0: Normal operation
  // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP,
`POWER_DOWN_LLC_GENERATOR, `POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY,
`POWER_SAVE_CONTROL}
```

```
///////////////////////////////////////////////////////////////////////////////
// Register 33
///////////////////////////////////////////////////////////////////////////////

`define PEAK_WHITE_UPDATE                      1'b1
  // 0: Update gain once per line
  // 1: Update gain once per field
`define AVERAGE_BIRIGHTNESS_LINES              1'b1
  // 0: Use lines 33 to 310
  // 1: Use lines 33 to 270
`define MAXIMUM_IRE                            3'h0
  // 0: PAL: 133, NTSC: 122
  // 1: PAL: 125, NTSC: 115
  // 2: PAL: 120, NTSC: 110
  // 3: PAL: 115, NTSC: 105
  // 4: PAL: 110, NTSC: 100
  // 5: PAL: 105, NTSC: 100
  // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL                             1'b1
  // 0: Disable color kill
  // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRE,
`AVERAGE_BIRIGHTNESS_LINES, `PEAK_WHITE_UPDATE}

`define ADV7185_REGISTER_10 8'h00
`define ADV7185_REGISTER_11 8'h00
`define ADV7185_REGISTER_12 8'h00
`define ADV7185_REGISTER_13 8'h45
`define ADV7185_REGISTER_14 8'h18
`define ADV7185_REGISTER_15 8'h60
`define ADV7185_REGISTER_16 8'h00
`define ADV7185_REGISTER_17 8'h01
`define ADV7185_REGISTER_18 8'h00
`define ADV7185_REGISTER_19 8'h10
`define ADV7185_REGISTER_1A 8'h10
`define ADV7185_REGISTER_1B 8'hF0
`define ADV7185_REGISTER_1C 8'h16
`define ADV7185_REGISTER_1D 8'h01
`define ADV7185_REGISTER_1E 8'h00
`define ADV7185_REGISTER_1F 8'h3D
`define ADV7185_REGISTER_20 8'hD0
`define ADV7185_REGISTER_21 8'h09
`define ADV7185_REGISTER_22 8'h8C
`define ADV7185_REGISTER_23 8'hE2
`define ADV7185_REGISTER_24 8'h1F
`define ADV7185_REGISTER_25 8'h07
`define ADV7185_REGISTER_26 8'hC2
`define ADV7185_REGISTER_27 8'h58
`define ADV7185_REGISTER_28 8'h3C
`define ADV7185_REGISTER_29 8'h00
`define ADV7185_REGISTER_2A 8'h00
`define ADV7185_REGISTER_2B 8'hA0
`define ADV7185_REGISTER_2C 8'hCE
`define ADV7185_REGISTER_2D 8'hF0
`define ADV7185_REGISTER_2E 8'h00
`define ADV7185_REGISTER_2F 8'hF0
`define ADV7185_REGISTER_30 8'h00
`define ADV7185_REGISTER_31 8'h70
`define ADV7185_REGISTER_32 8'h00
`define ADV7185_REGISTER_34 8'h0F
`define ADV7185_REGISTER_35 8'h01
`define ADV7185_REGISTER_36 8'h00
`define ADV7185_REGISTER_37 8'h00
`define ADV7185_REGISTER_38 8'h00
`define ADV7185_REGISTER_39 8'h00
`define ADV7185_REGISTER_3A 8'h00
`define ADV7185_REGISTER_3B 8'h00

`define ADV7185_REGISTER_44 8'h41
`define ADV7185_REGISTER_45 8'hBB
```

```
`define ADV7185_REGISTER_F1 8'hEF
`define ADV7185_REGISTER_F2 8'h80


module adv7185init (reset, clock_27mhz, source, tv_in_reset_b,
                   tv_in_i2c_clock, tv_in_i2c_data);

   input reset;
   input clock_27mhz;
   output tv_in_reset_b; // Reset signal to ADV7185
   output tv_in_i2c_clock; // I2C clock output to ADV7185
   output tv_in_i2c_data; // I2C data line to ADV7185
   input source; // 0: composite, 1: s-video

