Mastermind: The Game
Final Project Report
December 12, 2002

Carl Larsen and William Berriel
Abstract:

The logic game mastermind involves one player creating a hidden sequence out of colored pins and another player making a limited number of guesses about what the sequence is. The player with the hidden sequence gives the guesser feedback on each guess, revealing, through the use of black and white pegs, if the guess featured any pins that exactly matched the color of the hidden sequence and whether the guess had pins that only matched the color of the hidden sequence. We implemented this game, with the sequence being a sequence of numbers, using the 68hc11 to play as the player with a hidden sequence. The FPGA receives input through a matrix keypad, and the system outputs to the player through a matrix of LCD's for the feedback on the guesses, and an LCD screen that displays all guesses.
**Introduction:**

Mastermind is a logic game that involves one player making a hidden sequence of colored pins and a second player trying to guess the sequence. The original version featured the second player responding to the first player's guesses with a series of black and white pins, with the number of pins in one color representing the number of pins in the guess that are the correct pin color in the correct position, and the other color representing the number of pins in the guess that are the right color but in the wrong position. The second player had a limited number of guesses, after which they would lose. For our project, we implemented a version of Mastermind using the FPGA and 68hc11. The hidden sequence would be a 4 digit sequence of numbers from zero to five. The input is through a nine key matrixed keypad. The system outputs using both a 16x2 character LCD screen and 6x8 LED matrix. The LCD displays the current guess as well as previous guesses, and will also reveal the hidden sequence if the player wins or loses. The LED's are set up in six rows of eight, with four green and four red per row. The red LED's represent the correct digit in the correct position while the green represent the correct digit in the wrong position. The position of the lit LED's does not correspond in any way to the where in the sequence the correct digits occur. The 68hc11 acts as the player with the hidden sequence, randomly assigning the digits from zero through five to the positions in the sequence. It also controls the LCD screen output, and the FPGA output. The FPGA takes in the input from the keypad, and multiplexes the output to the LED matrix.
New Hardware:

We used a 16x2 character LCD from the stockroom, which although may have been used by E155 students before, featured 2 pitfalls that might not have been documented earlier. This LCD is controlled by a standard Hitachi HD4780A00 controller, featuring 14 pins. Controlling the LCD was rather simple, but we had problems with 2 aspects. The first was that the controller needs to be given negative voltage for ground. Secondly, the auto startup routine, which runs whenever the LCD is powered up normally, prevents the controller from being set to 2 line mode. In order to fix this, we needed to slowly ramp up the +5V rail on the power supply that powered the LCD. This prevents the auto reset feature from being used by the LCD, allowing us to initialize it manually. We would like to thank Aaron Stratton who provided much needed assistance by helping us debug the initialization and reset sequences. In addition, for a complete overview on how to use an LCD display, please look at reference 3.

Although each part of the LED matrix is readily available, its operation may not be extremely obvious. It works just like the multiplexed seven-segment displays from our earlier labs, except individual LED's are used. It is multiplexed in six rows by eight columns. Each row is powered by a transistor which is controlled by the FPGA. The FPGA provides a voltage on the opposite side of the LED to mask the LED's, selecting which columns are lit. Each port has a 330Ω resistor wired to it, to lower the current through the LED's, limiting them to 15mA of current. Since the LED's run at a 1/6 duty cycle, it might be possible to lower this a bit to raise the brightness of the LED's, if they are not sufficiently bright.
Microcontroller Design:

The 68hc11 controls the game logic and LCD screen. It controls the LCD screen using modified code from “The Super Happy Fun Game”, written by Ari Moradi and Ryan Stuck in E155 in the Fall of 2000. Two functions exist to communicate to the LCD, WRITEC and WRITED. WRITEC writes a binary command from accumulator B to the LCD. WRITED writes the ASCII character from accumulator B to the LCD. Between commands and data, the 68hc11 waits for at least two miliseconds.

The FPGA raises an XIRQ whenever it has input from the keypad. The 68hc11 reads the input from Port E. It first determines if the input is a reset, which has priority and will always reset the game state. If not a reset, it determines the state that the game is in and acts accordingly. The game has three finite state machines: One that records whether it has determined a hidden sequence yet, one that stores whether the user has input a full guess, and one that stores the number of guesses that the user has made.

