Laser Simon

Final Project Report
December 12, 2002
E155

Genevieve Breed and Nick Hertl

Abstract:

"Laser Simon" is much like the popular children's memory building game of "Simon." In the classic version, the user is given four uniquely colored buttons that light up in progressively more difficult sequences. Each button has a unique sound associated with it. After each sequence, the user is expected to push the buttons in the same order as they previously lit up. If they correctly perform this task, the system presents them with a more difficult sequence. If a wrong button is pressed, the system indicates failure and starts over. "Laser Simon" implements photo sensors instead of buttons and a laser pointer to activate them. The microcontroller creates a pseudo-random pattern for each game with a maximum of 256 different sequences that are stored in its internal SRAM. The HC11 also controls the LEDs displaying the sequence and outputs sound to the speaker system. The FPGA keeps track of the score and receives input from the photo diodes.

Introduction

It has been said that the average human brain can store 7-9 pieces of random information in its "cache." We wanted to design a game that could test this study on a variety of people. We also wanted to design a game that would implement lasers and photo diodes since this type of hardware was new and exciting to both team members.

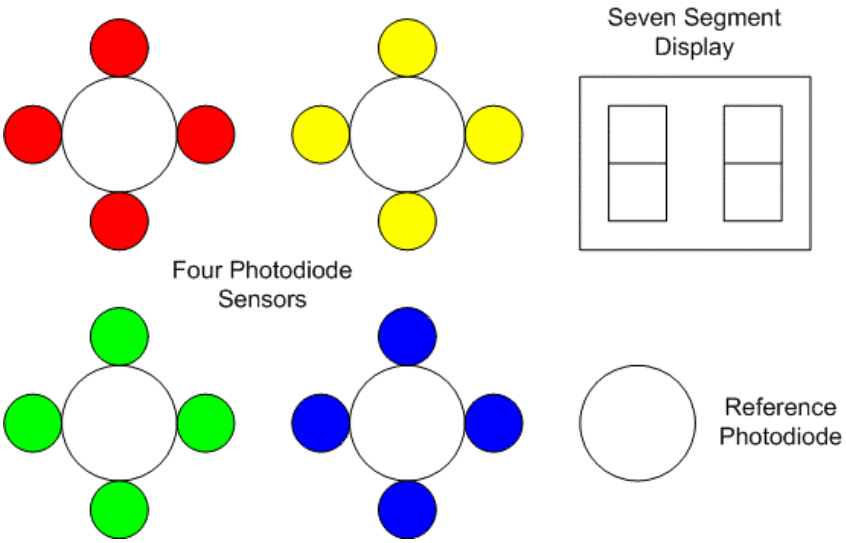


Figure 1: User Interface

Laser Simon plays a pseudo random sequence that the user tries to emulate. The system shows one light and plays one sound. If the player correctly repeats this pattern by shining the laser pointed onto the target that just lit up, he will be reward with the noise associated with that target and the sequence will increment by one. This continues until the player guesses wrong in which case a wrong (low frequency) sound will emit from the speaker. The game board for Laser Simon can be seen above in Figure 1.

The system breaks down into five main sections: sound hardware, laser sensing hardware, scoring, LEDs, and game play (when each of these systems operates). The HC11 handles game play because we need to store light states in memory and cycle through what would be a large number of states if implemented in hardware. The HC11 creates the light and sound emitting signals, while telling the FPGA when to increment the score. The FPGA controls the display for the score and relays the laser sensors data into the HC11. The sound hardware takes a square wave input and produces amplified sound of the appropriate frequency with smooth edges. The laser sensor hardware

outputs four bits representing whether or not the sensors are active at any given time. The overall block diagram in Figure 2 shows all system communication channels.

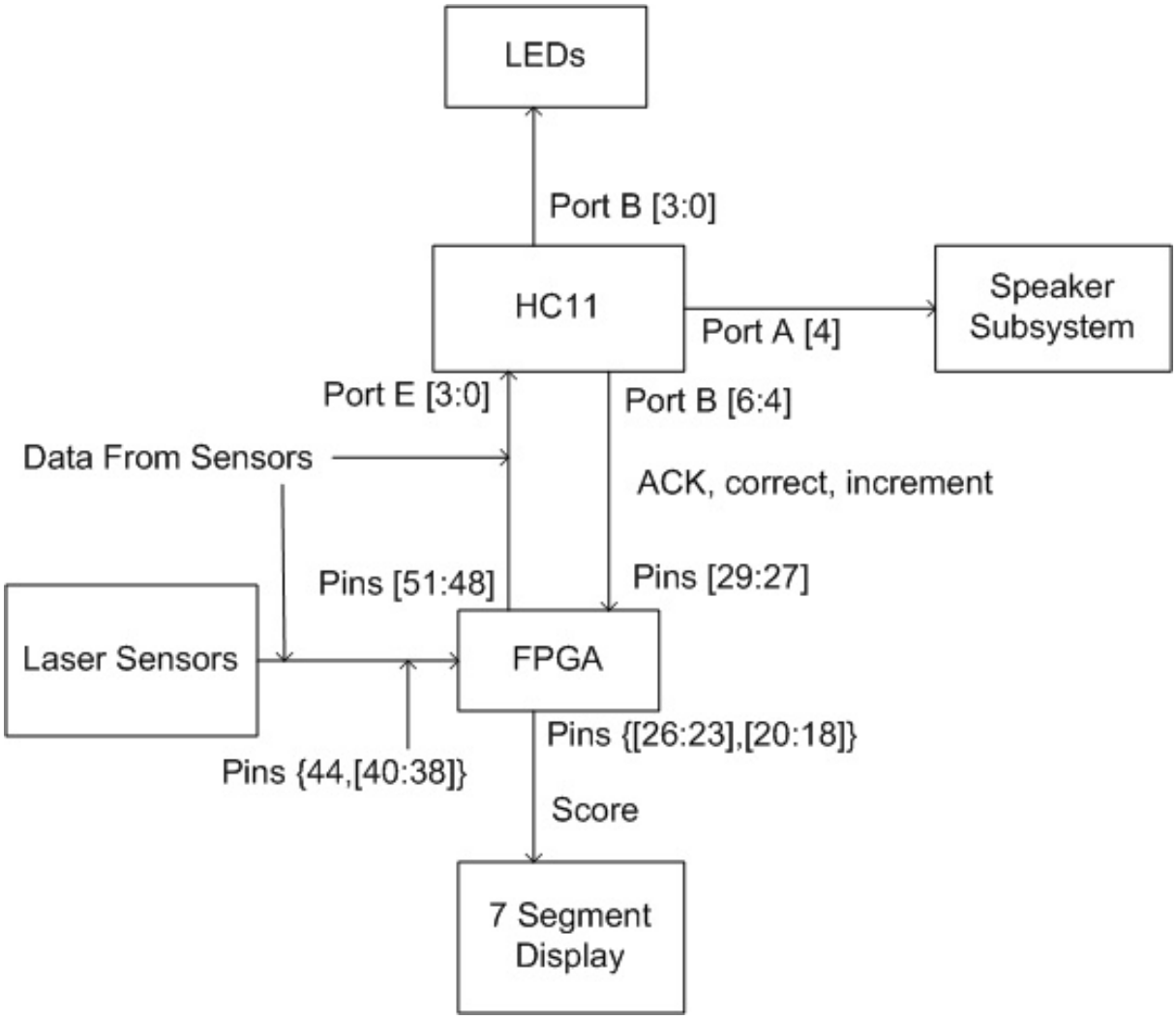


Figure 2: Overall Block Diagram

New Hardware

We used Optek's OP950 (Figure 3) photodiodes as our laser sensors. The OP950 provides a 50 percent relative response at 650 nm, which is the wavelength of the laser pointers (shown in Figure 4). The datasheet on these diodes can be found at http://www.optekinc.com/pdf/OP950.pdf.

