A Bomb Controller
Microprocessor-Based Systems (E155)
Final Project Report

Eric Angell
<eoa@cs.hmc.edu>

Matt Livianu
<mlivianu@cs.hmc.edu>

December 15, 2003

Abstract

A bomb controller integrates several different interesting systems, including timing and tamper protection. We present a controller implemented in an easy-to-carry metal toolbox with a cost of approximately $100 and 100 hours.

Contents

List of Figures

1 Overview

We designed and built a bomb controller using the PIC 18F452 Micro-controller and Xilinx Spartan XCS10-3PC84C FPGA on the E155 utility board (SMPS 1.0). Our controller features the following components:

- 4 seven-segment displays
- 4x4 matrix keypad
- Customizable security code
- User-defined countdown timer
- Tamper protection
- Bomb detonation output pin/indicator LED
- Piezoelectric buzzer to indicate both key presses and detonation

The FPGA handles the I/O, polling the keypad for input and acting as the display driver for the seven-segment displays. The micro-controller acts as the brain, handling the security code, countdown timer, buzzer, output LED, and tamper switch. The two chips communicate via Serial Peripheral Interface (SPI) with the PIC as the master device.

2 Background

Much of the logic in a bomb controller has a variety of other uses. Such a controller is essentially a combination of alarm clock, keypad security system, tamper detection system, and external event triggering system. In brainstorming our project proposal, we entertained the idea of an alarm clock by itself, but we wanted to create something with more functionality; ideal would be a device that could not be readily purchased and could potentially be easily leveraged to create other interesting applications. With relatively minor modifications, portions of our controller can be adapted to become an electronic keypad-based door lock, door alarm system, light timer, alarm clock, or several other similar devices.

It should be stressed that this project has no malicious intent and is simply an interesting academic exercise. While it is theoretically possible to trigger an explosive device with our controller, neither of us can support this as a wise idea. Regardless of its functionality, the stability of our assembly code is somewhat questionable, at least as regards actual use controlling explosives.

3 Implementation

Both the PIC assembly code and the FPGA Verilog represent significant amounts of effort, and significant complexity. We will examine here in detail the implementation of the project in its final form, as demonstrated in class.

3.1 PIC Micro-controller

Being that both of us are receiving degrees in computer science rather than engineering, we have several years of high-level coding under our belts, but only a moderate amount of assembly. As such, we find it more natural to think at a high level, and our code reflects that. While we respect the need to define our own calling conventions, we exhibit what is perhaps a gratuitous use of functions for PIC assembly. While it may have made sense to implement our code in C rather than assembly, we believe that the overhead involved in learning how to use and debug C in the MPLAB IDE would have negated any of its benefits, especially considering our use of interrupts.

Our code is separated into several distinct portions to handle various states, and is mainly event driven, busy-waiting in the main loop until new input becomes available. A timer interrupt is triggered every quarter second, at which time the countdown is potentially decremented, new input is potentially received, and the display is updated. The main program loop then proceeds to act on the newly available input, busy-waiting in various functions which require more input before they can return to the default state of main.

An interrupt is also triggered when the tamper-protection circuit is disturbed, at which point the

state of the bomb is checked (armed?) before deciding to ignore the event or explode.

As with most any software, more debugging would be useful, but things appear to behave normally most of the time. The system at least functions well enough to demonstrate successfully, though it should again be stressed that this should not be treated as production-quality code.

Useful in understanding the code will be the flowchart in figure 4.

3.1.1 Initialization

When the PIC first starts running, it initializes all of the necessary registers. It first sets SCK and SDO as output so that it can send information to the FPGA, and sets SDI as input, so that it can receive data from the FPGA. It sets pin 2 on Port B as input for the FPGA interrupt request and sets pin 4 on Port B as input for the tamper protection circuit interrupt. It also sets pins 0 and 7 on port D as output for the serial acknowledge pin and the explosion indication LED output.

The PIC then initializes Timer 1 to a pre-scale of 1:4, using Fosc/4 and turns the timer on. It sets the CCP interrupt to 1/4 of a second, sets the default security code of 1-2-3-4 and clears all of the flags. The next step is to initialize the SPI as master with an Fosc/64 pre-scale setting and to indicate transmission of data on the rising edge of the clock. Next on the task list is interrupts. The Pic initializes global, peripheral, and port B interrupts only and disables priority interrupts.

Finally, it sends the FPGA the data to display "OFF" to the user.

3.1.2 Main Program Loop

Once in the main loop, the PIC checks only for a few key presses, ignoring all others. If no new key was pressed, main busy-waits. If the main loop finds that the newinput flag has been set, then it checks the contents of the inputbuffer which stores new key presses. If the bomb is armed, the PIC only checks if the button pressed was cancel. Otherwise, the PIC checks to see if the button pressed was "cancel", "code", or "arm" and branches to the appropriate operation routine.

3.1.3 Timer1/CCP1

Originally, Timer 1 counted up 1 second. Every second, the CCP would compare positively to the timer and would trigger an interrupt, at which point the timer would be reset and other actions taken if appropriate. We originally had two interrupts, however: one interrupt for the input, and one for the one-second timer. We had problems with unwanted key bounces with the keypad we used when the user pressed a key. Our solution was to eliminate the input interrupt. We decreased the timer interrupt wait time to 1/4 second, and added a state machine to record which quarter second the PIC was in. This solution obviated the need for the second interrupt, because we simply dealt with input requests in the 1/4 second interrupt loop along with any other appropriate actions. The 1/4 second timer loop solved our key bounce problem, as the user could only press a maximum of 4 keys per second, rather than an undetermined number of key presses which often resulted in unwanted input.

3.1.4 Interrupts

When the timer counts up to 1/4 of a second, a counter interrupt occurs. In this interrupt, the timers are cleared, the timerstate is incremented, or reset if the process was in the last state, and the display function is called to send the contents of DISPLAYHI and DISIPLAYLO to the FPGA. If there is input waiting at the FPGA to be sent, then the PIC goes to the input interrupt function to receive that input. The contents of the display registers are determined by various flags. If the bomb armed flag is set, then the timer is decremented and its value is moved into the display registers. If there is less than one minute left on the timer, then the display will be sent blank or the counter value depending on the current state. If the bomb is armed and the can-

cel button is pressed, the display will alternate between the time and "CODE". Furthermore, if the old code flag is set, then the PIC alternates between "olD" and "CODE". Similarly, if the code flag is set, then "CODE" is moved into the display registers to be sent to the FPGA. After checking the relevant flags, the PIC sends the data to the FPGA over SPI and then checks if the bomb has reached zero in its count, in which case it branches to the explode operation.

The second interrupt which can occur is on Port B and is triggered by the tamper protection circuit on the bomb case. In the event of the bomb case opening while the bomb is armed, a connection is broken and the circuit triggers an interrupt. The interrupt handler decides which interrupt occurred and if it was the tamper protection circuit, then the bomb explodes. Unfortunately, since the bomb protection circuit causes an interrupt to occur, the explosion also occurs within the interrupt and so we could not implement a 'reset' key as we had desired.

3.1.5 Serial Peripheral Interface

All data between the PIC and the FPGA is sent over the serial peripheral interface. Since the PIC is the master of the SPI, the FPGA must wait until the PIC acknowledges a request to sent data before actually sending data. In order to facilitate the sending of data over SPI both from the PIC to FPGA and FPGA to PIC, a sendssp function performs the sending of data. The sendssp function starts sending the first byte and then busy-waits until the data is done sending, indicated by the buffer full (SSPSTAT, BF) flag being raised. The function then sends the second byte of data and returns.

The input interrupt routine is not really a separate interrupt, but is called from the counter interrupt. In this routine, the serial acknowledge(SAC) pin is raised high, indicating to the FPGA that the PIC is ready to receive the key press data. The PIC then sends dummy data to the FPGA and receives the key press over SPI, which it saves in the inputbuffer. The input interrupt routine then lowers SAC and returns to the counter interrupt routine.

3.1.6 Main Program Functions

The op functions perform the major functions for the PIC such as getting user input, setting the appropriate flags, and generally running the bomb controller. There are as many operation functions as there are special operation keys on the keypad, and this report discusses each of them in turn.

The cancel operation checks to see if the bomb is armed. If it is not, it branches back to main, ignoring the key press. If the bomb is armed, then it sets the cancel request flag and waits for user input of the security code. After obtaining the user input, it lets the nextaction routine take care of the rest.

The arm bomb operation checks to see if the arm flag is set. If it is, then it starts the countdown timer. Otherwise, it sets the time flag, displays "CODE", and gets the user input of the security code. After obtaining the user input, it lets the nextaction routine take care of the rest.

The set code operation sets the old code flag and gets the user input of the old code. After obtaining the old code input from the user, it branches to the nextaction routine.

The explode operation clears the armed flag, turns on the explode red LED, turns on the piezoelectric buzzer and then loops indefinitely. The PIC must be reset in order to reset the bomb and exit from this loop.

3.1.7 Miscellaneous Functions

Aside from the main loop, interrupts, and operation functions, there are many special functions which are crucial to the correct operation of the bomb controller.

The getuserinput function busy-waits until the new input flag is set by the interrupt routine. It then checks to see if the input was "return", "cancel", any other button. If the input is "enter", then the routine returns with the enter flag set. If input is "cancel", then the routine returns with the return flag cleared. Otherwise, if the input was a number, calls the inputshiftreg

routine to input the new key press into the shift register and loops back to wait for more input.