If the random sequence has not been fully generated, on a key press that is not reset the 68hc11 runs the random subroutine. This subroutine takes the least eight bits of the timer, which should be sufficiently random since the clock runs at eight megahertz, and overflows the eight bits in less than a millisecond. It multiplies the lower byte of the timer by six, generating a sixteen bit number with a value from 0x0000 to 0x05FF. The higher byte of the timer is then taken since, it is between zero and five, which is the range that we need. This method has a close to even distribution of numbers, with five having a 1.5% advantage. The random subroutine is not run after the random sequence is generated and the game is not reset.

After determining if it needs a to generate the hidden sequence or not,
the system handles the incoming number. By this point, since it was not a reset, the
digit must be between zero and five. Since the input must be valid, it is stored in the
proper place in the guest sequence.

Once we have the complete guess sequence input, we need to score the
sequence. Scoring the sequence consists of first determining the exact matches by
comparing each digit in the guess with each digit in the hidden sequence. For each set
of matching digits, the output to the FPGA gets another one in its high nibble. Thus if
none are correct, the high nibble is a 0x0, otherwise it's an 0xF. Once it has found the
exact matches, it finds the number of right digits in the wrong position. To do this,
each digit of the hidden sequence is compared to all four digits of the guess in
succession, until a match is found or all four have been compared. If a match is found,
the position in the guess is marked, the match is recorded, and the next digit in the
hidden sequence is compared to the guess, skipping over the marked digits. Once it
has compared each digit of the hidden sequence, it subtracts the number of correct
position digits from the number in the incorrect position and outputs the numbers as
one-hot encoded sequence corresponding to the LED's that will be on to the FPGA. It
holds this sequence on port c, raises the enable pin (a[6]) for a short time, shifting the
FPGA once, and then lowers enable. If after scoring, the guess is found to be a
winning guess, then the game enters the win state, printing the win message on the
LCD, and displaying the hidden sequence. If the six guesses have been used up and
the player has not won, the game enters the lose state and prints the lose message and
the hidden sequence on the LCD. It stays in the winning and losing states until reset is
pressed.
FPGA Design

The FPGA contains two independent modules, a user input module and a user feedback output module. Schematics of the breadboard circuits, block diagrams, and verilog code for these modules are in the appendix.

User Input Module:

The input module allows the user to input guesses for the mastermind game. It consists of a matrix keypad connected to the FPGA. The columns of the matrix keypad are polled by the FPGA, via a finite state machine, by sequentially setting each column low. If a button is pressed a short will occur in one of the rows of the keypad when the polling reaches the column of the pressed button. When it detects a short (i.e., a low value in the row input) the FPGA will then stop polling the columns and wait until the button is released and the short is gone. The combination of low column and low row values caused by the short on the matrix keypad are decoded on the FPGA to determine the binary value indicated by the particular key pressed. Our system uses the nine button located in the upper left of the key pad which are encoded as the following values:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Zero through five are used by the user to provide input for the game. Six, and the two sevens will be used to control game logic such as allowing the user to reset the game.

The binary value generated by the encoder is stored, until a new button press is detected, in a 3-bit asynchronously resetable flip-flop with enable. This flip-flop is enabled by an fsm that will output high for one cycle of the slow clock, or about 1 ms, each time a button is pressed. The following is a state transition diagram
of this fsm.

Transtions between states depend on whether the row input is all zero or not, in other
words whether the button is being pressed or not. The fsm will stay in state zero until
it detects a press at which point it will move to state 1, generating the enable signal,
and then move to state 2. It will stay in state 2 until the button is release at which
point it returns to state zero and is ready for another press. In all cases reset sends the
fsm back to state zero.