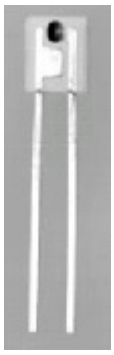


Figure 3: Photodiode

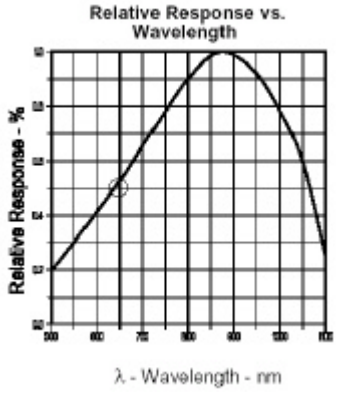


Figure 4: Relative Response of OP950 vs. Wavelength

Photodiodes output a small amount of current that is proportional to the amount of light entering the sensor. This current can be changed to a voltage using an op amp as shown in the bottom left portion of Figure 5. We used a 741 op amp in a common current-to-voltage converter circuit. The inverting input is a virtual ground since the diode can only generate a few tenths of a volt. The rest of the circuit will be explained later in the schematics section.

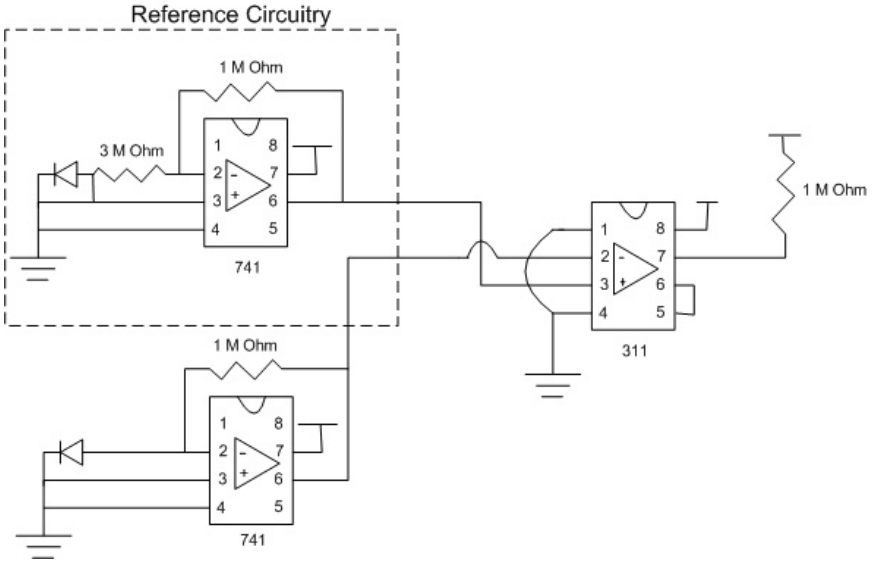


Figure 5: Target Sensing Circuitry

Schematics

We used the circuit in Figure 4 to sense when the target had been hit and respond with a TTL high level. We originally obtained the circuitry from Zehao Chang and Ben Schmidel's final E155 project entitled "The Laser Target Game" from 2000. (http://odin.ac.hmc.edu/~harris/class/e155/projects00/lasertargetgame.pdf) They included the reference detector to sense ambient light and hold its output at a higher voltage. This signal could then be sent to a comparator along with the signal from a target sensor which would output a TTL high if the target sensor had been hit and would output a TTL low in ambient light. A comparator is simply a high gain differential amplifier that goes into positive or negative saturation depending on the relationship of the inputs. The added 3 M Ohm resister was chosen to raise the output level of the reference circuitry to about 3 V. Since the target circuitry produces 2 V when there laser is not pointed on it and about 4 V when the laser is touching it, the 3 V produced by the reference circuitry will be low enough to trigger the target when 4 V is sent to the comparator yet high enough to output a TTL low from the comparator when it receives 2 V.

In the original circuitry from Chang and Schmidel, the output of the comparator (pin 7) had a resistor tied to ground. When we tested the circuitry we found that this did not work. This is because the 311 is an open collector. The manufacturers pull the output low but leave the output floating high so that multiple comparators wired together will produce a wired OR. This means that if any of the comparators are set high, the output of them wired together will also output a high. We also tied the balance pins (5 and 6) of the comparator together.

Each target is surrounded by four LEDs tied in parallel as shown in Figure 6.

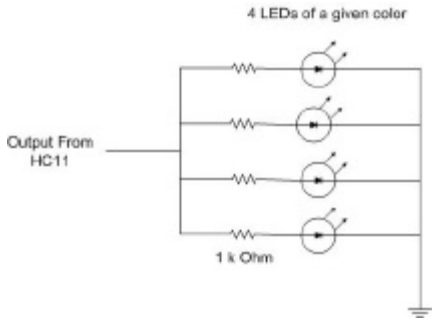


Figure 6: LED multiplexing

Microcontroller Design

On the highest level, the HC11 logic is very simple. It simply needs to seed a table in memory with a sequence of pseudo-random numbers, and then loop through increasingly large numbers of them, first demonstrating what to do by sending values out Port B on the lowest four bits, and then waiting for input on the lowest four bits of Port E and responding accordingly on bits 4 through 6 of Port B. Each time values go out to the lowest four bits of Port B to illuminate an LED, it also outputs a square wave of a given on the fourth Output Compare (OC4). Each of these sections however is fairly complicated.

The pseudo-random sequence comes from a fairly standard formula. Given a seed value, the next value is determined by: $R_{n+1} = (a*R_n+c) \bmod m$. In our case this is fairly simple since first of all, the "mod *m*" is taken care of automatically in the 8-bit memory of the HC11; m is 256 in this case. According to "The Art of Computer Programming" by Knuth, the values of *a* and *c* determine the length of the pseudo-random sequence you get out of such a formula. We chose values for *a* and *c* according to his recommendations to get a sequence of longest length before repetition begins. We used 17 and 57. Without going into details, these numbers need to be relatively prime, and the first needs to be one larger than a multiple of 4.