   initial begin
      $display("ADV7185 Initialization values:");
      $display("  Register 0:  0x%X", `ADV7185_REGISTER_0);
      $display("  Register 1:  0x%X", `ADV7185_REGISTER_1);
      $display("  Register 2:  0x%X", `ADV7185_REGISTER_2);
      $display("  Register 3:  0x%X", `ADV7185_REGISTER_3);
      $display("  Register 4:  0x%X", `ADV7185_REGISTER_4);
      $display("  Register 5:  0x%X", `ADV7185_REGISTER_5);
      $display("  Register 7:  0x%X", `ADV7185_REGISTER_7);
      $display("  Register 8:  0x%X", `ADV7185_REGISTER_8);
      $display("  Register 9:  0x%X", `ADV7185_REGISTER_9);
      $display("  Register A:  0x%X", `ADV7185_REGISTER_A);
      $display("  Register B:  0x%X", `ADV7185_REGISTER_B);
      $display("  Register C:  0x%X", `ADV7185_REGISTER_C);
      $display("  Register D:  0x%X", `ADV7185_REGISTER_D);
      $display("  Register E:  0x%X", `ADV7185_REGISTER_E);
      $display("  Register F:  0x%X", `ADV7185_REGISTER_F);
      $display("  Register 33: 0x%X", `ADV7185_REGISTER_33);
   end

   //
   // Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)
   //

   reg [7:0] clk_div_count, reset_count;
   reg clock_slow;
   wire reset_slow;

   initial
     begin
        clk_div_count <= 8'h00;
        // synthesis attribute init of clk_div_count is "00";
        clock_slow <= 1'b0;
        // synthesis attribute init of clock_slow is "0";
     end

   always @(posedge clock_27mhz)
     if (clk_div_count == 26)
       begin
          clock_slow <= ~clock_slow;
          clk_div_count <= 0;
       end
     else
       clk_div_count <= clk_div_count+1;

   always @(posedge clock_27mhz)
     if (reset)
       reset_count <= 100;
     else
       reset_count <= (reset_count==0) ? 0 : reset_count-1;

   assign reset_slow = reset_count != 0;

   //
   // I2C driver
   //
```

```
reg load;
reg [7:0] data;
wire ack, idle;

i2c i2c(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
        .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
        .sda(tv_in_i2c_data));

//
// State machine
//

reg [7:0] state;
reg tv_in_reset_b;
reg old_source;

always @(posedge clock_slow)
   if (reset_slow)
     begin
        state <= 0;
        load <= 0;
        tv_in_reset_b <= 0;
        old_source <= 0;
     end
   else
     case (state)
       8'h00:
         begin
            // Assert reset
            load <= 1'b0;
            tv_in_reset_b <= 1'b0;
            if (!ack)
              state <= state+1;
         end
       8'h01:
         state <= state+1;
       8'h02:
         begin
            // Release reset
            tv_in_reset_b <= 1'b1;
            state <= state+1;
                 end
       8'h03:
         begin
            // Send ADV7185 address
            data <= 8'h8A;
            load <= 1'b1;
            if (ack)
              state <= state+1;
         end
       8'h04:
         begin
            // Send subaddress of first register
            data <= 8'h00;
            if (ack)
              state <= state+1;
         end
       8'h05:
         begin
            // Write to register 0
            data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
            if (ack)
              state <= state+1;
         end
       8'h06:
         begin
            // Write to register 1
            data <= `ADV7185_REGISTER_1;
            if (ack)
              state <= state+1;
```