In addition to functioning as the enable for the flip-flop, the output of this fsm
is inverted, buffered in two 1-bit flip-flop in series, and then used as an interrupt
signal to tell the HC11 that new user input is ready. Buffering through two flip-flops
ensure that the interrupt signal arrives after the new input data is ready.
**Feedback Output Module:**

The feedback output module is used to display the values calculated by the HC11 game logic for the number of correct numbers in the correct position and the number of correct numbers in the incorrect position. These are displayed on six rows of four red LEDs and four green LEDs. Since these 48 LEDs are far more numerous than the number of available output pins on the FPGA it is necessary to multiplex the output from one set of eight output pins in order to simultaneously drive all 48 LEDs. This requires us to both provide an output for the individual values of each LED and an output sequentially enabling one row of LEDs. We achieve this by having the anode of all eight LEDs of a row connected to the collector of a PNP transistor which has its emitter tied to 5V. The FPGA then drives the base low or high in order to enable or disable a single row. Each LED’s cathode is then tied to the FPGA, if the FPGA pin is low it sinks the LED and turns the LED on, if the pin is held high it prevents current from passing through the LED and turns it off. Therefore the FPGA can individually turn LEDs on or off by setting each particular output pin low or high.

The output module stores up to six input values in a 8-bit wide, 6 bit shift register. The register is triggered to write a new output value when it receives an “update” signal from the HC11. This signal is used to clock the register. The output of each register is sent to an 8-bit wide for input mux that selects the correct value among the six options for the individual LED output pins of a given row. The select signal for the mux is generated by a 3 bit counter which counts from zero to five. The counter signal is also sent to a 6-bit priority encoder that sets all of the row enable outputs high except one. This signal, in conjunction with the output of the mux
multiplexes and drives the six rows of eight LEDs. However, in order to prevent any smearing of the values between rows we slow the clock input of the counter down by sending the system clock through another counter which divides the clock by 2048. This results in a clock rate of about 2 kHz. This should be slow enough to prevent smearing of the values but fast enough to prevent any perceptible flickering.
Results:

We were successfully able to implement the complete mastermind game. It performed all of the functions of the game that we outlined in the original proposal. Furthermore we were able to successfully use both the LCD module and multiplexed LED array for the output and implement interrupt based input exactly as we had originally envisioned.

The most difficult part of the design was determining the correct wiring and initialization procedure for the LCD Module. Learning that the module required negative contrast was quite a revelation. Additionally, the hardware initialization mode of the module did not quite behave as describe in the datasheet and necessitated the workaround of a slow power increase in order to get the LCD functional. Other than that, implementation was relatively straightforward and required only minor debugging.
References:


Parts List:

<table>
<thead>
<tr>
<th>Part</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi LM016H LCD Module</td>
<td>Stock Room</td>
</tr>
</tbody>
</table>
Appendix A: Schematics

Overall System

Motorola 68hc11

Xilinx Spartan FPGA

LED Matrix

Matrix Keypad
Appendix B: FPGA Block Diagrams

Input Module Block Diagram

Output Module Block Diagram
Appendix C: Assembly Code

* AUTHORS: William Berriel and Carl Larsen
* Purpose: The Code for the 68hc11 part of the Mastermind game which
  is the final project for HMC E155
* Date   : 12/9/02
* Email  : wberriel@hmc.edu

* Useful ports and such, as well as masks for the ports when
  necessary.
PORTA  EQU $1000
AMSK   EQU %01000000 ; masks used when pulsing bit 6 of
PORTA
AMSKNOT EQU %10111111
PORTB  EQU $1004
PORTC  EQU $1003 ; PORTC and control
register
DDRC   EQU $1007 ; Set bits high that are
outputs
PORTE  EQU $100A
EMSK   EQU %00000111 ; Mask to get the lower 3 bits of
porte
TCNTL  EQU $100F ; The lower byte of the timer.
GUESS  EQU $0001 ; Stores the number of Guesses that
have been made.
ENDSTAT EQU $0006 ; Whether we are in the ending state.
GLINE  EQU $03 ; Constant storing Guesses per line,
here it's 3
GGAME  EQU $06 ; Constant storing Guesses per Game,
here it's 6
FIRST  EQU $0015 ; Location of the first, second,
third, and fourth
SECOND EQU $0016 ; hidden digits respectively
THIRD  EQU $0017
FOURTH EQU $0018
STATE  EQU $0010 ; STATE is the address of the upper byte
of state (not used)
STATEL EQU $0011 ; STATEL is the address of the lower
byte of the state,
* it determines what digit in a
sequence is being input.
STATE2 EQU $0012 ; STATE2 stores whether we have a
random number yet.