The nextaction function serves the purpose of deciding what to do after the user has inputted a code, whether or not the code is correct. If the bomb has been armed and the code is correct then the bomb is off. If the code is wrong, the system simply continues to display the decreasing countdown timer. If the time flag is set and the code is correct, then the PIC branches to the entertime function, which prompts the user for a time and then sets that new time in the counter. If the time flag is set and the code is wrong, then the bomb is off. Otherwise, the code flag must be set, and so if the code is correct, the PIC branches to the enternewcode routine to request a new security code from the user.

Another important function in the operation of the bomb controller is the dectimer function that decrements the timer by one each time it is called. The interesting feature is that it takes hexadecimal numbers and turns them into decimal numbers as it counts down. The only time a hex number is shown is when the user first inputs the timer. At that point, the user can input a time with up to 99 minutes and 99 seconds, even though there are not 99 seconds in a minute. As soon as the seconds timer reaches zero, it loops back to 59. the dectimer function first checks to see if decrementing the timer causes it to produce a negative number. If so, then the minutes must have decreased by one and so the minutes are decreased and the seconds are wrapped to 59. If the low counter has not reached zero, then only the low counter has a chance at wrapping.

Other miscellaneous functions include the clear-all function, which clears all of the flags, buffers, and the sound, and tells the bomb to display "off". Finally, the many display functions simply serve to move the correct hexadecimal values into the DISPLAYHI and DISPLAYLO registers for transmission to the FPGA.

3.2 FPGA

The FPGA is the input/output driver for the PIC. Its two main functions are to output information on the four displays and input information from the keyboard. To accomplish these tasks it must support its third task, which is bidirectional communication with the PIC. As the comments in the main module describe, the overview of the system is fairly simple.

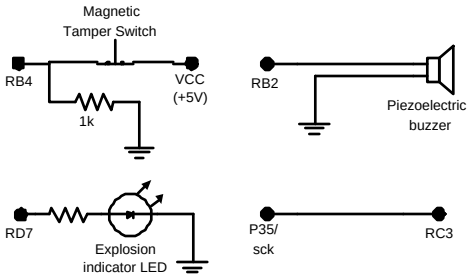

Figure 1: The magnetic tamper switch, piezoelectric buzzer, explosion LED, and serial clock from RC3 to P35. Not shown is FPGA reset tied to P26, one of the otherwise unused dip switches on the board (and grounded when open).

First we slow the clock since we don't need to be running at 1 MHz and we don't want our display to flicker. We poll the keypad continually to observe key presses, and we store the most recent one in an enabled flip-flop. We also store in a flip-flop the fact that there is a new key press which should be sent to the PIC; that flop is part of the bombspi module which handles all the communication with the PIC. We synchronize the digits being returned from the SPI logic because that logic runs on the serial clock provided by the PIC. After that synchronization, the digits go into yet more enabled flip-flops, and the display module takes the output from those flops to constantly keep the display updated.

3.2.1 Keypad

The keypadpoll module is pretty much the same as what Eric created for lab 4, although the names of some of the buttons have been changed. We push one column high at a time, and check to see if that causes any of the rows to go high (if so, the button connecting that row with that column is being pushed). When we get a but-

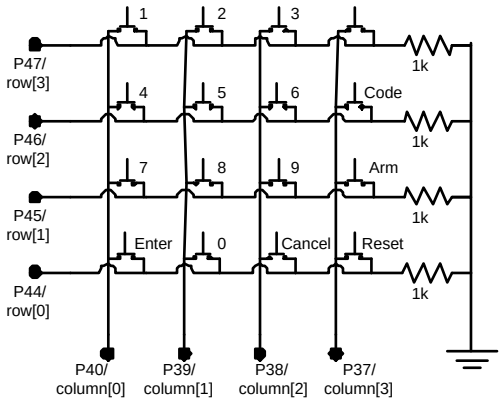

Figure 2: The matrix keypad has a ribbon cable running back to the breadboard to connect to the pins; the resistors are on the breadboard as well, and only the keypad is remote.

ton push, we pulse an enable signal high for one clock cycle to indicate that the value we are outputting at that moment corresponds to a key press. There is a fairly simple state machine that handles the polling, and a fairly straightforward always block that handles the digit output. See appendix C.2.3.

One fairly major thing we changed from lab 4 was that we switched to the other style of keypad. In lab 4, both of us had used the new red keypads, but they proved to be flaky enough both then and in our initial use of them here that we switched to one of the older keypads. The two keypads have different bouncing characteristics, but the keypadpoll module works despite those differences. This is somewhat surprising when combined with the fact that lab 4 used a 1 KHz clock and that we use a 32 KHz clock in this project (1 MHz divided by 32 rather than 2 MHz divided by 2048). The keys have not proven completely bounce-free, but it works well enough to at least demonstrate the device.

We have kept the digits 0–9 but renamed the hexadecimal keys for this project. 'A' and 'B' have become 'Enter' and 'Cancel', respectively, 'C' has been left blank and doesn't do anything, and 'D' through 'F' are 'Code', 'Arm', and 'Reset'.

The circuit to run the keypad (fig. 2) is fairly simple. Since we expect the rows to be low unless they are pulled high by a key press, we must ensure that they do not float when no key is being held. Hence we tie each row to ground through a resistor as well as connecting it to a pin on the utility board.

3.2.2 SPI

The bombspi module (appendix C.2.2) takes the serial communication pins from the PIC and creates a two-way conversation. However, the serial clock must only have one master, so the FPGA can't drive it directly when it wants to send data. To make bidirectional communication function, we've created a Serial Request pin (srq, in the Verilog) and a Serial Acknowledge pin (sac). The request pin is pulled high by the FPGA when it has data to be sent (actually, it's the output of a flop that gets set when a button is pushed and reset after data is sent). The acknowledge pin is pulled high by the PIC when it is ready to receive said data. If sac is high and the PIC is oscillating sck, we define it to be sending garbage data while receiving good data from the FPGA. Otherwise, if sac is low, we define the PIC to be sending data to be displayed on the seven-segments. Such data always comes in pairs, as we use four bits to define each of the displays and the PIC sends data in quantities of bytes. When the FPGA sends key press data, it sends four zero bits followed by four bits defining the most recently pressed key.

The module begins with a state machine in the form of a three bit counter. Since there are eight bits to each byte the PIC sends, when this counter is zero, we know we're between bytes, and that is when we can flush the shift register to a flop or become eligible to start sending the last key press. Actually, srq can be raised at any time, but sac will only go high between bytes. The always block in bombspi handles incrementing the counter each time we see a positive edge of sck, and running the shift registers appropriately. There is also a single bit of state taken care of in the block to keep track of which digit pair the current byte represents.

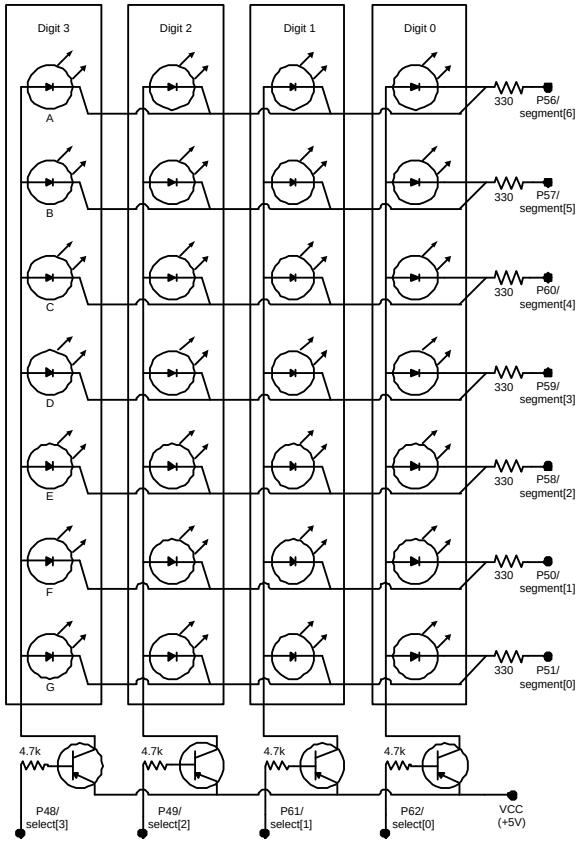

Figure 3: Four multiplexed seven-segment displays. All of the shown components are on a remote board with a ribbon cable running back to the breadboard to reach the FPGA pins on the utility board.

The assign statements which follow the always block were first included in the block, and caused nothing but trouble. Looking at the block diagram of the Verilog synthesis, we slowly determined several assignments which had to move outside the block because there were too many flip-flops in the data path, delaying the data. In its final form, however, it is pretty straightforward and works almost exactly as advertised.

3.2.3 Display

Lab 3 required us to build a pair of multiplexed seven-segment displays, but that only involved a few lines of Verilog in the main module. In order to drive our four displays in this project, we created the display module to package everything up nicely. A four-state machine drives each digit in turn for one clock cycle, running on our 32 KHz main clock. We see a small amount of bleed through with the clock that fast, but it is not very noticeable with a reasonable amount of ambient light present. We decided that the other parts of the project more warranted our time and energy so we didn't seek to improve this bleed through to surrounding digits. However, we did spend the energy to rewrite the basic seven-segment display decoder in the style of the Verilog tutorial in the lab manual, rather than leave it in the complicated logic of lab 1 (sevenseg.v, appendix C.2.5).