```
  end
8'h07:
  begin
     // Write to register 2
     data <= `ADV7185_REGISTER_2;
     if (ack)
       state <= state+1;
  end
8'h08:
  begin
     // Write to register 3
     data <= `ADV7185_REGISTER_3;
     if (ack)
       state <= state+1;
  end
8'h09:
  begin
     // Write to register 4
     data <= `ADV7185_REGISTER_4;
     if (ack)
       state <= state+1;
  end
8'h0A:
  begin
     // Write to register 5
     data <= `ADV7185_REGISTER_5;
     if (ack)
       state <= state+1;
  end
8'h0B:
  begin
     // Write to register 6
     data <= 8'h00; // Reserved register, write all zeros
     if (ack)
       state <= state+1;
  end
8'h0C:
  begin
     // Write to register 7
     data <= `ADV7185_REGISTER_7;
     if (ack)
       state <= state+1;
  end
8'h0D:
  begin
     // Write to register 8
     data <= `ADV7185_REGISTER_8;
     if (ack)
       state <= state+1;
  end
8'h0E:
  begin
     // Write to register 9
     data <= `ADV7185_REGISTER_9;
     if (ack)
       state <= state+1;
  end
8'h0F: begin
   // Write to register A
   data <= `ADV7185_REGISTER_A;
 if (ack)
   state <= state+1;
end
8'h10:
  begin
     // Write to register B
     data <= `ADV7185_REGISTER_B;
     if (ack)
       state <= state+1;
  end
8'h11:
  begin
     // Write to register C
     data <= `ADV7185_REGISTER_C;
     if (ack)
      state <= state+1;
  end
8'h12:
  begin
     // Write to register D
     data <= `ADV7185_REGISTER_D;
     if (ack)
      state <= state+1;
  end
8'h13:
  begin
     // Write to register E
     data <= `ADV7185_REGISTER_E;
     if (ack)
      state <= state+1;
  end
8'h14:
  begin
     // Write to register F
     data <= `ADV7185_REGISTER_F;
     if (ack)
      state <= state+1;
  end
8'h15:
  begin
     // Wait for I2C transmitter to finish
     load <= 1'b0;
     if (idle)
      state <= state+1;
  end
8'h16:
  begin
     // Write address
     data <= 8'h8A;
     load <= 1'b1;
     if (ack)
      state <= state+1;
  end
8'h17:
  begin
     data <= 8'h33;
     if (ack)
      state <= state+1;
  end
8'h18:
  begin
     data <= `ADV7185_REGISTER_33;
     if (ack)
      state <= state+1;
  end
8'h19:
  begin
     load <= 1'b0;
     if (idle)
      state <= state+1;
  end

8'h1A: begin
   data <= 8'h8A;
   load <= 1'b1;
   if (ack)
     state <= state+1;
end
8'h1B:
  begin
     data <= 8'h33;
     if (ack)
```

```
                 state <= state+1;
             end
           8'h1C:
             begin
                load <= 1'b0;
                if (idle)
                  state <= state+1;
             end
           8'h1D:
             begin
                load <= 1'b1;
                data <= 8'h8B;
                if (ack)
                  state <= state+1;
             end
           8'h1E:
             begin
                data <= 8'hFF;
                if (ack)
                  state <= state+1;
             end
           8'h1F:
             begin
                load <= 1'b0;
                if (idle)
                  state <= state+1;
             end
           8'h20:
             begin
                // Idle
                if (old_source != source) state <= state+1;
                old_source <= source;
             end
           8'h21: begin
              // Send ADV7185 address
              data <= 8'h8A;
              load <= 1'b1;
              if (ack) state <= state+1;
           end
           8'h22: begin
              // Send subaddress of register 0
              data <= 8'h00;
              if (ack) state <= state+1;
           end
           8'h23: begin
              // Write to register 0
              data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
              if (ack) state <= state+1;
           end
           8'h24: begin
              // Wait for I2C transmitter to finish
              load <= 1'b0;
              if (idle) state <= 8'h20;
           end
         endcase

endmodule

// i2c module for use with the ADV7185

module i2c (reset, clock4x, data, load, idle, ack, scl, sda);

   input reset;
   input clock4x;
   input [7:0] data;
   input load;
   output ack;
   output idle;
   output scl;
   output sda;
```

```
reg [7:0] ldata;
reg ack, idle;
reg scl;
reg sdai;

reg [7:0] state;

assign sda = sdai ? 1'bZ : 1'b0;