* In1 - In4 store the digits from the guess.
IN1    EQU $0019
IN2   EQU     $0020
IN3   EQU     $0021
IN4   EQU     $0022

IRQVEC EQU $00F1 ; The address of the XIRQ vector in buffalo.

ALOC   EQU     $15  * Temporary Answer Location
GLOC   EQU     $19  * Temporary Guess Location
FBCK   EQU     $03  * Feedback Output Location
FBCKT  EQU     $04  * Temporary CVCP Output Location
FBCKB  EQU     $05  * Temporary CVIP Output Location
MSB1   EQU     %10000000
MSB2   EQU     %00001000

NUM   EQU     $dd00 ; The Address of the Number Strings
to be output.

* LCD code was modified from Ari Moodi and Ryan Stuck, e155 2000.

* OUTPUT Masks
* b5 = RS  Register Select
* b4 = R/W Read/Write
* b3 = E   enable

WRD   EQU     %00100000
WRDEN  EQU    %00101000
WRC   EQU     %00000000
WRCEN  EQU   %00001000

* Commands

CLEAR  EQU     %00000001 ; $01
HOME   EQU     %00000010 ; $02
ENTRY  EQU     %00000110 ; $06
DISPON EQU     %00001111 ; $0c
FUNCT  EQU     %00111000 ; $38
INIT   EQU     %00110000 ; $30
DISPOFF EQU    %00001000 ; $08
DDRLN2 EQU     %11000000 ; $c0

* Time delay to allow for proper interfacing with the LCD
* HTIME is in milliseconds, and are much slower than
* necessary.

HTIME   EQU     $05
DTIME  EQU     $40

ORG     #IRQVEC
JMP     IRQISR

* Start the game by setting up the parameters, the ports, and setting
up interrupts
* for the XIRQ. Then Just busy wait, interrupt driven code.

ORG     $D100
TPA     ; Transfer CCR to A
ANDA  #10111111 ; To unmask the XIRQ, need to reset bit 6
TAP   ; Transfer A to CCR
JSR   RESETGAME
LDAA  #$FF  ; Set PORTC as OUTPUT
STAA  DDRC
JSR   INITLCD  ; Initiate the LCD
CLI   ; Enable Interrupts
BUSYW  BRA  BUSYW

* RESETGAME will initialize the game to a beginning state where it can begin playing the game.

RESETGAME
LDAA  #FIRST
STAA  STATEL  ; STATEL stores which digit is being input

CLRA
STAA  STATE2
STAA  STATE
STAA  GUESS
STAA  ENDSTAT
RTS

* Random Simply takes the lower 8 bits of the time clock, and multiplies by 6.
* The higher byte should be a number between 0 and 6.

RANDOM
LDAA  TCNTL
LDAB  #06
MUL
LDX  STATE
STAA  0,X
RTS

IRQISR
* ON Input, need to see if we're started (have a seed yet) if not, we seed, as long as not a reset, then we handle it.

LDAA  PORTE
ANDA  #EMSK  ; Clean up input, make sure only lower 3 bits are checked.
CMPA  #$05  ; are we not at reset?
BLE    NORESET  ; On reset, simply reset state fully and return.
JMP    RESET  ; would use BGT, but reset is too far away for 8 bit break

NORESET
LDAB  ENDSTAT
CMPB  #$0  ; Are we in an endstate? If so, only accept reset.
BNE    RETURN
LDX  STATE
STAA 4,X ; Store number as guessed input.

LDAA STATE2
CMPA #$0 ; state2 = 0 means we need random numbers
BNE HAVESEQ ; otherwise we don't
JSR RANDOM

HAVESEQ
LDAB PORTE ; To get the character to print, need to get input number

ANDB #EMSK
LDX #NUM ; and add it to the starting point for where the numbers
ABX  ; are stored.
LDAB 0,x ; That should give us the ascii value for the number.
JSR WRI TED ; (Write the character to the screen).

INC STATEL ; move to the next input state

LDAA STATEL
CMPA #IN1    ; If we're not at the 4th input number, return
BLT RETURN  ; otherwise handle it.