In the final implementation in the toolbox, the displays and accompanying resistors and transistors (see figure 3) are all mounted on a piece of perf board attached to the case. In previous labs and in development of this controller, the displays were on the breadboard right next to the utility board. However, it is significantly more usable if you can see the display from the outside, next to the keypad. All that runs from this remote board to the final breadboard is a section of ribbon cable, one pin of which connects to VCC, the rest of which connect directly to pins on the utility board.

We only need to display decimal numbers for our timer, but we wanted also to be able to prompt the user for certain input. Hence we changed our output driver so that 'A' and 'B' are instead blank and dash, respectively. The rest of the letters remain useful, so we still have 'C' through 'F', and we use '0' both as a digit and a letter.

3.3 Case

To house our device, we purchased a toolbox made out of sheet metal. With the help of a Dremel and a cordless drill, we made several cuts in the box and were successfully able to mount everything we wanted. We cut a rectangular hole in the top to fit the seven-segment displays and drilled four holes around it with which to mount the board to the inside. We had hoped that the

motherboard standoffs we had purchased would work for the displays, but they were too long so we created some standoffs out of several 4-40 nuts and washers and a few reasonably long screws. We cut another hole through which we passed the pins of the keypad, but only barely big enough to do so. Hence unscrewing the keypad will not give an attacker much more access to the innards of the case. We also drilled a hole to mount the explosion indicator LED. On the inside, the two pieces of the magnetic switch were attached with screws and adhesive, and the battery was attached with poster mounts. The breadboard and the wires were all held in place by electrical tape. The buzzer is attached to the wall of the case with a poster mount, and both the poster mount and the case have small holes in them, in an attempt to make the buzzer more audible outside. The alignment isn't perfect, but it helps anyway.

All things considered, it was significantly easier to purchase the toolbox rather than attempt to construct a container from scratch.

4 Lessons Learned

4.1 Relative Clock Speed

One of the difficulties we ran into with the SPI communication (specifically in the FPGA module) had to do with the timing of the two input clocks. The serial clock from the PIC and the main clock on the FPGA have no particular relation, especially since the serial clock only runs when there is data to be sent or received. We had initially been running the main clock slowed by a factor of 2048 since that was how Eric's lab 4 had been constructed, and it took us a while to realize that we needed the FPGA clock to be running faster than the serial clock. Since we didn't do any bidirectional SPI communication in any of the labs, we hadn't run into that issue before. Lab 6 had the FPGA implement a shift register to translate data from SPI into traffic light displays, but there it had no clock other than the serial clock from the PIC.

Since we were now trying to run a section of the FPGA on a separate clock, we discovered that the PIC was clocking SCK so quickly that the FPGA never even got to see the signals change. When we change the clock to be only slowed by a factor of 16, things worked a lot better. In the end, we settled on dividing the FPGA clock by 32 and having the PIC run the SCK at its clock divided by 64. This proved to be the major breakthrough it took us to get the data transfer from the PIC to the FPGA working correctly, and the ensuing bidirectional communication had no similar clock issues.

4.2 Tamper Protection

We had originally planned on using the A/D converter for our tamper protection circuit. The circuit would have detecting the voltage on wires running though diodes, and setting off the bomb if any of the voltages dropped too low (i.e. a wire was cut). We decided not to use this system because the magnetic circuit breaker was much more effective at tamper protection than wire cutting protection, and because we put the entire bomb into a box, we felt that we would have more problems with someone simply disconnecting the battery rather than cutting wires, and thus our protection circuit would fail. We needed a tamper protection device which would not only protect the bomb from cut wires, but also from battery disconnection. Thus, we selected the magnetic circuit as a more simplistic, but more effective solution.

5 Results

Overall, we achieved most of our original specification. From the original proposal to the midterm report to the final project, we changed some of the details of the bomb controller flow chart, modified the tamper protection circuit, and changed some of the extra output features as we dove into the nitty-gritty details of our bomb controller output. However, we met all of our goals as stated in our modified project proposal.

The most difficult part of our design process was the asynchronous clock problems we en-

countered, the bouncing keys, and the PIC code structure. We learned a great deal about synchronizing clocks and problems with communication of data from fast clocked systems to slow clocked systems. We also learned how to better diagnose problems on the oscilloscope throughout this project. The Oscilloscope proved invaluable to our success in this project, because so much of the problematic output, especially the SPI communication and bouncy key presses, could not be simulated accurately. Also, the sheer amount of PIC code that had to be written for the bomb controller really pushed us to code in higher level helper functions such that our debugging would be tractable.

Our demonstration to Professor Harris went very well, and we demonstrated each of the features of the bomb controller as well as the organization of the code. Also, the class presentation was enjoyable, even though the entire class "died" when they triggered the tamper protection switch while trying to open the armed bomb.

A Parts List

Part	Source	Vendor Part #	Price
3-20VDC Buzzer	Radioshack	273-0059	$2.99
PC board	Radioshack	276-0149	$1.69
Heatsink (fits 7805C)	Radioshack	276-1368	$1.59
Heatsink Compound	Radioshack	276-1372	$1.99
Magnetic Switch	Radioshack	490-0497	$4.99
Female Spade Terminals	Radioshack	640-4039	$1.69
4-40 Screws, Nuts, Washers	Radioshack	640-30{11, 18, 22}	3 @ $1.99
Male DB25/Ribbon Connector	MarVac	IDC-25MA-P	$5.04
16” Metal Tool Tray	Lowe’s	132774	$12.96
12V SLA Battery	Old UPS	Panasonic UP-RW1245P1	
Ribbon Cable	Old Computer		

B PIC Behavior

B.1 Program Flow

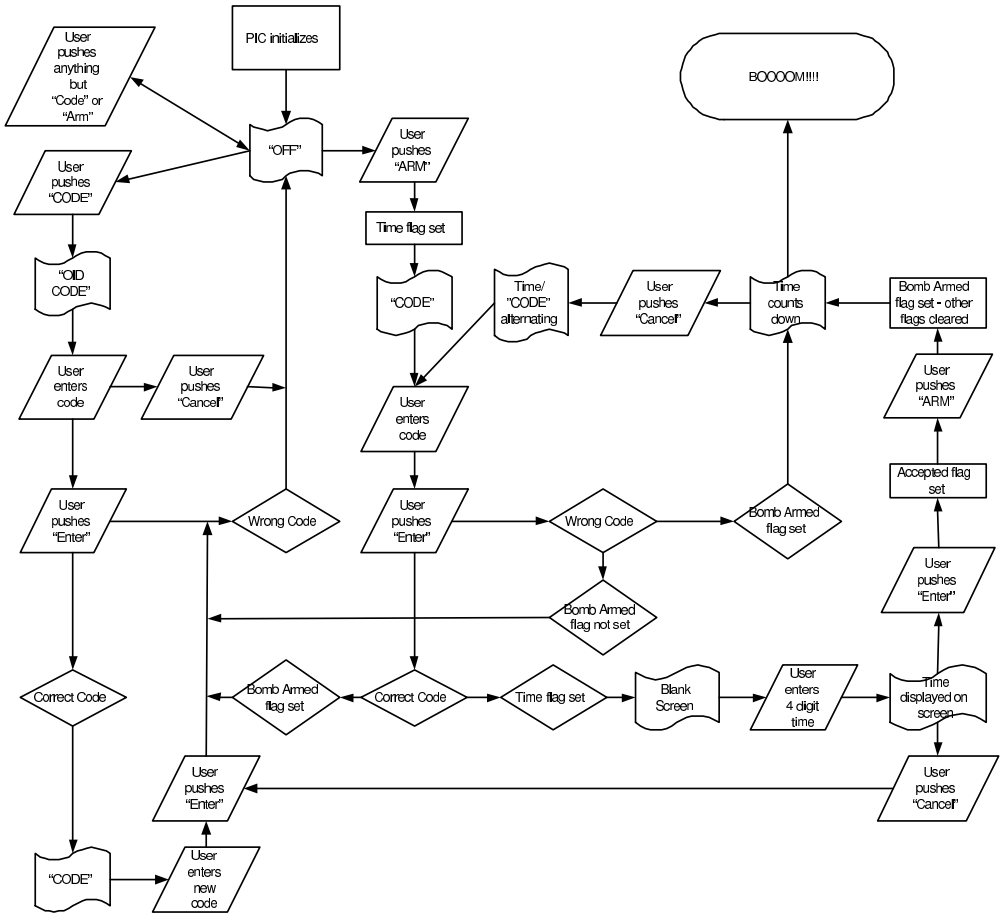

Figure 4: Flowchart describing execution path of main program loop in the PIC assembly