Timing in a real-time computing environment such as this is of highest importance. Sometimes in our code, we would like to wait for specified periods of time. For example, when we turn on a light and play a sound for the user, they probably want to see and hear if for more than a few microseconds. For this, we have simply borrowed some of our previous code from the traffic light control lab.

We keep all light signals in memory one-hot encoded to ease comparisons and loops, but it turns out that to generate a more random sequence of lights, we needed to look at each two-bit pair in each random byte rather than simply chopping them down to two bits and using the lowest order bits each time. The latter method generates sequences of length four instead of 256. This produces the need for another function to take a packed representation of the bits and produce a one-hot representation to actually store in

memory. This function is fairly trivial, but since it is used many times, it gets its own space.

Whenever a light turns on, for demonstration purposes, or in response to user input, a corresponding sound will also play. These sounds come from the HC11 producing a frequency-modulated signal sent to a sound system. The sound system consists of an inverting op-amp amplifier to make the sound louder and a capacitor tied to ground on the output to turn the square wave into a smoother triangle wave making the sound less harsh to the ear. We tied the output of these to a small speaker to actually produce the sound. The code to produce this input wave is pretty much taken right out of the solution for lab 7 but modified for our purposes where the high and low times are identical and those lengths are hard-coded to correspond with which light is also on rather than adjusting based on input voltage.

The most interesting part of the HC11 code is where it actually demonstrates the sequence of lights and waits for input. For the demonstration, it loops though the known number of bytes and outputs sound and light for each one for a set period of time. Once the demonstration is over, it starts the loop over again and waits for input on Port E. If the input is all zeros, it knows that the user just has not input anything yet. It also checks to see if the input is the same as the last input to prevent the possibility of a double hit. As soon as it is something other than zero or the last value, it checks if the input matches what it was expecting and sends output accordingly. If it was not correct, it simply sends an ACK without correct or increment. If it was correct but it is not done with the current loop yet, it will send an ACK with correct high. If it is not only correct, but the user has completed a full sequence of correct inputs and it is about to demo the next one, it also sends back the increment signal telling the FPGA to increase the users score.

To abstract this logic of which signals to send from the looping logic, we have created a sendFPGA function that looks at input from the FPGA and expected input and creates the appropriate signals to output: ACK, correct, and increment. The correct and/or increment signals must be sent before the ACK signal to prevent a setup time violation on the flip-flops that read this data. If signals are sent simultaneously, the result out of the flip-flops remains low regardless of the values of correct and increment.

Function Name	**Function Purpose**
main	General program flow
random	Create table of randomness
demo	Demonstrate light sequence
wait	Wait for input
sendFPGA	Send results to the FPGA
sound	Play one of about 5 sounds
delay	Wait for some period of time
onehot	Turn packed bits into one hot bits

FPGA Design

The main reason we originally thought we would need this piece of hardware is to control the input from the photodiodes since the HC11 is not able to constantly monitor it's input ports. The FPGA takes the photodiodes as input and sends those as output to the HC11. It also has three inputs collecting data from the HC11. In addition to these internal communication channels, the FPGA keeps score and sends that as output to the seven-segment display unit along with a clocked signal to control the seven-segment display (newclk). These top-level communication channels are expressed more clearly in graphics in the Appendix.

After some testing, it became apparent that complicated control of the laser sensor input is not necessary. In the end, all the FPGA does with the input signals from the lasers is send them right back out the other side. The meat of the FPGA now consists of listening to the HC11's signals and incrementing the score or stopping when asked to. The three signals are ACK, correct, and increment. ACK acts almost like a clock signal indicating to the FPGA that it should read the input. When that happens, if correct is high, the user was correct, and if increment was high, it was also the end of a sequence and the user gets another point. In hindsight, we really only needed one of these signals: increment. The other two remain from when the design was more complicated. In a simpler design, the flip-flops would be triggered by the increment signal and the score would be incremented every time it happened. This would also simplify the sendFPGA function in the HC11.

Results

The final product works quite well. The speed of the lights greatly contributes to the enjoyment of the game. Slow lights (one second) make it boring, and fast lights (less then half a second) make it more exciting. Initial user testing prompted us to speed things up to this faster speed. Of all the problems however, this was trivial. We ran into a few tough situations in the process of creating this game.

The most recent problems have arisen from loose wiring issues. Sometimes if you handle the game board too roughly, things readjust themselves and the game behaves strangely. This sort of problem actually prevented us from presenting on the final day until the very end when everybody was about to leave. Physical protection and securing of complicated wiring could prevent this type of issue, but this is not the focus of the project.

Another difficult problem was sending two signals simultaneously from the HC11 to the FPGA causing setup time violations on the flip-flops. If one signal is the clock and the other is the data for the same flip-flop, you need to send the data shortly before the clock to ensure the propagation of the new value. This caused scoring to not work for a while at full speed while manually setting signal values worked fine. This sort of timing issue can be difficult to debug sometimes.

Two main systems of our design require analog electronics: the sound system, and the laser sensing system. These required exploration of concepts not covered in this course. Genevieve did the laser sensors and Nick did the sound system. Genevieve had problems because the schematics she used did not work as advertised and required some adjusting. Nick had trouble with the sound system because even though sound came through without any additional hardware, more power was needed to make it louder. The original amplifier circuits he built amplified voltage instead of current, which did not really help. Noah Philips helped with the final sound system design, which can be found in the appendix.

The main thing different between the proposal and final project is the flow of laser sensor data. Originally, we thought the HC11 would be too slow to pick up on quickly changing input. For this reason, the FPGA would need to relay the info until the HC11 sent an acknowledgement. It turns out this is not needed, but the signals still pass

through the FPGA to remain within specifications from the proposal. Additionally, the FPGA/HC11 communication regarding correct answers has some wasted bits. These were originally needed to convey more information that we ended up actually needing. Without straying from within specifications, we could have removed nine wires from between the play board and the FPGA, four each way for sensor data and one extra from the three communication bits.

References

311 Datasheet: <http://www.national.com/ds/LM/LM111.pdf>.

741 Datasheet: <http://www.national.com/ds/LM/LM741.pdf>.

Horowitz & Hill. The Art of Electronics. Cambridge UP, Cambridge 2001.

Knuth, Donald. *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*

Laser Target Game:
<http://odin.ac.hmc.edu/~harris/class/e155/projects00/lasertargetgame.pdf>.