STARTED
LDAB SPACE ; Upon receivin g 4 input digits, write a space to the screen.

JSR WRI TED
LDAB #HTIME
JSR IDELAY

JSR SCORE ; Score the inputs.
LDAA FDBCK ; Load the feedback and print it to the FPGA

STAA PORTC ; Output the DATA output first
LDAA PORTA ; Then output the enable, being sure to preserve the state of A
ORAA #AMSK ; Since we only care about bit 6, whereas the LCD runs off of bits
STAA PORT A ; 5,4, and 3.
LDAA PORTA
ANDA #AMSKNOT
STAA PORTA ; Raise the enable for a short time, then lower it.

LDAA #FIRST ; Point the State back at the first digit for the input guess
STAA STATEL
LDAA #1
STAA STATE2

* Now that we have output everything see if we need to go to a win state.

LDAA FDB CK
CMPA #$F0 ; Feeback of #$F0 means we have 4 right in the right place
BNE NOWIN
JSR WIN
BRA RETURN
NOWIN
   INC GUESS ; If not win, increment the number of
   guesses
   LDAA GUESS
   CMPA #GLINE ; See if we need a carriage return
   BNE SAMELINE
   LDAB #HTIME
   JSR IDELAY
   LDAB #DDRLN2
   JSR WRITEC
   LDAB #HTIME
   JSR IDELAY

SAMELINE
   CMPA #GGAME ; See if we are in a lose state
   BNE RETURN
   JSR LOSE

RETURN
   RTI

* Upon Reset, clear the LCD, reset the game and return.

RESET
   LDAB #CLEAR
   JSR WRITEC
   LDAA #HTIME
   JSR IDELAY

   LDAB #HOME
   JSR WRITEC
   LDAA #HTIME
   JSR IDELAY

   JSR RESETGAME
   BRA RETURN

* Upon winning, enter winning state, clear the lcd and output
  winmessage.

WIN
   LDAA #$01
   STA ENDSTAT
   JSR CLEARHOME
   LDX WINMESS

WINLOOP
   LDAB 0,X
   JSR WRITED
   LDAA #HTIME
   JSR IDELAY

   INX
   CMPX #LOSMESS
   BNE WINLOOP

   JSR HIDDENPRINT

RTS
* Upon a loss, Enter the ending state, clear the LCD and output the losing message

LOSE

    LDA #$01
    STA ENDSTAT
    JSR CLEARHOME
    LDX #LOSMESS

LOSELOOP

    LDA 0,X
    JSR WRITED
    LDA #HTIME
    JSR IDELAY
    INC
    CMPX #ENDPT
    BNE LOSELOOP
    JSR HIDDENPRINT
    RTS

* Print the hidden sequence.

HIDDENPRINT

    LDA #DDRLN2
    JSR WRITEC
    LDA #HTIME
    JSR IDELAY

PRINTLOOP

    LDA FIRST
    LDX #NUM
    ABX
    LDA 0,X
    JSR WRITED
    LDA #HTIME
    JSR IDELAY
    LDA SECOND
    LDX #NUM
    ABX
    LDA 0,X
    JSR WRITED
    LDA #HTIME
    JSR IDELAY
    LDA THIRD
    LDX #NUM
    ABX
    LDA 0,X
    JSR WRITED
    LDA #HTIME
    JSR IDELAY
    LDA FOURTH
    LDX #NUM
    ABX
    LDA 0,X
    JSR WRITED
    LDA #HTIME
* Send the clear and home commands.

```
CLEARHOME
LDAB #CLEAR
JSR WRITEC
LDAA #HTIME
JSR IDELAY

LDAB #HOME
JSR WRITEC
LDAA #HTIME
JSR IDELAY
RTS
```

* Check Correct Value Correct Position

```
SCORE
LDAB #$00
LDX #ALOC
LDY #FDBCKT

CVCP
LDAA 0,X * Check first two numbers
CMPA 4,X
BNE NOMATCH
LSLB * If a match Shift left
INCB * and increment

NOMATCH
INX * Move to next number
CPX #GLOC
BNE CVCP * If not 4th no. loop
STAB FDBCKT * Store result
BEQ DNCVCP * If zero result don't shift

SHIFT1
BRSET 0,Y MSB1 DNCVCP
LSL FDBCKT * Loop till output is shifted
BRA SHIFT1 * completely to the MS Bits
```