B.2 Assembly Code

We developed our code in raw assembly for our PIC 18F452 Micro-controller, and used Microchip's MPLAB development environment to compile the code and program the chip. Our code as of the demonstration follows, and though it has been slightly edited for formatting and readability, no instructions have been changed.

```
; bomb.asm
; written 10/26/2003 by Matt_Livianu@hmc.edu and Eric_Angell@hmc.edu
; bomb controller
; $Id: bomb.asm,v 1.11 2003/12/15 07:55:53 eric Exp $
;   - accepts keypad input
;   - talks to FPGA over SPI
;   - keeper of bomb security code

; Use the 18F452 PIC microprocessor
    LIST p=18F452
    include "p18f452.inc"

; variables (uppercase)
INPUT_TEST  EQU 0x12
; time remaining till boom (in seconds)
COUNTHI     EQU 0x20
COUNTLO     EQU 0x21
; display registers
DISPLAYHI   EQU 0x23
DISPLAYLO   EQU 0x24

; state register to count seconds
TIMERSTATE  EQU 0x27

INPUTBUF    EQU 0x30                ; input from FPGA

; input buffer
BUF1        EQU 0x31
BUF2        EQU 0x32
BUF3        EQU 0x33
BUF4        EQU 0x34

; security code register
SEC1        EQU 0x35
SEC2        EQU 0x36
SEC3        EQU 0x37
SEC4        EQU 0x38

; timer registers
TIME1       EQU 0x39
TIME2       EQU 0x40
```

```
TIME3       EQU 0x41
TIME4       EQU 0x42

COUNT       EQU 0x43                ; for debugging
TMRTEST     EQU 0x44
COUNTERWRAP EQU 0x45

; flag bits
; 7 = time flag
; 6 = enter code flag
; 5 = armed flag
; 4 = accept flag
; 3
; 2
; 1
; 0
FLAGS       EQU 0x70
TIMEF       EQU 7                   ; timer flag
GETTIMEF    EQU 6                   ; waiting for new time input flag
ONEMINF     EQU 5                   ; less than a minute left flag
ARMEDF      EQU 4                   ; armed flag
ACCEPTF     EQU 3                   ; accept flag
NEWINPUT    EQU 2                   ; new input in buffer flag
ENTERF      EQU 1                   ; enter pressed flag
ARMF        EQU 0                   ; 'bomb has ability to be armed' flag

FLAGS2      EQU 0x71
OLDCODEF    EQU 7                   ; old code flag
CANCELF     EQU 6                   ; cancel flag
CODEF       EQU 5                   ; code flag

; constants (lowercase)

; number of states needed to count to 1...note: start at 0
num_states  EQU 0x04

; time to count 1/2 second using TMR1 (at 1Mhz)
timerval_hi EQU 0x3D
timerval_lo EQU 0x09

; pre-defined outputs to display
offhi       EQU 0xA0                ; blank | 0
offlo       EQU 0xFF                ; F | F
codehi      EQU 0xC0                ; C | 0
codelo      EQU 0xDE                ; D | E
oldhi       EQU 0xA0                ; blank | 0
oldlo       EQU 0x1D                ; 1 | D
```

```
dashes      EQU 0xBB                ; - | -
blank       EQU 0xAA                ; blank | blank

; inputs from keypad
enter       EQU 0x0A
cancel      EQU 0x0B
time        EQU 0x0C
setcode     EQU 0x0D
arm         EQU 0x0E
resetall    EQU 0x0F


; program
    org 0x00
        bra     start
    org 0x08
        bra     highinthandle
    org 0x18
        bra     lowinthandle

    org 0x20
highinthandle:
        btfsc   PIR1, CCP1IF        ; is the timer interrupt set?
        bra     counter_interrupt   ; yes: go to timer interrupt
        btfsc   INTCON, RBIF        ; is the RB flag set?
        bra     check_rbif          ; yes: check which interrupt it is
        retfie
check_rbif
        btfss   PORTB, RB4          ; is tamper protection circuit open?
        bra     tamper_interrupt    ; yes? bomb tampered with..
        ;btfsc  PORTD, RD2          ; is srq set?
        ;bra    input_interrupt     ; yes? fpga wants to send info
        retfie                      ; no? unsupported interrupt, return

lowinthandle:
        retfie

start:

        ; set SCK, SDO as output, SDI as input
        bcf     TRISC, RC3
        bcf     TRISC, RC5
        bsf     TRISC, RC4

        clrf    TRISB               ; set most of port b for output/unused
        bsf     TRISB, RD2          ; but leave RD2 for input for SRQ
        bsf     TRISB, RB4          ; input for tamper protection
```

```
bcf     TRISB, RB2          ; and set port B pin 0 as output
bcf     PORTB, RB2          ; turn off sound element

setf    TRISD               ; leave most of portd floating
bcf     TRISD, RD0          ; SPI acknowledge bit = output
bcf     PORTD, RD0          ; clear SAC
bcf     TRISD, RD7          ; explode_op indicator
bcf     PORTD, RD7          ; clear explode indicator

; ********timeR 1 SETTINGS********
; 7     RD16    = 1         (ie. synchronously read all 16 bits)
; 5-4   T1CKPS[1:0] = 10    (ie. clock prescale = 1:4)
; 3     T1OSCEN = 0         (ie. disable timer oscillator
; 1     TMR1CS  = 0         (ie. use internal clock Fosc/4)
; 0     TMR1ON  = 1         (ie. turn on timer)
; note: timer1 has to be turned on before initializing the
; CCP module, or it will not compare to the timer correctly
; and interrupts will never occur
movlw   0xA1
movwf   T1CON
; Disable timer 3
clrf    T3CON

; initial CCP to interrupt every second
movlw   timerval_lo
movwf   CCPR1L
movlw   timerval_hi
movwf   CCPR1H

; initialize countdown timer and security code for testing
movlw   0x99
movwf   COUNTHI
movlw   0x59
movwf   COUNTLO
movlw   0x01
movwf   SEC1
movlw   0x02
movwf   SEC2
movlw   0x03
movwf   SEC3
movlw   0x04
movwf   SEC4
; initialize timer state to 0
clrf    TIMERSTATE
; clear flags
clrf    FLAGS
clrf    FLAGS2
```

```
        ; **************SPI**************
        ; set SSPCON1
        bsf     SSPCON1,SSPEN       ; enable serial port
        bcf     SSPCON1,CKP         ; idle clock is low
        ; master mode with OSC/64
        bcf     SSPCON1,SSPM0
        bsf     SSPCON1,SSPM1
        bcf     SSPCON1,SSPM2
        bcf     SSPCON1,SSPM3
        ; set SSPSTAT
        bsf     SSPSTAT, CKE        ; transmit data on rising edge clk
        bsf     SSPSTAT, SMP        ; sample at end of output time


        ; ************INTERRUPTS************
        clrf    INTCON
        bsf     INTCON, GIE         ; set global interrupts
        bsf     INTCON, PEIE        ; set peripheral interrupts
        bcf     INTCON, RBIF        ; must clear flag before enabling interrupt
        movf    PORTB, 0            ; read port B to ensure RBIF cleared
        bsf     INTCON, RBIE        ; set port B change interrupt
        ; disable priority interrupts
        movlw   0x1F
        movwf   RCON
        ; CCP1CON
        ; 3-0 CCP1M[3-0] = 1010 (ie. compare mode and set interrupt flag)
        movlw   0x0A
        movwf   CCP1CON
        ; set CCP1 interrupt enable
        clrf    PIR1                ; make sure PIR1, CCP1IF is off
        clrf    PIE1                ; clear all interrupt enables
        bsf     PIE1, CCP1IE        ; ...except CCP1 interrupt

        clrf    TMR1H               ; clear timer1
        clrf    TMR1L
        ; display off
        call    queueoff
        call    display

;*****************************************************************
;***********************   MAIN WAIT LOOP  ***********************
;*****************************************************************

main:
        btfss   FLAGS, NEWINPUT     ; test if new input has been entered
        bra     main                ; no? busy-wait
```

```
        ; check inputs
        btfsc   FLAGS, ARMEDF       ; armed?
        bra     checkcancelonly     ; yes? check only if cancel pressed
        movlw   arm
        cpfslt  INPUTBUF            ; arm?
        bra     armbomb_op
        movlw   setcode
        cpfslt  INPUTBUF            ; set code?
        bra     setcode_op
;       movlw   time
;       cpfslt  INPUTBUF            ; set time?
;       bra     settime_op
checkcancelonly
        movlw   cancel
        cpfslt  INPUTBUF            ; cancel?
        bra     cancel_op
        bra     main

;*****************************************************************
;************************    INTERRUPTS  ************************
;*****************************************************************

; counter interrupt
counter_interrupt:
        ; one second timer (TMR1) has interrupted
        ; Transmit the next display output to the FPGA (if counting down)
        incf    COUNT, 1
        clrf    TMR1H
        clrf    TMR1L
        bcf     PIR1, CCP1IF        ; clear interrupt flag
        movlw   0x03
        cpfseq  TIMERSTATE          ; are we in the last state?
        bra     no_decrement        ; no? don't decrement timer
;       clrf    TIMERSTATE
        btfsc   FLAGS, ARMEDF       ; yes? is the bomb armed?
        call    dectimer            ; yes? decrement timer
no_decrement
        btfsc   PORTD, RD2          ; check if srq high
        call    input_interrupt     ; yes? get input

        btfsc   FLAGS, ONEMINF      ; check if < one min left
        call    onemindisplay       ; yes? call special display function
        btfsc   FLAGS2, CANCELF     ; check if cancel pressed and bomb armed
        call    cancelarmeddisplay  ; yes? call special display function
        btfsc   FLAGS2, OLDCODEF    ; check if asking for old code
        call    oldcodedisplay      ; yes? call special display function
        btfsc   FLAGS2, CODEF       ; check if asking for code
```

```
        call    codedisplay         ; yes? call special display function

        incf    TIMERSTATE          ; increment timer state
        movlw   num_states
        cpfseq  TIMERSTATE          ; are we in the last state?
        bra     disp                ; no? don't decrement timer
        clrf    TIMERSTATE
disp
        call    display             ; update display always
        btfss   FLAGS, ARMEDF       ; bomb armed?
        retfie                      ; no? return, enabling global interrupts
        movlw   0x00
        cpfseq  COUNTHI             ; check if top of timer is zero
        retfie                      ; no? return, enabling global interrupts
        bsf     FLAGS, ONEMINF      ; set less than one minute left flag
        cpfseq  COUNTLO             ; yes? check if bottom of timer is zero
        retfie                      ; no? return, enabling global interrupts
        bra     explode_op          ; yes? explode!

;*****************************************************************
; input interrupt
input_interrupt:
        bsf     PORTB, RB2          ; sound element on
        call    waitabit            ; make sure that SPI is finished before
                                    ; raising SAC
        bsf     PORTD, RD0          ; raise SAC (SPI acknowledgement of RD2)
        movf    DISPLAYLO, 0        ; "arbitrary" dummy data (cheap hack)
        call    sendssp             ; send dummy data over SPI
recloop
        btfss   SSPSTAT, BF         ; wait for SPI to finish
        bra     recloop
        movff   SSPBUF, INPUTBUF    ; read digit in from SPI
        bsf     FLAGS, NEWINPUT     ; set new input flag
        movf    PORTB, 0            ; clear interrupt FLAGS
        bcf     INTCON, RBIF
        bcf     PIR1, SSPIF
        bcf     PORTD, RD0          ; clear SAC
        bcf     PORTB, RB2          ; sound element off
        ;retfie                     ; return, enabling global interrupts
        return                      ; return to counter_interrupt
;*****************************************************************
; tamper interrupt
tamper_interrupt:
        btfsc   FLAGS, ARMEDF       ; bomb armed?
        bra     explode_op          ; yes? EXPLODE!
        retfie                      ; no? back to main
```

```
;*****************************************************************
;**********************    OP FUNCTIONS   **********************
;*****************************************************************

; cancel operation
; METHOD
; checks to see if bomb armed. If so, prompts for code
cancel_op:
        bcf     FLAGS, NEWINPUT     ; reading...clear new input flag
        btfss   FLAGS, ARMEDF       ; armed?
        bra     main                ; no? nothing to cancel, back to main
        bsf     FLAGS2, CANCELF     ; yes? set cancel request flag
        call    getuserinput        ; get user input
        bra     nextaction          ; perform next action

;*****************************************************************
; arm bomb operation
; METHOD
; if armflag not set
;  sets timeflag, sets display to show "code" and calls the
;  getuserinput function
; else
;  sets armedflag to start bomb
armbomb_op:
        bcf     FLAGS, NEWINPUT     ; reading...clear new input flag
        btfsc   FLAGS, ARMF         ; if time and code inputted
        bra     startcountdown      ; return to main
        bsf     FLAGS, TIMEF        ; otherwise...set time flag
;       call    queuecode           ; display "code"
        bsf     FLAGS2, CODEF       ; set code flag
        call    getuserinput        ; get user input
        bra     nextaction          ; perform next action
startcountdown
        bsf     FLAGS, ARMEDF       ; set armed status
        bra     main                ; return to main

;*****************************************************************
; set code operation
setcode_op:
        bcf     FLAGS, NEWINPUT     ; reading...clear new input flag
        bsf     FLAGS2, OLDCODEF    ; set old code flag
        call    getuserinput        ; get user input
        bra     nextaction          ; perform next action

;*****************************************************************
; arm bomb operation
;armbomb_op:
```

```
;       bcf     FLAGS, NEWINPUT     ; reading...clear new input flag
;       bra     main
        ; start counter if code accepted
;       btfss   FLAGS, ARMF         ; can bomb be armed?


;*****************************************************************