always @(posedge clock4x)
  if (reset)
    begin
      state <= 0;
      ack <= 0;
    end
  else
    case (state)
      8'h00: // idle
        begin
          scl <= 1'b1;
          sdai <= 1'b1;
          ack <= 1'b0;
          idle <= 1'b1;
          if (load)
            begin
              ldata <= data;
              ack <= 1'b1;
              state <= state+1;
            end
        end
      8'h01: // Start
        begin
          ack <= 1'b0;
          idle <= 1'b0;
          sdai <= 1'b0;
          state <= state+1;
        end
      8'h02:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h03: // Send bit 7
        begin
          ack <= 1'b0;
          sdai <= ldata[7];
          state <= state+1;
        end
      8'h04:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h05:
        begin
          state <= state+1;
        end
      8'h06:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h07:
        begin
          sdai <= ldata[6];
          state <= state+1;
        end
      8'h08:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h09:
        begin
          state <= state+1;
        end
      8'h0A:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h0B:
        begin
          sdai <= ldata[5];
          state <= state+1;
        end
      8'h0C:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h0D:
        begin
          state <= state+1;
        end
      8'h0E:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h0F:
        begin
          sdai <= ldata[4];
          state <= state+1;
        end
      8'h10:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h11:
        begin
          state <= state+1;
        end
      8'h12:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h13:
        begin
          sdai <= ldata[3];
          state <= state+1;
        end
      8'h14:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h15:
        begin
          state <= state+1;
        end
      8'h16:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h17:
        begin
          sdai <= ldata[2];
          state <= state+1;
        end
      8'h18:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h19:
        begin
          state <= state+1;
        end
      8'h1A:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h1B:
        begin
          sdai <= ldata[1];
          state <= state+1;
        end
      8'h1C:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h1D:
        begin
          state <= state+1;
        end
      8'h1E:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h1F:
        begin
          sdai <= ldata[0];
          state <= state+1;
        end
      8'h20:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h21:
        begin
          state <= state+1;
        end
      8'h22:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h23: // Acknowledge bit
        begin
          state <= state+1;
        end
      8'h24:
        begin
          scl <= 1'b1;
          state <= state+1;
        end
      8'h25:
        begin
          state <= state+1;
        end
      8'h26:
        begin
          scl <= 1'b0;
          if (load)
            begin
```

```
                  ldata <= data;
                  ack <= 1'b1;
                  state <= 3;
               end
             else
               state <= state+1;
          end
        8'h27:
          begin
             sdai <= 1'b0;
             state <= state+1;
          end
        8'h28:
          begin
             scl <= 1'b1;
             state <= state+1;
          end
        8'h29:
          begin
             sdai <= 1'b1;
             state <= 0;
          end
       endcase

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Wake-up Module
//
// File:   wakeup.v
// Last Updated:   December 13, 2005
// Author: Aaron Stonely
//
// This module extends the clock module in order to implement the behavior of
// a digital alarm clock.
//
////////////////////////////////////////////////////////////////////////////////


module wakeup(clk,reset,wake_ctrl,numbers,thetime,one_min_enable,burglary,time_selector,
                                buzzer,done,alarm,wake_state,wake_time,light_time);

input [15:0] thetime;
input clk, reset, one_min_enable, burglary, time_selector;
input [3:0] numbers;
input [2:0] wake_ctrl;
output [15:0] wake_time, light_time;
output [3:0] wake_state;
output buzzer, done, alarm;

// One_hz_enable declared to be used for snooze sequence.
wire one_hz_enable;
one_hz_enable seconds(clk,reset,one_hz_enable);

// This selector allows the user to select whether the system keeps time
// in seconds or minutes, with the former used for demonstration
// purposes.
wire time_enable = time_selector ? one_hz_enable : one_min_enable;

wire [2:0] wake_control = burglary ? 3'h0 : wake_ctrl;

reg [15:0] wake_time=0;
reg [3:0] wake_state=0, snooze_count=0;
reg alarm=0, done=0, buzzer=0;

// Below the states of the wakeup finite state machine are
// parameterized.

parameter S_0000 = 0; // Reset/Off
parameter S_0001 = 1; // Program 1st number
parameter S_0010 = 2; // Program 2nd number
parameter S_0011 = 3; // Program 3rd number
parameter S_0100 = 4; // Program 4th number
parameter S_0101 = 5; // Alarm on
parameter S_0110 = 6; // Wake-up sequence begin
parameter S_0111 = 7; // Wake-up Active
parameter S_1000= 8;  // Snooze

// Below the keyboard control commands are parameterized.
parameter no_command = 3'h0;
parameter alarm_on = 3'h1;
parameter alarm_off = 3'h2;
parameter snooze = 3'h3;
parameter set_wake = 3'h4;