* Check Correct Value Incorrect Position

```
DNCVCP
LDAB #$00
LDX #ALOC
LDY #GLOC

CVIP1
LDAA 0,X * Check answer no.
CMPA 0,Y * vs first guess no.
BNE CVIP2
LSLB * If match shift and increment
INCB
LDAA #$FF * Mark guess no. as used
STAA 0,Y
BRA DNCHK

CVIP2
CMPA 1,Y * Check answer no.
BNE CVIP3 * vs second guess no.
LSLB
INCB * If match shift and increment
LDAA #$FF
STAA 1,Y * Mark guess no. as used
BRA DNCHK

CVIP3
CMPA 2,Y * Check answer no.
```
BNE CVIP4 * vs third guess no.
LSLB
INCB
LDAA #$FF
STAA 2,Y * Mark guess no. as used
BRA DNCHK
CVIP4
CMPA 3,Y * Check answer no.
BNE DNCHK * vs fourth guess no.
LSLB
INCB * If match shift and increment
LDAA #$FF
STAA 3,Y * Mark guess no. as used
DNCHK
INX
CPX #GLOC * If not all answer numbers checked
BNE CVIP1 * loop back and compare vs. guess again
STAB FDBCKB * Store result
BEQ DNCVIP * If zero result skip shift
LDY #FDBCKB
SHIFT2
BRSET 0,Y MSB1 DNCVIP
LSL FDBCKB * Loop till output is shifted
BRA SHIFT2 * completely to the MS Bits

* Calculate Final Output

DNCVIP
LDAA FDBCKB * Subtract CVCP value from CVIP value
SUBA FDBCKT
LSRA * Shift new CVIP to Lower Nibble
LSRA
LSRA
LSRA
STAA FDBCKB
BEQ DNSH3
LDY #FDBCKB
SHIFT3
BRSET 0,Y MSB2 DNSH3
LSL FDBCKB * Shift CVIP to MS Bits
BRA SHIFT3 * of lower nibble
DNSH3
LDAA FDBCKT
ORAA FDBCKB * Or CVCP and CVIP to get final output
STAA FDBCK* Store final Output
RTS

* Write Data that's in accumulator b

WRITED
LDAA #WRD
STAA PORTA
JSR STALL
LDAA #WRDEN
STAA PORTA
JSR STALL
STAB PORTB
LDAA #WRD
STAA PORTA
RTS

* Write Command in accumulator b
WRITEC
LDAA #WRC
STAA PORTA
JSR STALL
LDAA #WRCEN
STAA PORTA
JSR STALL
STAB PORTB
LDAA #WRC
STAA PORTA
RTS

* Stall Function
STALL
LDY #$0100
LOOP
DEY
CPY #$0000
BNE LOOP
RTS

* DELAY Function, to delay for 1 ms
DELAY
LDY #$01E8 ; 1000 loops
MORE
DEY ; 4
NOP ; 2
NOP ; 2
NOP ; 2
NOP ; 2
CPY #$0000 ; 5
BNE MORE ; 3
RTS

* Instruction Delay, delays for number of seconds in A
IDELAY
DECA
JSR DELAY
CMPA #$00
BNE IDELAY
RTS

* Initialize the LCD, hardware initialize the LCD to 2 lines,
blinking cursor,
* 8 bit input, and the cursor beginning in the home position.
INITLCD
LDAB #INIT
JSR WRITEC
LDAA #HTIME
JSR IDELAY
LDAB #INIT
JSR WRITEC
LDAA #HTIME
JSR IDELAY
LDAB #INIT
JSR WRITEC
LDAA #HTIME
JSR IDELAY
LDAB #FUNCT
JSR WRITEC
LDAA #HTIME
JSR IDELAY