; blow up the bomb now...for debugging
explode_op:
        bcf     FLAGS, NEWINPUT     ; reading...clear new input flag
        bcf     FLAGS, ARMEDF       ; clear armed flag
        bsf     PORTB, RB2          ; turn speaker on
        bsf     PORTD, RD7          ; turn on led
makenoise
;       btfss   FLAGS, NEWINPUT     ; check for new input
        bra     makenoise
;       movlw   resetall
;       cpfseq  INPUTBUF            ; is the input 'resetall'?
;       bra     clearflag
;       bra     clearall
;clearflag
;       bcf     FLAGS, NEWINPUT
;       bra     makenoise



;*****************************************************************
;********************    HELPER FUNCTIONS   **********************
;*****************************************************************


;*****************************************************************
; getuserinput
; METHOD:
; waits till newinput flag is set then checks if enter or cancel were pressed
; and returns, setting the appropriate flag if so. if neither enter nor cancel
; were pressed, puts the new input into the shift register and waits for more
; input

getuserinput:   ; busy wait loop
        btfss   FLAGS, NEWINPUT     ; test if new input has been entered
        bra     getuserinput        ; no? busy-wait
        bcf     FLAGS, NEWINPUT     ; reading...clear new input flag
        movlw   0x0C
        cpfslt  INPUTBUF            ; check if input other than number,
                                    ; ..enter, or cancel
        bra     getuserinput        ; yes? ignore input and continue
```

```
        movlw   enter               ; no? ...
        cpfseq  INPUTBUF            ; check if input is "enter"
        bra     notpressedenter     ; no?  keep checking
        bra     pressedenter        ; yes? return with return flag set
notpressedenter
        movlw   cancel
        cpfseq  INPUTBUF            ; check if input is "cancel"
        bra     notpressedcancel    ; no?  keep checking
        bra     quit                ; yes? return with enter flag not set
notpressedcancel
        call    inputshiftreg       ; put code in shift reg
        btfsc   FLAGS, GETTIMEF     ; inputting time?
        call    displaytime         ; yes? display the current time
        bra     getuserinput        ; either way wait for more input
pressedenter
        bsf     FLAGS, ENTERF       ; set enter flag
        return
quit
        bcf     FLAGS, ENTERF       ; clear enter flag
        return

;*****************************************************************
; nextaction
; PURPOSE:
; performs the next action after code is entered. 'pass' or 'fail' should be
; in the WREG when this function is called
nextaction:
        btfss   FLAGS, ENTERF       ; test if enter flag set
        bra     failed              ; no? clear all and wait in main
        bcf     FLAGS, ENTERF       ; yes? clear enterflag
        call    checkcode           ; check if entered code is correct
        btfss   FLAGS, ACCEPTF      ; correct?
        bra     failed              ; no - go to failed
passed                              ; yes - passed
        btfsc   FLAGS, ARMEDF       ; bomb armed flag set?
        bra     clearall            ; yes? clear all and wait in main
        btfsc   FLAGS, TIMEF        ; time flag set?
        bra     entertime           ; yes? branch to enter time function
;       btfsc   FLAGS, CODEF        ; code flag set?
        bra     enternewcode        ; yes? branch to enter new code function
failed
        btfsc   FLAGS, ARMEDF       ; bomb is armed?
        bra     clearbuf            ; yes? clear buffer only
        btfsc   FLAGS, CANCELF      ; cancel flag set?
        bra     clearbuf            ; yes? clear buffer only
        bra     clearall            ; no? clear all
```

```
;*****************************************************************
; decrement the timer

dectimer:                           ; decrements the timer by 1
        decf    COUNTLO
        bnn     declowonly
dechighandlow                       ; checks if low part of timer wrapped
                                    ; ..past 0x00 to avoid going to 0xFF
        decf    COUNTHI
        movf    COUNTHI, 0          ; move counter into wreg
        call    onewrapped
        movwf   COUNTHI
        movlw   0x59
        movwf   COUNTLO
        bra     next
declowonly
        movf    COUNTLO, 0          ; COUNTLO -> WREG
        call    onewrapped
        movwf   COUNTLO
next
        movff   COUNTLO, DISPLAYLO
        movff   COUNTHI, DISPLAYHI
        movlw   0x00
;       LFSR    FSR0, 0x00
;       movwf   TIMERSTATE          ; reset timerstate
        return

;*****************************************************************
; turn hex timer into decimal timer
onewrapped:
        movwf   COUNTERWRAP
        movlw   0x0F
        andwf   COUNTERWRAP, 0      ; bitmask top 4 bits to check for 0x_F
        movwf   TMRTEST
        movlw   0x0E
        cpfsgt  TMRTEST
        bra     done
        movlw   0x06
        subwf   COUNTERWRAP
done
        movf    COUNTERWRAP, 0      ; COUNTERWRAP -> WREG
        return

;*****************************************************************
; cancel bomb function
cancelbomb:
```

```
;*****************************************************************
; entertime function
entertime:

        movlw   blank
        movwf   DISPLAYLO
        movwf   DISPLAYHI
        bcf     FLAGS, NEWINPUT     ; reading...clear new input flag
        bcf     FLAGS2, CODEF       ; clear code flag
        bsf     FLAGS, GETTIMEF     ; getting new time
        movlw   0x00                ; clear buffer
        movwf   BUF1
        movwf   BUF2
        movwf   BUF3
        movwf   BUF4
getinput
        call    getuserinput
        call    displaytime
        movlw   0x00
        cpfseq  COUNTHI
        bra     timeok
        cpfseq  COUNTLO
        bra     timeok
        bra     getinput
timeok
;       bcf     FLAGS, GETTIMEF     ; done getting new time
        clrf    FLAGS               ; make sure no extra flags set
        clrf    FLAGS2
        bsf     FLAGS, ARMF
        bra     main

;*****************************************************************
; displaytime consolidates the buffers and displays them

displaytime:
        swapf   BUF1, 0             ; combine buf1 and buf2
        addwf   BUF2, 0
        movwf   DISPLAYHI
        movwf   COUNTHI
        swapf   BUF3, 0             ; combine buf3 and buf4
        addwf   BUF4, 0
        movwf   DISPLAYLO
        movwf   COUNTLO
        return
;*****************************************************************
; enternewcode
```

```
enternewcode:
        bcf     FLAGS, NEWINPUT     ; reading...clear new input flag
        bcf     FLAGS2, OLDCODEF    ; clear oldcode flag
        bsf     FLAGS2, CODEF       ; set code flag
;       call    queuecode           ; display "code"
        movlw   0x00                ; clear buffer
        movwf   BUF1
        movwf   BUF2
        movwf   BUF3
        movwf   BUF4
        call    getuserinput        ; get new code
        movff   BUF1, SEC1          ; store new code
        movff   BUF2, SEC2
        movff   BUF3, SEC3
        movff   BUF4, SEC4
        bra     clearall            ; back to main

;*****************************************************************
clearbuf:   ; clear buffer and return to main
        movlw   0x00
        movwf   BUF1
        movwf   BUF2
        movwf   BUF3
        movwf   BUF4
        bcf     FLAGS2, CANCELF     ; clear cancel flag
        bra     main

;*****************************************************************
clearall:   ; clear all flags, buffer, and timer registers
            ; stop timer and return to main
        clrf    FLAGS               ; clear flags
        clrf    FLAGS2
        clrf    BUF1
        clrf    BUF2
        clrf    BUF3
        clrf    BUF4
        clrf    TIME1
        clrf    TIME2
        clrf    TIME3
        clrf    TIME4
        bcf     PORTB, RB2          ; turn sound off
        call    queueoff            ; set to display "off"
        bra     main

;*****************************************************************
```

```
error1:
        ; input error...reset code accept FLAGS and branch back to main
        movlw   0
;       movwf   accept_flag         ; reset accept flag
        movlw   0x11
        movwf   INPUTBUF            ; reset input buffer
        bra     main

;*****************************************************************

sendssp:
        bcf     SSPCON1, WCOL
        movwf   SSPBUF
        btfss   SSPCON1, WCOL
        return
        bra     sendssp

;*****************************************************************
inputshiftreg:  ; shift register for input
        movff   BUF2, BUF1
        movff   BUF3, BUF2
        movff   BUF4, BUF3
        movff   INPUTBUF,BUF4
        return

;*****************************************************************
checkcode:      ; check if values in buf match the stored code
        movf    SEC1, 0
        cpfseq  BUF1
        bra     codefail
        movf    SEC2, 0
        cpfseq  BUF2
        bra     codefail
        movf    SEC3, 0
        cpfseq  BUF3
        bra     codefail
        movf    SEC4, 0
        cpfseq  BUF4
        bra     codefail
        bsf     FLAGS, ACCEPTF
codefail
        return

;*****************************************************************
display:
        movf    DISPLAYHI, 0        ; DISPLAYHI -> WREG
        call    sendssp             ; WREG -> SPI
```

```
        movf    DISPLAYLO, 0        ; DISPLAYLO -> WREG
        call    sendssp             ; WREG -> SPI
        return
;*****************************************************************
; onemindisplay flashes the timer if less than one min remaining
onemindisplay:
        movlw   0x01
        cpfsgt  TIMERSTATE          ; check if timerstate > 2
        bra     blankscreen         ; no? show clear
        call    queuetimer          ; yes? show timer value
        return
blankscreen
        call    queueblank
        return

;*****************************************************************
cancelarmeddisplay:
        movlw   0x01
        cpfsgt  TIMERSTATE          ; check if timerstate < 1
        bra     showtime            ; no? show clear
        call    queuecode           ; yes? show timer value
        return
showtime
        call    queuetimer
        return

;*****************************************************************
oldcodedisplay:
        movlw   0x01
        cpfsgt  TIMERSTATE          ; check if timerstate > 2
        bra     old                 ; no? show clear
        call    queuecode           ; yes? show timer value
        return
old
        call    queueold
        return

;*****************************************************************
codedisplay:
        movlw   0x00
        cpfsgt  TIMERSTATE          ; check if timerstate = 0
        bra     code1               ; no? show "C   "
        movlw   0x01
        cpfsgt  TIMERSTATE          ; check if timerstate = 1
        bra     code2               ; no? show "C0  "
        movlw   0x02
        cpfsgt  TIMERSTATE          ; check if timerstate = 2
```

```
        bra     code3               ; no? show "C0D "
        call    queuecode           ; yes? show "C0DE"
        return
code1
        call    queuecode1
        return
code2
        call    queuecode2
        return
code3
        call    queuecode3
        return


;*****************************************************************
queuecode1:
        movlw   0xCA                ; set to display "C   "
        movwf   DISPLAYHI
        movlw   0xAA
        movwf   DISPLAYLO
        return

;*****************************************************************
queuecode2:
        movlw   0xC0                ; set to display "C0  "
        movwf   DISPLAYHI
        movlw   0xAA
        movwf   DISPLAYLO
        return

;*****************************************************************
queuecode3:
        movlw   0xC0                ; set to display "C0D "
        movwf   DISPLAYHI
        movlw   0xDA
        movwf   DISPLAYLO
        return

;*****************************************************************
queuecode:
        movlw   codehi              ; set to display "C0DE"
        movwf   DISPLAYHI
        movlw   codelo
        movwf   DISPLAYLO
        return

;*****************************************************************
```

```
queueblank:
        ; display bomb status (over SPI): blank
        movlw   blank
        movwf   DISPLAYHI
        movlw   blank
        movwf   DISPLAYLO
        return

;*****************************************************************
queueold:
        movlw   oldhi               ; set to display "OlD"
        movwf   DISPLAYHI
        movlw   oldlo
        movwf   DISPLAYLO
        return

;*****************************************************************
queuetimer:
        ; display bomb status (over SPI): timer value
        movff   COUNTHI, DISPLAYHI
        movff   COUNTLO, DISPLAYLO
        return

;*****************************************************************
queueoff:
        ; display bomb status (over SPI): " OFF"
        movlw   offhi
        movwf   DISPLAYHI
        movlw   offlo
        movwf   DISPLAYLO
        return

;*****************************************************************
waitabit:
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
```

```
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        call    waitabitmore
        return

waitabitmore:
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
```

```
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        return
    end
```