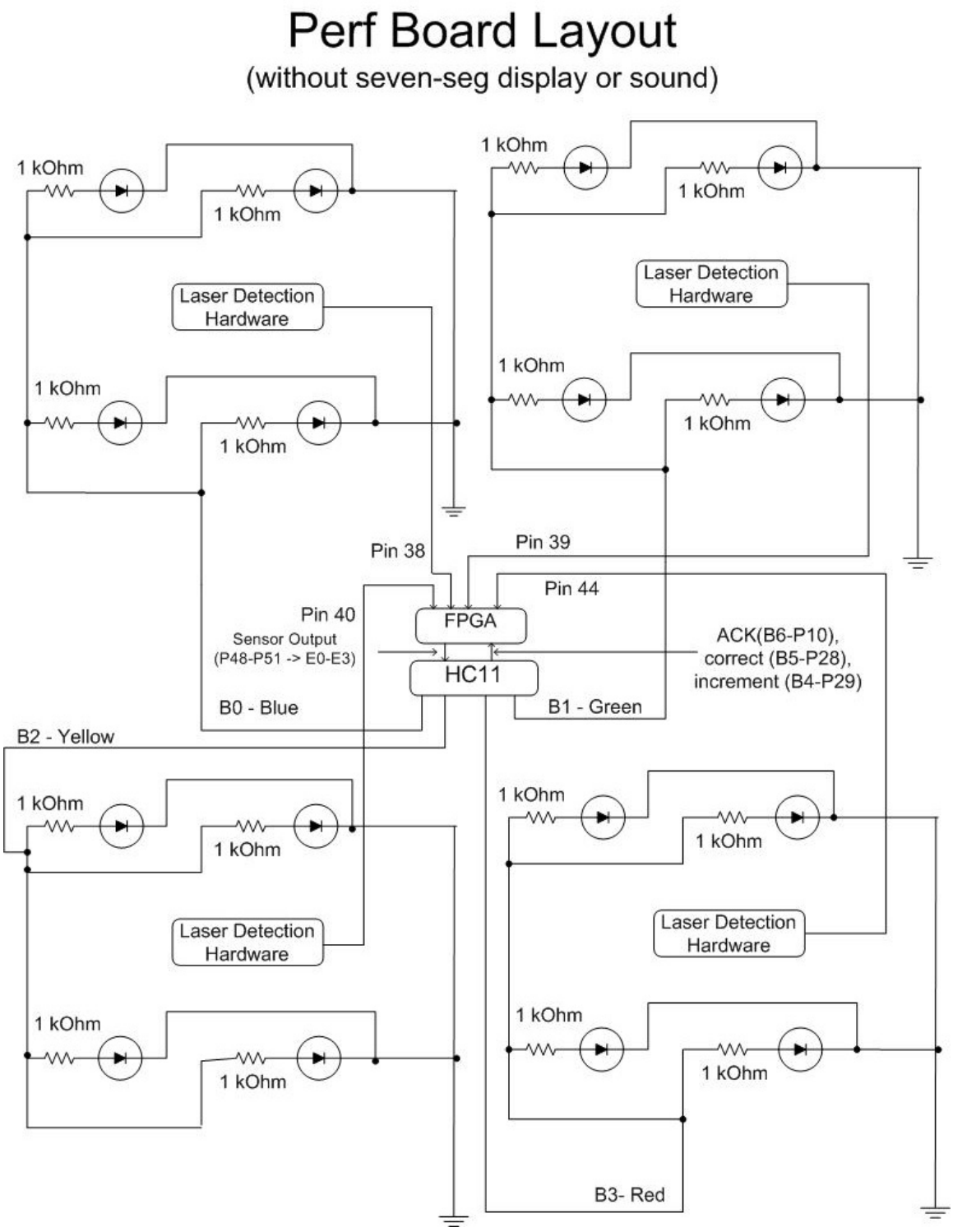
Photodiode Datasheet: <http://www.optekinc.com/pdf/OP950.pdf>.

Professor Harris Lab 7 Solutions

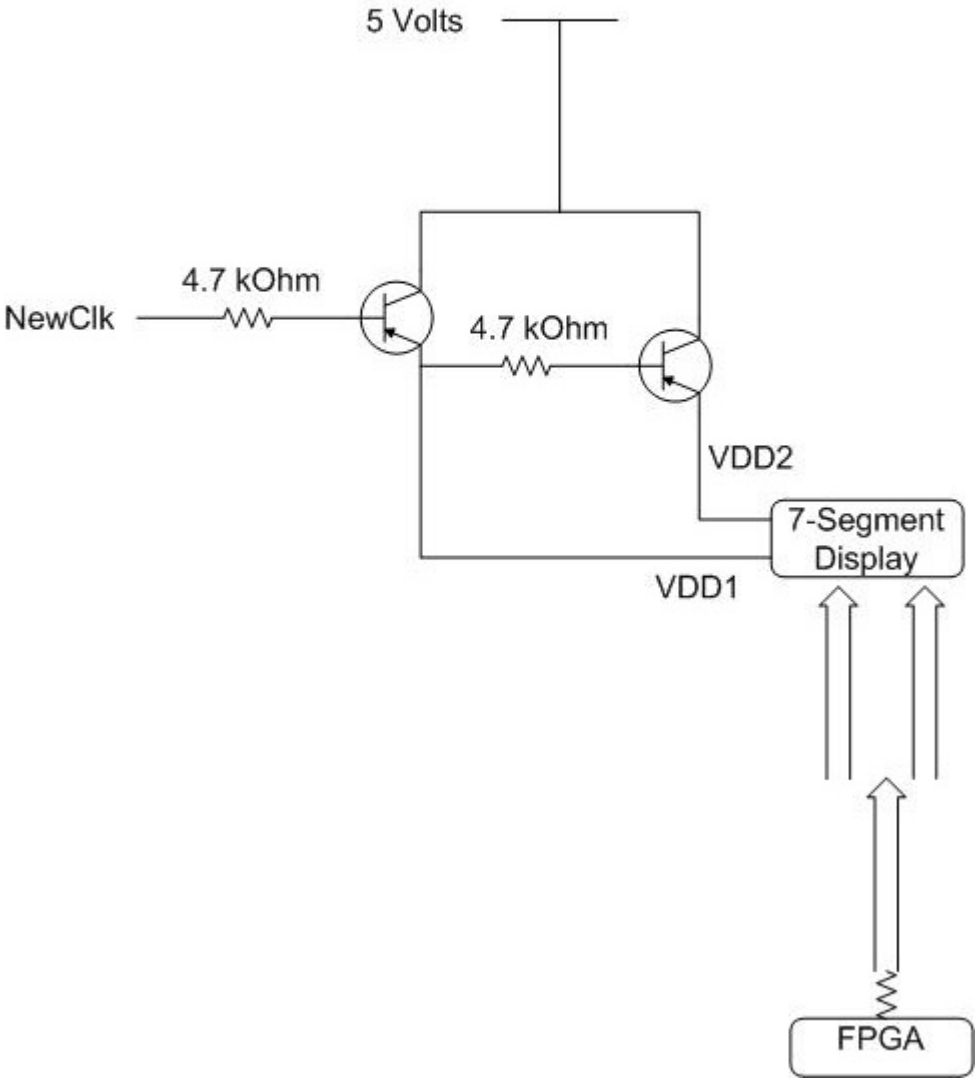
Parts List

Part	**Source**	**Vendor Part #**	**Price**
Photodiode	Arrow Electronics	OP950	$27.08
PC Predrilled Proto Board	Mar Vac	ODY9400T	$34.97
Low Current LEDs	Mar Vac	LIN B4300H5LC	$5.16
Speaker	Electronics Lab		
741 Op Amp	Electronics Lab		
311 Comparator	Electronics Lab		
Laser Pointer	Genevieve's Room		