// Below is a statement used to determine if the keyboard has requested
// to set the time, or if the keyboard is sending a valid number to reprogram
// the wakeup time.
wire set_time = (((wake_ctrl == set_wake) ||
                       ((wake_state == S_0001) && (numbers < 4'h3)) ||
                       ((wake_state == S_0010) && (numbers < 4'hA) && (wake_time[15:12] <
2)) ||
                       ((wake_state == S_0010) && (numbers < 4'h4) && (wake_time[15:12]
== 2)) ||
                       ((wake_state == S_0011) && (numbers < 4'h6)) ||
                       ((wake_state == S_0100) && (numbers < 4'hA))) &&
                       !burglary);

// Below are several statements used to calculate the lighting time
// (15 minutes before the wake-up time).
wire [3:0] wake_min1 = ((wake_time << 12) >> 12);
wire [3:0] wake_min2 = ((wake_time << 8) >> 12);
wire [3:0] wake_hour1 = ((wake_time << 4) >> 12);
wire [3:0] wake_hour2 = (wake_time >> 12);
wire [3:0] min1_plus5 = (wake_min1 + 5);
wire [3:0] min1_minus5 = (wake_min1 - 5);
wire [3:0] min2_minus2 = (wake_min2 - 2);
wire [3:0] min2_minus1 = (wake_min2 - 1);
wire [3:0] min2_plus4 = (wake_min2 + 4);
wire [3:0] hour2_minus1 = (wake_hour2 - 1);
wire [3:0] hour1_minus1 = (wake_hour1 - 1);

// Below, the lighting time is calculated.
wire [15:0] light_time =
                                ((wake_min1 > 4) ?
                                     (wake_min2 == 0) ?
                                     (wake_hour1 == 0) ?
                                     (wake_hour2 == 0) ?
                                     {12'h235,min1_minus5} :
                                     {hour2_minus1,8'h95,min1_minus5} :
                                     {wake_hour2,hour1_minus1,4'h5,min1_minus5} :
                                     {wake_hour2,wake_hour1,min2_minus1,min1_minus5} :
                                     (wake_min2 < 2) ?
                                     (wake_hour1 == 0) ?
                                     (wake_hour2 == 0) ?
                                     {8'h23,min2_plus4,min1_plus5} :
                                     {hour2_minus1,4'h9,min2_plus4,min1_plus5} :
                                     {wake_hour2,hour1_minus1,min2_plus4,min1_plus5} :
                                     {wake_hour2,wake_hour1,min2_minus2,min1_plus5});

// Below is the Finite state machine used by the wakeup alarm in order
// to control the state transitions and behavior of the wakeup alarm.
// The alarm signal is used to trigger a wakeup sequence in the lighting
// module. The done signal is used to signal the keyboard that a program
// has been completed.

always @ (posedge clk)
begin
    if(reset | burglary)
    begin
```

```
        wake_state <= 0;
        wake_time <= burglary ? wake_time : 0;
        alarm <= 0;
        snooze_count <= 0;
          done <= 0;
    end
    else
    case(wake_state)
        S_0000: // Reset/Off
        begin
                             // Transition to another state when the user requests
                             // to turn the alarm on or to set the wake-up time.
           wake_state <= (wake_control == alarm_on) ? S_0101 :
                         set_time ? S_0001 :
                         wake_state;
            wake_time <= set_time ? 0 : wake_time;
                             alarm <= 0;
                             done <= 0;
        end

                 // States S_0001 through S_0100 transition to the next state
                 // once a valid number is received from the keyboard. After
                 // a valid number has been received in S_0100, the FSM
                 // returns to the alarm off state.

        S_0001: // Program 1st number
        begin
            wake_state <= set_time ? S_0010 :
                           wake_state;
            wake_time <= set_time ?
                          ({numbers,12'h000} + wake_time) :
                          wake_time;
        end
        S_0010: // Program 2nd number
        begin
            wake_state <= set_time ? S_0011 :
                           wake_state;
            wake_time <= set_time ?
                          ({4'h0,numbers,8'h00} + wake_time) :
                          wake_time;
        end
        S_0011: // Program 3rd number
        begin
            wake_state <= set_time ? S_0100 :
                           wake_state;
            wake_time <= set_time ?
                          ({8'h00,numbers,4'h0} + wake_time) :
                          wake_time;
        end
        S_0100: // Program 4th number
        begin
            wake_state <= set_time ? S_0000 :
                           wake_state;
            wake_time <= set_time ?
                          ({12'h000,numbers} + wake_time) :
                          wake_time;

                 done <= set_time ? 1 : 0;
        end
        S_0101: // Alarm on
        begin
                             // A transition to the next state occurs when the current
time
                             // reaches the light_time or the system time. If it is the
former,
                             // the next state is wake-up sequence begin, it the latter,
then
                             // the next sequence is wake-up active.
                             // The user can also turn off the alarm.