C FPGA Behavior

C.1 Block Diagrams

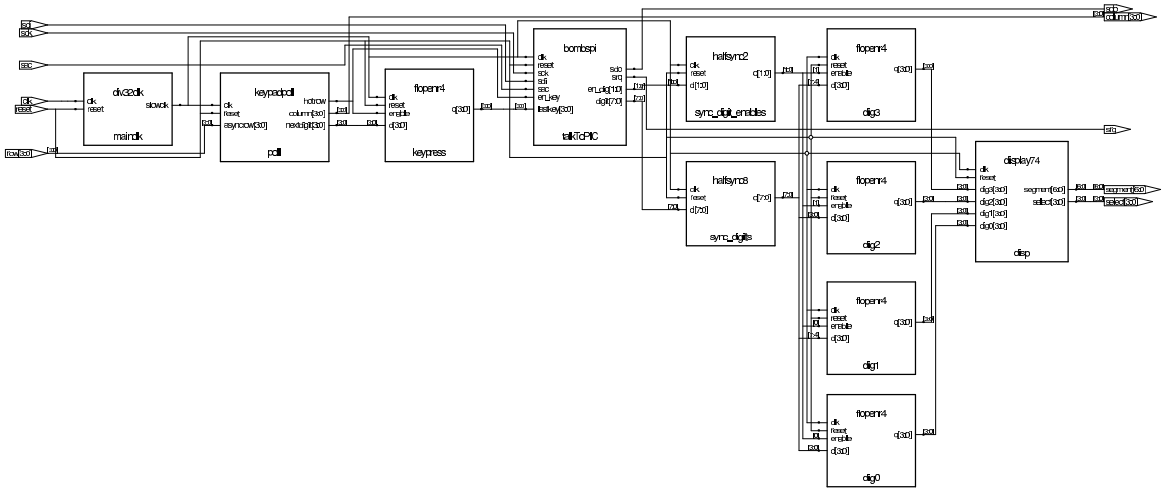

Figure 5: Overview of FPGA functionality

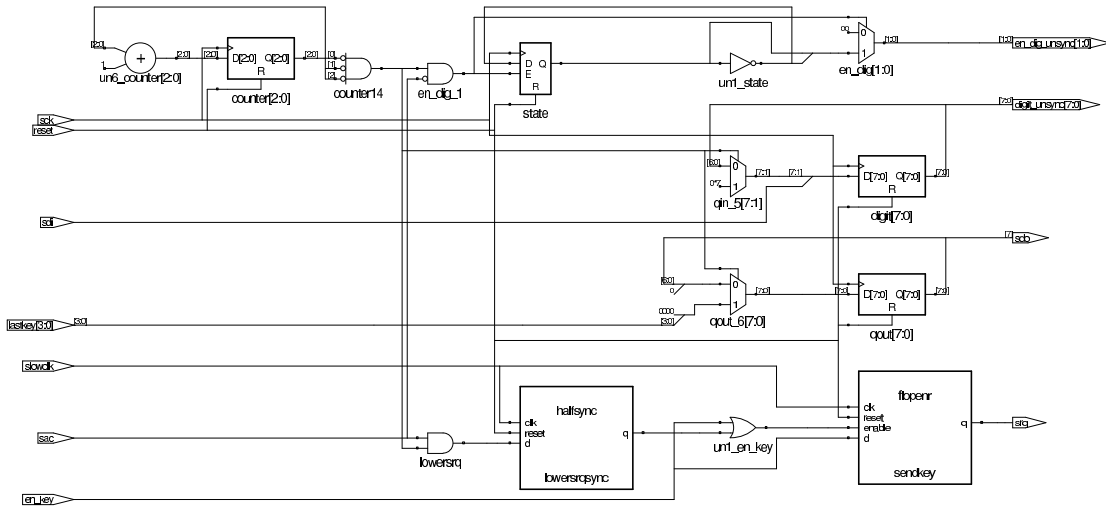

Figure 6: Overview of SPI communication implemented on the FPGA

C.2 Verilog

For the sake of completeness, all Verilog has been included in this appendix. However, the modules of primary interest are main, bombspi, and possibly keypadpoll, though it is similar to the lab 4 code.