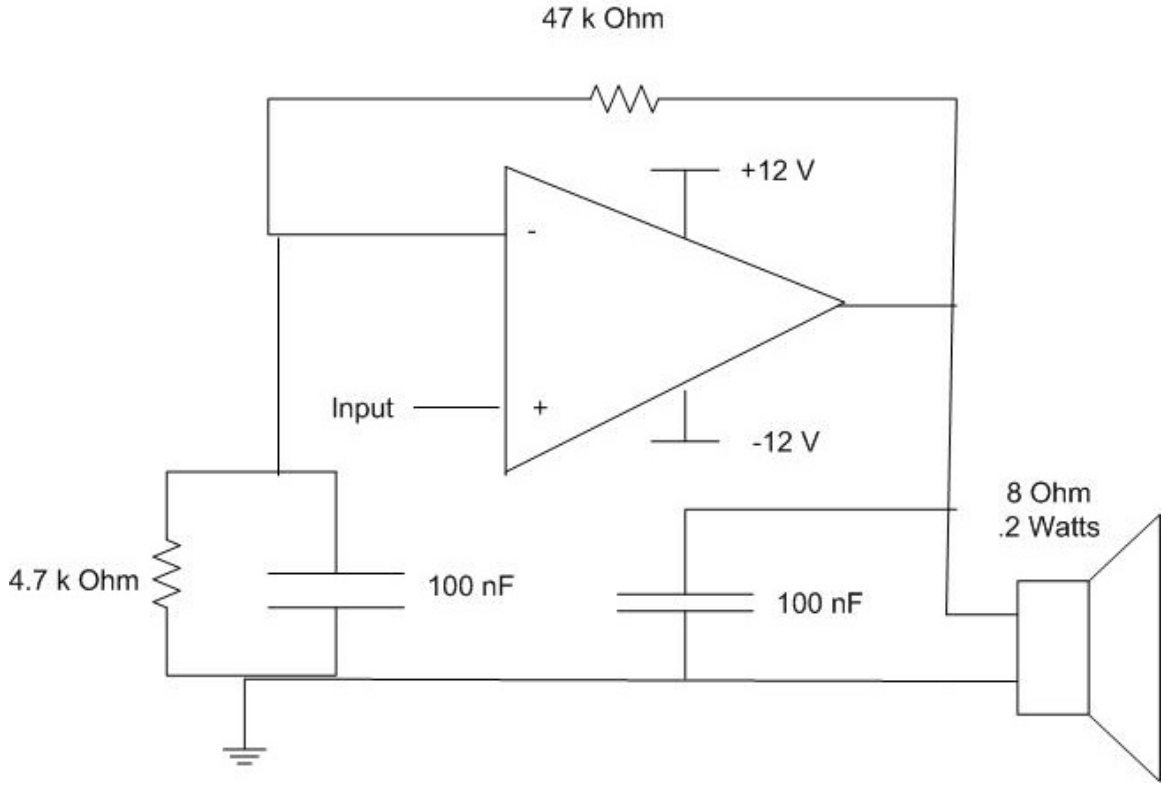
Appendix: Code and Graphics



7 Segment Display



Sound System



```
/* laserSimon.v
*
* November-Decenber 2002
*
* Nicolas_Hertl@hmc.edu
* Genevieve_Breed@hmc.edu
*
* This is the top level layout of Laser Simon game, our Final Project for E155
* It controls a seven-segment display with the score and collects the data for
* what to display from a secondary module (simon)
*/

module laserSimon(clk,reset,sensorsIN,ACK,correct,increment,sensorsOUT,seg,newclk);
            input clk;
            input reset;
   input [3:0] sensorsIN;
   input ACK;
   input correct;
            input increment;
   output [3:0] sensorsOUT;
   output [6:0] seg;
            output newclk;

           wire [3:0] displaysig;
           wire [3:0] rightsig;
           wire [3:0] leftsig;


           counter mycounter(clk,reset,newclk);
                                                        // Slow down the clock

           // module to take in sensorsIN, ACK, correct, and output sensorsOUT, leftsig, rightsig
           simon mysimon(reset,sensorsIN,ACK,correct,increment,sensorsOUT,leftsig,rightsig);

           mux2 mymux2(leftsig,rightsig,newclk,displaysig);
                                // Select which input signal to use

           sevenseg mysevenseg(displaysig,seg);
                                                        // Drive the sevensegment display with selected
signal


endmodule
```

```
/*
* November-Decenber 2002
*
* Nicolas_Hertl@hmc.edu
* Genevieve_Breed@hmc.edu
*
* This is the heart of the FPGA code for the Laser Simon game.  It keeps track of the score and increments
it
* whenever the HC11 tells it to.  The sensorsOUT was going to be controled more closely but it turns out
that's
* not needed.  sensorsOUT equals sensorsIN.  Debouncing isn't needed here.
*
* This could have been done much better...
*
* always @(posedge increment)
*                       if(gameOver)
*                               score = '00'
*                       else
*                               score++
*
* The additional signals remain from a time when we thought it would need to be more complicated.
*/

module simon(reset,sensorsIN,ACK,correct,increment,sensorsOUT,leftsig,rightsig);
   input reset;
   input [3:0] sensorsIN;
   input ACK;
   input correct;
          input increment;
   output [3:0] sensorsOUT;
   output [3:0] leftsig;
   output [3:0] rightsig;

          parameter s_DOING = 2'b1;
          parameter s_DONE = 2'b0;

          reg state;
          reg nextstate;

          reg [3:0] leftsig;
          reg [3:0] nextleftsig;

          reg [3:0] rightsig;
          reg [3:0] nextrightsig;

          parameter ZERO = 4'b0000;
          parameter NINE = 4'b1001;

          assign sensorsOUT = sensorsIN;

          // use ACK as the clock here since that's when I want to change state

          always @(posedge ACK or posedge reset)
                  if(reset)
                          begin
                                  state <= s_DOING;
```

```
                leftsig <= ZERO;
                rightsig <= ZERO;
            end
        else
            begin
                state <= nextstate;
                leftsig <= nextleftsig;
                rightsig <= nextrightsig;
            end

    always @(state or correct or increment or rightsig or leftsig or ACK)
        if(state == s_DONE)             // you've already lost
            begin
                nextstate <= state;
                nextleftsig <= leftsig;
                nextrightsig <= rightsig;
            end
        else if(ACK & ~correct)    // you were wrong
            begin
                nextstate <= s_DONE;
                nextleftsig <= leftsig;
                nextrightsig <= rightsig;
            end
        else if(ACK & correct)     // you were right
            begin
                nextstate <= state;
                if(increment)               // you get a point
                    if(rightsig != NINE)
                        begin
                            nextrightsig <= rightsig + 1;
                            nextleftsig <= leftsig;
                          end
                    else
                        begin
                            nextrightsig <= ZERO;
                            nextleftsig <= leftsig + 1;
                        end
                else                                    // you were right, but no
point yet
                    begin
                        nextleftsig <= leftsig;
                        nextrightsig <= rightsig;
                    end
            end
        else
            begin
                nextstate <= state;
                nextleftsig <= leftsig;
                nextrightsig <= rightsig;
            end
endmodule
```

```
/*
* Fall 2002 - E155
*
* Nicolas_Hertl@hmc.edu
*
* From Lab 3, but used for Final Project. "Laser Simon"
*/

module counter(clk,reset,newclk);
   input clk;
            input reset;
   output newclk;


            /*
            The purpose here is to just count to a big number in binary.
            Then assign the highest ordered bit to the newclk output.
            This makes it so newclk is high for a while and then low for a while,
            but just takes way longer to toggle back and forth.  Lower frequency.
            */

           reg [9:0] count;

           always @(posedge clk or posedge reset)
                     if(reset) count <= 0;
             else count <= count + 1;

           assign newclk = count[9];

endmodule
```

```
/*
* Fall 2002 - E155
*
* Nicolas_Hertl@hmc.edu
*
* From Lab 3, but used for Final Project. "Laser Simon"
*
* Taken right out of the lab notebook
*/

module mux2(s1,s2,select,out);
   input [3:0] s1;
   input [3:0] s2;
   input select;
   output [3:0] out;

         tristate t1(s1,select,out);
         tristate t2(s2,~select,out);

endmodule
```

```
/*
* Fall 2002 - E155
*
* Nicolas_Hertl@hmc.edu
*
* From Lab 3, but used for Final Project. "Laser Simon"
*
* Taken right out of the lab notebook
*/

module tristate(a,en,y);
   input [3:0] a;
   input en;
   output [3:0] y;

         assign y = en ? a : 4'bz;

endmodule
```

```
/*
* Fall 2002 - E155
*
* Nicolas_Hertl@hmc.edu
*
* From Lab 2, but used for Final Project. "Laser Simon"
*
*/

module sevenseg(s,seg);
   input [3:0] s;
   output [6:0] seg;

            // I saw a much better way to do this in class, but I'm just using the one from lab 2 for consistancy.

            assign seg[0] = ~(s[1]&~s[0] | s[1]&~s[3] | s[1]&s[2] | ~s[0]&~s[2] | s[0]&s[2]&~s[3] |
~s[0]&~s[1]&s[3] | ~s[1]&s[3]&~s[2]);