LDAB #DISPOFF
JSR WRITEC
LDAA #HTIME
JSR IDELAY

LDAB #CLEAR
JSR WRITEC
LDAA #HTIME
JSR IDELAY

LDAB #ENTRY
JSR WRITEC
LDAA #HTIME
JSR IDELAY

LDAB #DISPON
JSR WRITEC
LDAA #HTIME
JSR IDELAY

RTS

ORG NUM
FCC "0"
FCC "1"
FCC "2"
FCC "3"
FCC "4"
FCC "5"
SPACE FCC " "
WINMESS FCC "YOU WIN"
LOSMESS FCC "YOU LOSE"
ENDPT FCC "E"
Appendix D: Verilog

/*
Name: mastmind
Author: Carl V. Larsen
Date: 10 - 24 - 02

This module is the top level of the FPGA portion of the mastermind game. It combines the keypad input and multiplexed feedback output portions into one module and provides the correct reset behaviour for the output module.
*/

module mastmind(clk,reset,update,hc11val,row,leds,select,column,s0,intr);

input clk;
input reset;
input update;
input [7:0] hc11val;
input [2:0] row;
output [7:0] leds;
output [5:0] select;
output [2:0] column;
output [2:0] s0;
output intr;

wire outreset;
assign outreset = s0[2]&s0[1];

mminput inpart(clk,reset,row,column,s0,intr);
moutput outpart(clk,outreset,update,hc11val,leds,select);
endmodule

User Input Module:

/*
Name: mminput
Author: Carl V. Larsen
Date: 10 - 24 - 02

This module is the top level module for the user input of the mastermind game. It decodes matrix keypad input into binary and generates an interrupt signal each time a button is pressed.
*/

module mminput(clk,reset,row,column,s0,intr);

input clk;
input reset;
input [2:0] row;
output [2:0] column;
output [2:0] s0;
output intr;
wire u;
wire update;
wire upnot;
wire [2:0] num1;
wire [2:0] s0;
wire sclk;

assign upnot = ~update;

// slow down internal clock
div2k slwclk(clk,reset,sclk);

// scan for input
scanner scanfsm(row,sclk,reset,column);

// generate hC11 input interrupt
wrtenb enabler(sclk,reset,row,update);

// decode matrix input to binary
number numdecd(row,column,num1);

// store most recent input
flipflop reg0(sclk,reset,update,num1,s0);

// store the input interrupt
flopr intrreg(sclk,reset,upnot,intr);

endmodule

/*
Name: flipflop
Author: Carl V. Larsen
Date: 10 - 07 - 02

This module is a simple 3-bit asynchronously resettable
flip-flop with enable. It is used to store the column output for the
keypad.
*/

module flipflop(clk,reset,en,d,q);
  input clk;
  input reset;
    input en;
  input [2:0] d;
  output [2:0] q;

    reg [2:0] q;

    always @(posedge clk or posedge reset)
      if (reset) q <= 3'b0;
      else if (en) q <= d;

endmodule
module flopr(clk, reset, d, q);
  input clk;
  input reset;
  input d;
  output q;

  reg q;

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

module number(row, column, num);
  input [2:0] row;
  input [2:0] column;
  output [2:0] num;


endmodule
module scanner(row, clk, reset, state);
  input [2:0] row;
  input clk;
  input reset;
  output [2:0] state;

  reg [2:0] state, nextstate;

  parameter NP = 3'b111;
  parameter S0 = 3'b110;
  parameter S1 = 3'b101;
  parameter S2 = 3'b011;

  always @(posedge clk or posedge reset)
    if (reset) state <= S0;
    else state <= nextstate;

  always @(state or row)
    case (state)
      S0: begin
        if (row == NP)
          nextstate <= S1;
        else nextstate <= state;
      end

      S1: begin
        if (row == NP)
          nextstate <= S2;
        else nextstate <= state;
      end

      S2: begin
        if (row == NP)
          nextstate <= S0;
        else nextstate <= state;
      end

      default: nextstate <= S0;
    endcase
endmodule
module wrtenb(clk,reset,row,update);
    input clk;
    input reset;
    input [2:0] row;
    output update;

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

    reg [1:0] state, nextstate;

    always @(posedge clk or posedge reset)
        if (reset) state <= S0;
        else state <= nextstate;

    always @(state or row)
        case (state)
            S0: if (~&row) nextstate <= S1;
                else nextstate <= state;
            S1: nextstate <= S2;
            S2: if (&row) nextstate <= S0;
                else nextstate <= state;
            default: nextstate <= S0;
        endcase

    assign update = state[0];
endmodule
Feedback Output Module:

/*
Name: mm output
Author: Carl V. Larsen
Date: 10 - 24 - 02

This module is the top level module for the mastermind feedback output. It uses an 8-bit shift register to store the feedback information retrieved from the hc11 and then multiplexes these 6 six values to display on 48 LEDs.
*/

module mmoutput(clk,reset,update,hc11val,leds,select);
  input clk;
  input reset;
  input update;
  input [7:0] hc11val;
  output [7:0] leds;
  output [5:0] select;

  wire [7:0] q0, q1, q2, q3, q4, q5;
  wire [3:0] selmux;
  wire [7:0] invleds;

  // shift register stores feedback values. Write is enabled
  // by an output signal from the hc11.
  flopr8 flop0(update,reset,hc11val,q0);
  flopr8 flop1(update,reset,q0,q1);
  flopr8 flop2(update,reset,q1,q2);
  flopr8 flop3(update,reset,q2,q3);
  flopr8 flop4(update,reset,q3,q4);
  flopr8 flop5(update,reset,q4,q5);

  // slow down the clk to prevent smearing of LED output
  div2k slowclk(clk,reset,sclk);

  // generate signals to cycle through each of the six outputs
  switcher switgen(sclk,reset,selmux,select);

  // multiplex the outputs
  mux6_8 bigmux(q0,q1,q2,q3,q4,q5,selmux,invleds);

  assign leds = ~invleds;
endmodule
module div2k(clk, reset, y);
    input clk;
    input reset;
    output y;

    parameter S0 = 11'b000_0000_0000;
    parameter SF = 11'b111_1111_1111;

    reg [10:0] state, nextstate;

    always @(posedge clk or posedge reset)
        if (reset) state <= S0;
        else state <= nextstate;

    always @(state)
        if (state == SF) nextstate <= S0;
        else nextstate <= state + 1;

    assign y = state[10];
endmodule

module flopr8(clk, reset, d, q);
    input clk;
    input reset;
    input [7:0] d;
    output [7:0] q;

    reg [7:0] q;

    always @(posedge clk or posedge reset)
        if (reset) q <= 8'b0;
        else q <= d;
endmodule
module mux6_8 (d0, d1, d2, d3, d4, d5, s, y);
    input [7:0] d0;
    input [7:0] d1;
    input [7:0] d2;
    input [7:0] d3;
    input [7:0] d4;
    input [7:0] d5;
    input [2:0] s;
    output [7:0] y;
    wire [7:0] A, B, C, AA;
    mux2_8 Amux(d0, d1, s[0], A);
    mux2_8 Bmux(d2, d3, s[0], B);
    mux2_8 Cmux(d4, d5, s[0], C);
    mux2_8 AAmux(A, B, s[1], AA);
    mux2_8 finalmux(AA, C, s[2], y);
endmodule

module mux2_8 (d0, d1, s, y);
    input [7:0] d0;
    input [7:0] d1;
    input s;
    output [7:0] y;
    assign y = s ? d1 : d0;
endmodule
This module has a 3 bit counter which counts from 0 to 5. It uses these values to switch between values on the mux and calls the priority encoder to generate the select output for each row of leds.

```verilog
module switcher(clk, reset, selmux, seldisp);
    input clk;
    input reset;
    output [2:0] selmux;
    output [5:0] seldisp;
    wire [5:0] invsel;
    reg [2:0] q;

    always @(posedge clk or posedge reset)
        if (reset) q <= 2'b0;
        else q <= q[2]&q[0] ? 0 : q + 1;

    assign selmux = q;
    d2x6 priority(selmux, invsel);
    assign seldisp = ~invsel;
endmodule
```
module d2x6(select, out);
  input [2:0] select;
  output [5:0] out;

  reg [5:0] out;

  always @(select)
  begin
    out = 0;
    case (select)
      0: out[0] = 1;
      1: out[1] = 1;
      2: out[2] = 1;
      3: out[3] = 1;
      4: out[4] = 1;
      5: out[5] = 1;
      default: out[0] = 1;
    endcase
  end
endmodule