C.2.1 main.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Input from keypad, output to 7-segment displays, communication via SPI
// $Id: main.v,v 1.12 2003/12/15 23:42:58 eric Exp $
module main(clk,reset,sck,sdi,sac,srq,sdo,row,column,segment,select);
    input           clk, reset;  // Clock and ground
    input           sck, sdi; // Serial clock and Serial data in
    input           sac;      // Serial data request accepted
    output          srq;      // Serial data send request
    output          sdo;      // Serial data out
    input   [3:0]   row;      // Keypad rows
    output  [3:0]   column;   // Keypad columns
    output  [6:0]   segment;  // 7-segment display output
    output  [3:0]   select;   // Which digit is segment currently showing?

    wire            slowclk;  // divide 1MHz down to about 32KHz.
    wire    [3:0]   newkey, lastkey, dig3, dig2, dig1, dig0;
    wire            en_key;   // enable shifting the value of the keypress
                              // only when new data is available.
    wire    [1:0]   en_dig_unsync, en_dig;  // enable bit for each digit pair
```

```
    wire    [7:0]   digit_unsync, digit;    // the digits themselves

    // Make the slow clock.
    div32clk mainclk(clk, reset, slowclk);

    // Do the hard work of polling.
    keypadpoll poll(slowclk, reset, row, column, newkey, en_key);

    // Save the most recent key press.
    flopenr4 keypress(slowclk, reset, en_key, newkey, lastkey);

    // Deal with the I/O from the PIC
    bombspi talkToPIC(slowclk, reset, sck, sdi, sdo, srq, sac, en_key,
    lastkey, en_dig_unsync, digit_unsync);

    halfsync8 sync_digits(slowclk, reset, digit_unsync, digit);
    halfsync2 sync_digit_enables(slowclk, reset, en_dig_unsync, en_dig);

    // Save the currently displaying digits.
    flopenr4 dig3(slowclk, reset, en_dig[1], digit[7:4], dig3);
    flopenr4 dig2(slowclk, reset, en_dig[1], digit[3:0], dig2);
    flopenr4 dig1(slowclk, reset, en_dig[0], digit[7:4], dig1);
    flopenr4 dig0(slowclk, reset, en_dig[0], digit[3:0], dig0);

    // And display those digits.
    display74 disp(slowclk, reset, dig3, dig2, dig1, dig0, segment, select);
endmodule
```

C.2.2 bombspi.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Deal with communication with PIC via SPI for bomb controller.
// $Id: bombspi.v,v 1.11 2003/12/16 05:37:25 eric Exp $
module bombspi(clk,reset,sck,sdi,sdo,srq,sac,en_key,lastkey,en_dig,digit);
    input           clk, reset;
    input           sck, sdi; // Serial clock, serial data in
    output          sdo;      // Serial data out
    output          srq;      // Serial data send request
    input           sac;      // Serial data request accepted
    input           en_key;   // Keypress trigger
    input   [3:0]   lastkey;  // The last key pushed
    output  [1:0]   en_dig;   // Which digit pair is currently being output?
    output  [7:0]   digit;    // Output digit

    reg  [7:0]  qin, qout;
```

```
wire        lowersrq;
wire [1:0]  en_dig;
wire [7:0]  digit;
reg         state;
reg  [2:0]  counter;


always @(posedge sck, posedge reset)
    if (reset) begin
        counter <= 3'b000;
        qin <= 8'b0000_0000;
        state <= 0;
        qout <= 8'b0000_0000;
    end
    else case (counter)                 // data transfer in 8-bit blocks
        3'b000: begin                   // so we do special stuff between
            counter <= counter + 1;     // always increment counter
            qin <= {7'b000_0000, sdi};  // restart qin with the next bit
            qout <= {4'b0000, lastkey}; // reset qout with the output data
            if (~sac) state <= ~state;  // if this is a receive cycle,
                                        // ..then it's for the next digit,
                                        // ..so switch state
        end
        default: begin                  // just send and/or receive
            counter <= counter + 1;     // increment counter
            qin <= {qin[6:0], sdi};     // shift new bit into qin
            qout <= {qout[6:0], 1'b0};  // shift new bit out of qout into
                                        // ..sdo (happens below) and shift
                                        // ..qout over by one.
            //state <= state;           // no reason to change state, but
                                        // ..specifying that explicitly
                                        // ..makes the compiler laugh
        end
    endcase


// lower srq when it's acknowledged (sac is high) and we get to the
// ..beginning sending state (counter 0)
assign lowersrq = (sac & (counter == 3'b000));
// en_dig either selects one pair of digits to be active, but only when
// ..we're in counter state 0 and sac is low.  If sac is high, we're
// ..sending data so we'd better not try to display the garbage that's
// ..being sent to us
assign en_dig = (~sac & (counter == 3'b000)) ? {state, ~state} : 2'b00;
// sdo is always going to be the high bit of qout, even though it doesn't
// ..matter if sac is high
assign sdo = qout[7];
```

```
    // digit only matters when en_dig is hot, but that only happens when qin
    // ..is okay to be read
    assign digit = qin;

    halfsync lowersrqsync(clk, reset, lowersrq, sync_lowersrq);
    flopenr sendkey(clk, reset, en_key | sync_lowersrq, en_key, srq);

endmodule
```

C.2.3 keypadpoll.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Controller to sample input from 4x4 matrix keypad.
// $Id: keypadpoll.v,v 1.7 2003/12/15 23:42:58 eric Exp $
module keypadpoll(clk,reset,asyncrow,column,nextdigit,hotrow);
    input               clk;
    input               reset;
    input      [3:0]    asyncrow;
    output     [3:0]    column;
    output reg [3:0]    nextdigit;
    output              hotrow;

    wire       [3:0]    row;
    /* Actually, synchronizing the row input delays it by two clock cycles
     * and breaks everything.  However, we can get away with not synchronizing
     * it because the clock we're running here is approximately 62.5kHz and the
     * aperture in which metastability could be a problem is 20ns, a tiny
     * fraction of the clock cycle.
     */
    //fullsync4 rowsync(clk, reset, asyncrow, row);
    assign row = asyncrow;

    // First we define output values.
    parameter ZERO  = 4'b0000;
    parameter ONE   = 4'b0001;
    parameter TWO   = 4'b0010;
    parameter THREE = 4'b0011;
    parameter FOUR  = 4'b0100;
    parameter FIVE  = 4'b0101;
    parameter SIX   = 4'b0110;
    parameter SEVEN = 4'b0111;
    parameter EIGHT = 4'b1000;
    parameter NINE  = 4'b1001;
    parameter ENTER = 4'b1010;
    parameter CANCEL= 4'b1011;
```

```
parameter TIME  = 4'b1100;
parameter CODE  = 4'b1101;
parameter ARM   = 4'b1110;
parameter BOOM  = 4'b1111;

// Now we define states for polling various columns.
// One-hot encoding seems like a good way to do this
parameter poll0 = 4'b0001;
parameter poll1 = 4'b0010;
parameter poll2 = 4'b0100;
parameter poll3 = 4'b1000;

/* When a row first goes hot, we want to pulse enable to the flip-flops
 * that store the most recently pressed number, but we don't want to
 * leave it high while the button is being held down, or the number will
 * just flood through.  This pulses hotrow for one clock cycle when any
 * row first goes high.
 */
edgepulse enable(clk, reset, |row, hotrow);

reg [3:0] state, nextstate;
// State Register
always @(posedge clk, posedge reset)
    if (reset) state <= poll0;
    else       state <= nextstate;

// Next State
always @(state, row)
    case (state)
        // Stay in current state if any row hot, else move to next state.
        poll0:  if (|row) nextstate <= poll0;
                else nextstate <= poll1;

        poll1:  if (|row) nextstate <= poll1;
                else nextstate <= poll2;

        poll2:  if (|row) nextstate <= poll2;
                else nextstate <= poll3;

        poll3:  if (|row) nextstate <= poll3;
                else nextstate <= poll0;

            default: nextstate <= poll0;
    endcase

// Output
assign column = state;  // Hence the encoding for the different states.
```