            assign seg[1] = s[2]&s[1]&~s[0] | s[3]&s[1]&s[0] | ~s[3]&s[2]&~s[1]&s[0] |
s[3]&s[2]&~s[1]&~s[0];
            assign seg[2] = ~(s[3]&~s[2] | ~s[1]&s[0] | ~s[3]&s[2] | ~s[1]&~s[3] | s[0]&~s[3]);
            assign seg[3] = s[2]&s[0]&s[1] | ~s[0]&~s[1]&s[2]&~s[3] | s[0]&~s[1]&~s[2]&~s[3] |
~s[0]&s[1]&~s[2]&s[3];
            assign seg[4] = ~(s[1]&~s[0] | s[3]&s[2] | s[3]&s[1] | ~s[0]&~s[2]);
            assign seg[5] = s[0]&~s[2]&~s[3] | s[1]&~s[2]&~s[3] | s[1]&~s[2]&~s[3] | ~s[3]&s[0]&s[1] |
s[3]&s[2]&~s[1]&s[0];
            assign seg[6] = ~(s[1]&~s[0] | s[1]&~s[2] | s[1]&s[3] | s[0]&s[3] | ~s[1]&~s[3]&s[2] |
s[3]&~s[2]);

endmodule
```

```
* final.asm
*
* November-Decenber 2002
*
* Nicolas_Hertl@hmc.edu
* Genevieve_Breed@hmc.edu
*
* This program controls "Laser Simon" game, our final project in E155
* It produces a pseudo random sequence, flashes the LEDs in the correct
* pattern and whenever one is hit, checks to see if the hit sensor is the
* correct sensor and tells the FPGA when the hit is correct and when
* the score should be incremented.  Finally it produces the sounds for
* our speaker system
* It starts the game and ends the game.


* Port Addresses
REG     EQU     $1000   * base address of registers
PORTA   EQU     $1000    * square wave output on A[4]
PORTB EQU       $1004    * LEDs and FPGA signals out
PORTE   EQU     $100A   * FPGA input for Lazer Sensors

* Timer Addresses
TCNTH EQU       $100E    * not used actually...
TCNTL  EQU      $100F    * random seed.

* Constants for interrupt stuff for sound
TOC4    EQU     $101C
TCTL1   EQU     $1020
TMSK1   EQU     $1022
TFLG1   EQU     $1023

* Constants for color encoding
* pCOLOR is the packed encoding
* hCOLOR is the one Hot encoding
pRED EQU %00000000
pYELLOW EQU %00000001
pGREEN EQU %00000010
pBLUE EQU %00000011
hRED EQU %00001000
hYELLOW EQU %00000100
hGREEN EQU %00000010
hBLUE EQU %00000001
hWRONG EQU %00000000

* Constants for I/O
ACK     EQU %01000000* ACK bit
Cbits     EQU %00100000* Correct bits
Ibits       EQU %00110000* Increment bits

* User variables
STOP    EQU     $0025    * Which byte of lights to stop on when looping
LASTL  EQU      $0027    * The last Lazer input.  Ignore duplicates.


START  EQU      $D200   * Table starts at this address
```

```
LENGTH          EQU     $0064    * Table is this long
TABLE EQU       $0000    * This isn't really the start of the table... but since table is in high memory, it
needs to be like this.
LASTT  EQU      $0001    * One past the current entry in the TABLE

* Masks
OC4F    EQU     %00010000
BIT2    EQU     %00000100
TWOMASK EQU             %00000011
FOURMASK EQU            %00001111

* Local variables
        ORG     $20
PWMLO   FDB     $0080   * Low time for pulse width modulation
PWMHI   FDB     $0080   * High time for pulse width modulation

* Interrupt vector
        ORG     $00D6   * Output compare 4 Interrupt service routine
        JMP     OC4ISR  * hardwired to $00D6, where we put jump to our routine

*****************************************************************************************
*********************
*
* Main()
*
*       STOP = 0
*       LASTL = 0
*       random();
*       for(i=0;i!=100;i++)
*               demo(i)
*               wait(i)
*               delay()
*
* First zero out some memory storage locations, then fill the random table.
* The actual program is to loop through the 100 random light values, first demoing, and then waiting
* for input from the user up to that point i, getting progressively harder.  Between each wait period
* and the next demo, it waits about a second to more clearly indicate the start of a new demo cycle.
*
*****************************************************************************************
*********************

* Main Program
        ORG     $D000

* Initialize
        LDAA    #$00    * load zero
        STAA    STOP    * preload bits with zero so things work from the start
        STAA    LASTL

        BSR     random

        LDX     #$D200 * loop through all states in memory starting at i = 0
byte:   BSR     demo    * demonstrate what to do
        BSR     wait    * wait for some input from the FPGA
        BSR     Fdelay  * give it a short rest so user can see some blank time before next sequence
        INX             * i++
```

```
        CPX     #$D264
        BNE     byte    * i==100?keeplooping:fallthrough

        SWI             * done :-)


*****************************************************************************************
**********************
* This subroutine just creates the table of randomness
*
* random(0) = TCNTL
* random(n+1) = (random(n)*17 + 57) % 256
*
* for(i=100;i!=0;i--)
*       TABLE[100-i] = random(100-i)
*
* 17 because it's one greater than a multiple of 4 and that's
* needed because 256 (memory width) is also a multiple of 4
* Furthermore, 17 and 256 are relatively prime.  This is good.
* 57 is kinda just pulled outa thin air, but it's prime, and
* seems to work.
*****************************************************************************************
**********************

random: PSHA
        PSHB
        PSHX
        PSHY
        LDX     #START+#LENGTH      * start counting from the end of the table
        LDAA    #$00
        STAA    LASTT,X             * zero out the spot for comparisons to keep first value more
random
        LDAA    TCNTL           * lower 8 bits of Timer to seed randomness

        LDY     #$0004          * use Y to count which set of bits to use

load:   DEX                     * coming in here, we expect A has the last random value
retry:  LDAB    #$11            * multiplier for randomness ($11 = 17)
        DEY                     * change which bits to look at
        BNE     skip
        LDY     #$0004          * only reload if Y==0
skip:   MUL                     * multiply random(i)*17 and lead result in register D
        ADDB    #$37            * additive constant for randomness
        TBA                     * remember last random value for next calculation

* since we're generating 8 bit randomness, we need to pull from all 8 bits rather than always the low two.
* using the low two each time would produce very poor randomness equivalent to two bit randomness.

        CMPY    #$0001
        BEQ     one
        CMPY    #$0002
        BEQ     two
        CMPY    #$0003
        BEQ     three
        CMPY    #$0004
        BEQ     out     * don't rotate at all
```

```
one:    RORB            * rotate one set of two bits
        RORB
        BRA     out
two:    RORB            * rotate two sets
        RORB
        RORB
        RORB
        BRA     out
three:  RORB            * rotate three sets