            wake_state <= (wake_control == alarm_off) ? S_0000 :
```

```
                          set_time ? S_0001 :
                          ((thetime == light_time) | (thetime == wake_time)) ?
                                                                  S_0110 : wake_state;
            wake_time <= set_time ? 0 : wake_time;
            alarm <= ((thetime == light_time) |
                                                  (thetime == wake_time)) ? 1 : 0;
        end
        S_0110: // Wake-up sequence begin
        begin
                              // Once a wake-up sequence has begun, the user can turn off
                              // the alarm, snooze the alarm, or continue to sleep.
                              // A transition occurs upon input from the user, or
                              // if the current time reaches the wakeup time.

            wake_state <= (wake_control == alarm_off) ? S_0000 :
                          set_time ? S_0001 :
                          (wake_control == snooze) ? S_1000 :
                          (thetime == wake_time)  ? S_0111 :
                          wake_state;
            wake_time <= set_time ? 0 : wake_time;
            alarm <= ((wake_control == snooze) |
                                (wake_control == alarm_off)) ? 1 : 0;
            snooze_count <= 0;
        end
        S_0111: // Wake-up Active
        begin
                              // In the wake-up active state, the lights are at
                              // full intensity and the buzzer is active.
                              // The user can snooze or turn off the alarm.

            wake_state <= (wake_control == alarm_off) ? S_0000 :
                          set_time ? S_0001 :
                          (wake_control == snooze) ? S_1000 :
                          wake_state;
            wake_time <= set_time ? 0 : wake_time;
            alarm <= ((wake_control == snooze) |
                                (wake_control == alarm_off)) ? 1 : 0;
            snooze_count  <= 0;
        end
        S_1000: // Snooze (a snooze lasts 15 time units)
        begin
                              // The system will exit the snooze state after 15 time
                              // units have elapsed, or if the user turns the alarm
                              // off.

            wake_state <= (wake_control == alarm_off) ? S_0000 :
                          set_time ? S_0001 :
                          (snooze_count == 15) ? S_0111 :
                          wake_state;
            snooze_count <= time_enable ? (snooze_count + 1) :
                            snooze_count;
            wake_time <= set_time ? 0 : wake_time;
            alarm <= (snooze_count == 15) ? 1 : 0;
        end
        default:
        begin
            wake_state <= 0;
            wake_time <= 0;
            alarm <= 0;
            snooze_count <= 0;
                 done <= 0;
        end
    endcase
end

// The wake-up alarm buzzes the buzzer every 2 seconds for
// a duration of 1 second.
always @ (posedge clk)
begin
       buzzer <= (wake_state == S_0111) ?
                       one_hz_enable ? !buzzer : buzzer :
                        0;
end

endmodule


//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//
// ****** NOTE ******
// This file was not actually used in our project demonstration since we
// didn't get the zbt working by demonstration time.
//
/////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.

module zbt_6111(clk, cen, we, addr, write_data, read_data,
                 ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk;                 // system clock
   input cen;                 // clock enable for gating ZBT cycles
   input we;                  // write enable (active HIGH)
   input [18:0] addr;         // memory address
   input [35:0] write_data;   // data to write
   output [35:0] read_data;   // data read from memory
   output       ram_clk;      // physical line to ram clock
   output       ram_we_b;     // physical line to ram we_b
   output [18:0] ram_address; // physical line to ram address
   inout [35:0]  ram_data;    // physical line to ram data
   output       ram_cen_b;    // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire         ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                   // times if its clk edges equal FPGA's
                                   // so we clock it on the falling edges
                                   // and thus let data stabilize longer
   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
   assign      read_data = ram_data;

endmodule // zbt_6111
```