```
    always @(state, row)    // This could have been combined with next state
        case (state)        // logic, but that would intermingle logically
                            // separate parts.
            poll0:  if (row[0]) nextdigit <= ENTER;
                    else if (row[1]) nextdigit <= SEVEN;
                    else if (row[2]) nextdigit <= FOUR;
                    else if (row[3]) nextdigit <= ONE;
                    else nextdigit <= CANCEL;
                        /* This CANCEL should never actually propogate to the
                         * flip-flops that store the two recent numbers because
                         * if none of the rows is hot, those flip-flops won't
                         * get their enable signal.  However, we want to
                         * specify it here to avoid implying latches.
                         */
            poll1:  if (row[0]) nextdigit <= ZERO;
                    else if (row[1]) nextdigit <= EIGHT;
                    else if (row[2]) nextdigit <= FIVE;
                    else if (row[3]) nextdigit <= TWO;
                    else nextdigit <= CANCEL;    // Same with this CANCEL.
            poll2:  if (row[0]) nextdigit <= CANCEL;
                    else if (row[1]) nextdigit <= NINE;
                    else if (row[2]) nextdigit <= SIX;
                    else if (row[3]) nextdigit <= THREE;
                    else nextdigit <= CANCEL;    // And with this one.
            poll3:  if (row[0]) nextdigit <= BOOM;
                    else if (row[1]) nextdigit <= ARM;
                    else if (row[2]) nextdigit <= CODE;
                    else if (row[3]) nextdigit <= TIME;
                    else nextdigit <= CANCEL;    // And this one.
            default: nextdigit <= CANCEL;        // And this one.
                /* If we get to the default, we managed to find our way into
                 * an invalid state, which shouldn't happen.
                 */
            endcase
endmodule
```

C.2.4 display.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Display module to handle four seven segment displays.
// $Id: display.v,v 1.5 2003/11/26 11:12:06 eric Exp $
module display74(clk,reset,dig3,dig2,dig1,dig0,segment,select);
    input        clk, reset;
    input  [3:0] dig3, dig2, dig1, dig0;
```

```
    output [6:0] segment;   // Connect the actual display to these pins
    output [3:0] select;    // Which digit is segment currently showing?

    reg    [3:0] state;
    reg    [3:0] nextstate;
    reg    [3:0] digit;

    parameter DIG0 = 4'b1110;
    parameter DIG1 = 4'b1101;
    parameter DIG2 = 4'b1011;
    parameter DIG3 = 4'b0111;

    // State Register
    always @(posedge clk, posedge reset)
        if (reset) state <= DIG0;
        else       state <= nextstate;

    // Next State
    always @(state, dig3, dig2, dig1, dig0)
        case (state)
            DIG0: begin
                digit <= dig0;
                nextstate <= DIG1;
            end
            DIG1: begin
                digit <= dig1;
                nextstate <= DIG2;
            end
            DIG2: begin
                digit <= dig2;
                nextstate <= DIG3;
            end
            DIG3: begin
                digit <= dig3;
                nextstate <= DIG0;
            end
            default: begin  // should never get here
                digit <= dig0;
                nextstate <= DIG0;
            end
        endcase

    // Output
    assign select = state; // here's the benefit of 1-hot state encoding
    sevenseg current(digit, segment);

endmodule
```

C.2.5 sevenseg.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Seven segment display decoder
// Based on example in section 4.4 of David Harris's Introduction to Verilog
// $Id: sevenseg.v,v 1.5 2003/11/26 11:12:06 eric Exp $
module sevenseg(s,seg);
    input       [3:0] s;
    output reg  [6:0] seg;

    //          Number          abc_defg     hex
    parameter   BLANK   =   7'b111_1111; // 0x7f
    parameter   ZERO    =   7'b000_0001; // 0x01
    parameter   ONE     =   7'b100_1111; // 0x4f
    parameter   TWO     =   7'b001_0010; // 0x12
    parameter   THREE   =   7'b000_0110; // 0x06
    parameter   FOUR    =   7'b100_1100; // 0x4c
    parameter   FIVE    =   7'b010_0100; // 0x24
    parameter   SIX     =   7'b010_0000; // 0x20
    parameter   SEVEN   =   7'b000_1111; // 0x0f
    parameter   EIGHT   =   7'b000_0000; // 0x00
    parameter   NINE    =   7'b000_0100; // 0x04
    parameter   CEE     =   7'b011_0001; // 0x31
    parameter   DEE     =   7'b100_0010; // 0x42
    parameter   EEE     =   7'b011_0000; // 0x30
    parameter   EFF     =   7'b011_1000; // 0x38
    parameter   DASH    =   7'b111_1110; // 0x7e

    always @(s)
        case (s)
             0: seg <= ZERO;
             1: seg <= ONE;
             2: seg <= TWO;
             3: seg <= THREE;
             4: seg <= FOUR;
             5: seg <= FIVE;
             6: seg <= SIX;
             7: seg <= SEVEN;
             8: seg <= EIGHT;
             9: seg <= NINE;
            10: seg <= BLANK; // No need for hex digits here, but we want to
            11: seg <= DASH;  // ..be able to spell out certain things
            12: seg <= CEE;
            13: seg <= DEE;
            14: seg <= EEE;
```

```
            15: seg <= EFF;
            default: seg <= BLANK;
        endcase
endmodule
```

C.2.6 divclk.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Divide a clock signal
// $Id: divclk.v,v 1.3 2003/12/07 04:07:37 eric Exp $
module div16clk(clk,reset,slowclk);
    input   clk, reset;
    output  slowclk;

    reg [3:0] counter;

    always @(posedge clk, posedge reset)
        if (reset) counter <= 0;
        else counter <= counter + 1;

    assign slowclk = counter[3];
endmodule

module div32clk(clk,reset,slowclk);
    input   clk, reset;
    output  slowclk;

    reg [4:0] counter;

    always @(posedge clk, posedge reset)
        if (reset) counter <= 0;
        else counter <= counter + 1;

    assign slowclk = counter[4];
endmodule
```

C.2.7 edgepulse.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Go hot for one cycle when a signal goes hot
// $Id: edgepulse.v,v 1.5 2003/12/07 04:35:43 eric Exp $
module edgepulse(clk,reset,hot,enable);
    input clk;
    input reset;
```

```
    input hot;
    output enable;

    reg [1:0] state, nextstate;

    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;

    /* The idea here is to take a signal, hot, and when hot goes high, push
     * enable high for one clock cycle.  Then bring enable low again,
     * regardless of how long hot stays high.  Once hot goes low, be ready to
     * do it again.  Basically go high for one cycle at posedge hot.
     */

    // State Register
    always @(posedge clk, posedge reset)
        if (reset) state <= S0;
        else       state <= nextstate;

    // Next State
    always @(state, hot)
        case (state)
            S0: if (hot) nextstate <= S1;
            else nextstate <= S0;

            S1: nextstate <= S2;

            S2: if (~hot) nextstate <= S0;
            else nextstate <= S2;

            default: nextstate <= S0;
    endcase

    // Output
    assign enable = (state == S1);
endmodule

module edgepulse4(clk,reset,hot,enable);
    input clk;
    input reset;
    input  [3:0] hot;
    output [3:0] enable;

    edgepulse e3(clk, reset, hot[3], enable[3]);
    edgepulse e2(clk, reset, hot[2], enable[2]);
    edgepulse e1(clk, reset, hot[1], enable[1]);
```

```
    edgepulse e0(clk, reset, hot[0], enable[0]);
endmodule
```

C.2.8 flop.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Enabled, resettable, 4-bit flip-flop
// $Id: flop.v,v 1.4 2003/11/26 11:12:06 eric Exp $
module flopenr4(clk,reset,enable,d,q);
    input clk,reset,enable;
    input [3:0] d;
    output reg [3:0] q;

    always @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else if (enable) q <= d;
endmodule

module flopenr(clk,reset,enable,d,q);
    input clk,reset,enable;
    input d;
    output reg q;

    always @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else if (enable) q <= d;
endmodule
```

C.2.9 sync.v

```
// Eric Angell <eoa@cs.hmc.edu>
// Matt Livianu <mlivianu@hmc.edu>
// 2003/11/12
// Syncronizers of different bus widths
// $Id: sync.v,v 1.4 2003/12/15 23:42:58 eric Exp $
module fullsync(clk,reset,d,q);
    input clk,reset,d;
    output reg q;

    reg d2;

    always @(posedge clk, posedge reset)
        if (reset) d2 <= 0;
        else d2 <= d;
```

```
    always @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else q <= d2;

endmodule

module halfsync2(clk,reset,d,q);
    input clk,reset;
    input [1:0] d;
    output [1:0] q;

    halfsync q1(clk, reset, d[1], q[1]);
    halfsync q0(clk, reset, d[0], q[0]);

endmodule

module halfsync8(clk,reset,d,q);
    input clk,reset;
    input [7:0] d;
    output [7:0] q;

    halfsync2 q76(clk, reset, d[7:6], q[7:6]);
    halfsync2 q54(clk, reset, d[5:4], q[5:4]);
    halfsync2 q32(clk, reset, d[3:2], q[3:2]);
    halfsync2 q10(clk, reset, d[1:0], q[1:0]);

endmodule

module halfsync(clk,reset,d,q);
    input clk,reset,d;
    output reg q;

    always @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else q <= d;

endmodule
```