        RORB
        RORB
        RORB
        RORB
        RORB
out:    ANDB    #TWOMASK    * only look at the lowest 2 bits
        BSR     onehot      * one hot encode the randomness
        CMPB    LASTT,X             * See if it's the same as the last one
        BEQ     retry       * Don't load it. It was the same... retry
        STAB    TABLE,X             * store randomness
        CPX     #START              * keep going as long as you haven't hit the botom yet
        BNE     load
        PULY
        PULX
        PULB
        PULA
        RTS

*****************************************************************************************
**********************
* HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK
*
* the code is so long that branches are out of range.  This allows for two BSRs so it will work.
*****************************************************************************************
**********************
demo:   BSR     Rdemo
        RTS
Fdelay: BSR     OFdelay
        RTS

*****************************************************************************************
**********************
* wait(byte) takes argument in register X, and handles I/O for that one.
*
*       for(int i=0; i!=howfar; i++)
*               waitforFPGA(i)
*               sendFPGA(input)
*               sound(input)
*
* For each of the light values up until X, wait for input from the FPGA and send back results to the FPGA
* as well as playing a sound for long enough that the user hears it and knows they hit the sensor.
*
*****************************************************************************************
**********************

wait:   PSHA
```

```
        PSHB
        PSHY

        LDY     #START-1        * start counter at beginning of table - i=0
        STX     STOP            * remember where to stop
wloop:  INY                     * i++
        LDAB    TABLE,Y                 * get what value we are expecting

loadE:  LDAA    PORTE           * get input from FPGA
        ANDA    #FOURMASK       * only look at the lowest four bits (the lazer inputs)
        BEQ     loadE           * keep waiting if it was zero
        CMPA    LASTL           * Is this new signal the same as the last one we got?
        BEQ     loadE           * if so, keep waiting

        STAA    PORTB           * light up what the shot
        STAA    LASTL           * remember this as the last Laser input

        CBA                     * did we get the correct input?
        BEQ     good            * yes, they were right.
        LDAA    #$00            * no, they were wrong... play "wrong" sound

good:   BSR     sendFPGA        * send info to FPGA, A is the data,

        BSR     Fsound          * play the sound

        CMPA    #$00            * were they wrong?
        BNE     wgo             * no... skip this part and go on
        LDAA    #%00001111      * yes, wrong, turn on all lights
        STAA    PORTB           * Light them up cause game is over
        SWI                     * GAME OVER

wgo:    CPY     STOP            * i==howfar?
        BNE     wloop           * no?... keep looping. -  yes?... exit
        PULY
        PULB
        PULA
        RTS


*******************************************************************************************
*******************
*
* onehot(packedColor&)
*
* this takes a bit-packed representation of color and one-hots it for output - pass by reference
*
* This function doesn't save the original value of B, it overwrites with new value.
*
*******************************************************************************************
*******************

onehot: CMPB    #pRED           * Determine which packed color it is and branch accordingly
        BEQ     oRed
        CMPB    #pYELLOW
        BEQ     oYellow
        CMPB    #pGREEN
```

```
        BEQ     oGreen
        CMPB    #pBLUE
        BEQ     oBlue

oRed:   LDAB    #%00001000          * Load the correct onehot value and exit
        BRA     oDone
oYellow: LDAB   #%00000100
        BRA     oDone
oGreen: LDAB    #%00000010
        BRA     oDone
oBlue:  LDAB    #%00000001
        BRA     oDone
oDone:  RTS


*******************************************************************************************
**********
* HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK
*
* the code is so long that branches are out of range.  This allows for two BSRs so it will work.
*******************************************************************************************
**********

OFdelay:        BSR     delay
                RTS
Fsound:         BSR     sound
                RTS

*******************************************************************************************
**********
* demo(byte) takes argument in register X, and demonstrates what the user should do that far
*
* for(i=0;i!=X;i++)
*       lightup(TABLE[i])
*       sound(TABLE[i])
*
*******************************************************************************************
**********

Rdemo:  PSHA
        PSHY

        LDY     #START-1        * start counter at beginning of table - (i=0)
        STX     STOP            * this is how far we're demoing - remember X for comparison
dloop:  INY                     * i++
        LDAA    TABLE,Y                 * get the thing to demo (TABLE[i])
        STAA    PORTB           * Show the light (lightup(TABLE[i]))
        BSR     sound           * play the sound (sound(TABLE[i]))
        CPY     STOP            * are we done yet? (i==X?)
        BNE     dloop           * if not, keep looping - if so, exit
        PULY
        PULA
        RTS
```

```
*******************************************************************************************
*********************
*
* sendFPGA(signal,i,END)
*
*       if(signal==WRONG)
*               Send(WRONG-ACK)
*       else if(i==END)
*               Send(Increment-ACK)
*               LAST_LASER_INPUT = 0
*       else
*               Send(Correct-ACK)
*
* Send(WHICH_ACK)
*       tmp = ACK
*       tmp = tmp | current_light_values
*       PORTB = tmp
*       tmp = tmp | WHICH_ACK
*       PORTB = tmp
*
*
* This takes input in Register A, either 0000, 0001, 0010, 0100, or 1000,
* which table element we're on in Register Y,
* as well as in memory STOP value as to where the END of this cycle is
*
* depending on whether you were right or wrong and if we're done with that sequence or not, it needs to
* write different signals to the FPGA.
*
* Note:  The above C code and the below assemble differ in one major detail.  The C code simple sends
* out the appropriate ACK directly, but the assembly does it in two steps.  It first sets up the right
* type of ACK bits as to whether it's correct or increment, and then outputs the ACK signal as well.
* The reason for this was a setup time violation for the flip flops.  If the flip flops trigger at the
* exact same time as the values change, they will miss and every signal looks like a WRONG ACK.
Setting
* up the signals before sending off the ACK trigger allows the FPGA to operate correctly.
*
*******************************************************************************************
*********************
sendFPGA:       PSHB

                CMPA    #hWRONG                 * are you wrong?
                BNE     right           * if not... skip this part
                LDAB    #ACK            * send Wrong ACK to FPGA
                STAB    PORTB           * send info to FPGA.  Lights don't matter... they lost anyway
                BRA     sfEnd           * get outa here
right:          CPY     STOP            * are we on the last byte yet?
                BNE     correct         * ok, not last byte... send correct, no increment
                LDAB    #$00
                STAB    LASTL           * zero out the last lazer thing since we're done with a cycle
                LDAB    #Ibits          * load increment ACK bits
                BRA     FPGAend                 * get outa here
correct:  LDAB  #Cbits          * load correct ACK bits

FPGAend:        ORAB    PORTB           * remember the lights
                STAB    PORTB           * output the new signals
                ORAB    #ACK            * add on the ACK signal
```

```
                STAB    PORTB           * re-output with ACK to trigger the read
                ANDB    #FOURMASK       * block out the ACK bits
                STAB    PORTB           * keep the lights on
sfEnd:          PULB
                RTS


*****************************************************************************************
*********************
*****************************************************************************************
*********************
*
* Stuff to do the pulse width modulation
*
*****************************************************************************************
*********************
*****************************************************************************************
*********************

*****************************************************************************************
*********************
*
* delay() is a function that delays for just less than a second using busy waiting
*
* void delayonesecond()
*       for(int j=##; j!=0; j--)
*               for(int k=#$7F; k!=0; i--)
*
*****************************************************************************************
*********************

delay:  PSHA            * remember A
        PSHX            * remember X
        LDX     #$0BED* put magic one second delay number into A (j)
dOuter: DEX             * j--
        CPX     #$0000  * j==0?
        BEQ     dDone   * j==0
        LDAA    #$7F    * k=#$7F   if it's 3F, it's more like a half second
dInner: DECA            * k--
        BNE     dInner  * k!=0
        BRA     dOuter  * k==0
dDone:  PULX            * restore X
        PULA            * restore A
        RTS             * return from subroutine

*****************************************************************************************
*********************
* sound(A)
*
* This function sends a square wave off to the speaker system based upon which signal it gets in reg A
*
*****************************************************************************************
*********************

sound:  PSHA
        PSHB
```

```
            PSHX
            PSHY

*compare value to red
            CMPA   #hRED
        BEQ     red
*compare value to yellow
            CMPA   #hYELLOW
        BEQ     yellow
*compare value to green
            CMPA   #hGREEN
        BEQ     green
*compare value to blue
            CMPA   #hBLUE
        BEQ     blue
*compare value to "wrong"
            CMPA   #hWRONG
        BEQ     wrong

* test samples to determine reasonable frequency values.
* $0080 -> 7810 Hz          (very piercing and annoying)
* $0800 -> 500 Hz           (sound good)
* $8000 -> 30 Hz (scratchy, below the good-sounding range of our speaker)
*

red:    LDD     #$0800
        STD     PWMHI
        STD     PWMLO
        BRA             sEnd
yellow: LDD     #$0B00
        STD     PWMHI
        STD     PWMLO
        BRA             sEnd
green:  LDD     #$0900
        STD     PWMHI
        STD     PWMLO
        BRA             sEnd
blue:   LDD     #$0A00
        STD     PWMHI
        STD     PWMLO
            BRA       sEnd
wrong:  LDD     #$1800
        STD     PWMHI
        STD     PWMLO

sEnd:
* Initialize interrupts
            LDAA    #%00001000      * Set OC2 to set output pin low
            STAA    TCTL1
            LDAA    #%00010000      * Enable OC4 interrupt
            STAA    TMSK1

            CLI                 * turn on intrerrupts
            BSR     delay       * wait for one second for the sound to play.
            SEI                 * turn off interrupts before we leave the sound function
```

```
* De-Initialize interrupts to prevent funny noises
* If interrupt ends with OC2 set high, we get really bad noise through speaker
* For this reason, we force it low after each sound period

        LDAA   #$FF     * Set OC# to set output pins all low
        STAA   TCTL1

        CLR     PORTB * Turn lights off after sound is done
        PULY
        PULX
        PULB
        PULA
        RTS


*******************************************************************************************
*********************
* Interrupt service routine for OC4
*
* Taken directly from Lab 7 solutions written by Professor David Harris
*
*******************************************************************************************
*********************

OC4ISR  LDX    #REG
        BRCLR   TFLG1-REG,X OC4F RTOC4     * Ignore other interrupts
        LDAA    #OC4F                      * Store 1 to clear flag
        STAA    TFLG1-REG,X                * Zeros do nothing
        BRSET   TCTL1-REG,X BIT2 LASTHI
        BSET    TCTL1-REG,X BIT2           * set OC4 to set pin high
        LDD     TOC4-REG,X                 * Increment output compare time
        ADDD    PWMLO
        STD     TOC4-REG,X
        BRA     RTOC4

LASTHI  BCLR    TCTL1-REG,X BIT2           * set OC4 to set pin low
        LDD     TOC4-REG,X                 * Set wait time
        ADDD    PWMHI
        STD     TOC4-REG,X

RTOC4   RTI                                * Return from the interrupt
```