PONG GAME

FINAL PROJECT REPORT

DECEMBER 12, 2002

E155

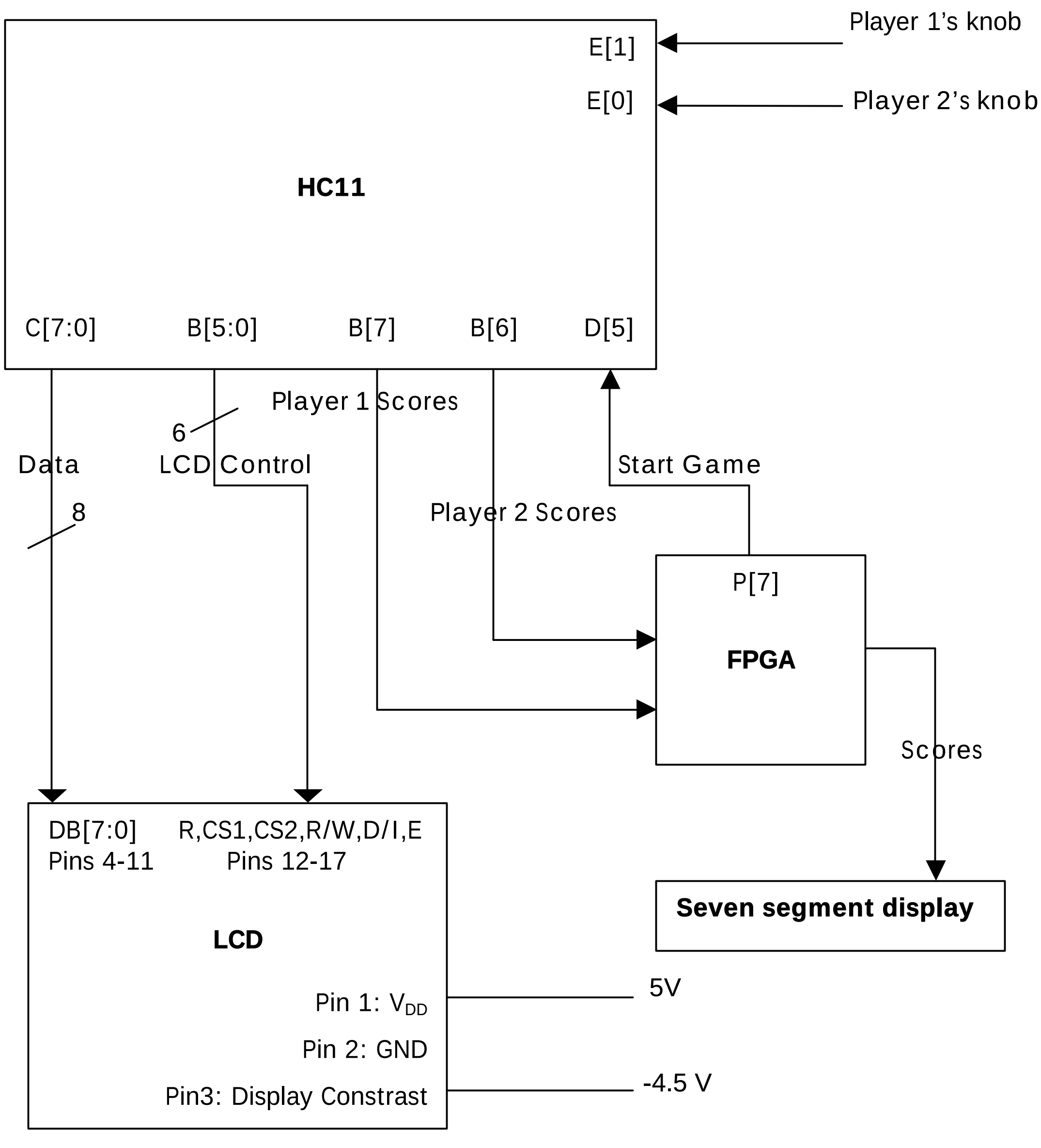
Reneé Logan & Philip Vegdahl

Abstract:

Among the earliest computer games ever to hit the American household was pong. It is a fairly simple game based on hitting a ball back and forth between two players. The goal of each player is to hit the ball in such a way that it will move past the other player's paddle without them being able to deflect it back. When this happens, that player scores a point. For our remake of this classic we used an LCD to display the game, an analog knob to control each paddle, and a dual seven segment display to show the scores. Our final product performs entirely to the specifications we set out, except for the start up sequence which may be related to a bug in the microcontroller.

Introduction

This project is an adaptation of the classic video game, Pong. It is a contest between two players to see who can use their paddle to knock a bouncing ball past the other player's paddle. Whenever a player achieves this, they score a point. After a score, the ball reappears moving straight down the centre of the screen, away from the side where it just scored. The first player to reach seven points wins.



When a player has won the game stops until the FPGA is reset. The reset button on the FPGA will zero the scores and start the game over. If the reset button is pressed at any other time then the score will simply be reset. The score is displayed on a dual seven segment display. The game itself appears on a LCD with a resolution of 128x64 pixels. Each player controls their paddle with a separate analogue knob. These knobs will provide a DC voltage between 0 and 5 volts depending on the angular position of the knob. These values are then interpreted by the HC11 A/D converter to create paddle positions on the short sides of the display.

All of the controlling logic for the game is done in the FPGA and the HC11. The HC11 calculates the physics of the game and controls the LCD. The FPGA keeps track of the score and outputs it to the dual seven segment display.

Starting the Game

To run this game, first power up the hc11 and load the file main.s19 in. The FPGA can now be powered up. Type 'g d000' at the command prompt which should start the game with some flaws (see results section). Reset the HC11 and reload the file. Type 'g d000' again. This time the game should start correctly. It is unknown why this is necessary, or even helps.

Microcontroller Design

The HC11 provided the control for most of the game. The entire body of code can be found in the appendix. All of the code was written using its standard assembly language. The code can be divided up into 6 basic sections that will be explained in more detail later. There is the run once, initialization code. This code turns on the LCD and sets the HC11 to a few initial states. There is the idling control code. This code is usually doing nothing except polling for specific port values. The remaining four sections are all called in the following order once ever eight milliseconds as part of the real time interrupt. One section uses the A/D converter to get paddle positions. There are two sections that control ball movement, one for each direction of motion. The last section updates the LCD based on the ball and paddle positions stored in memory.

Start up Sequence

During the start up sequence the A/D converter and LCD are turned on and port D is configured for output on its lowest two bits. These are the bits that are used for player score signals. They are both stored with an initial value of 1 to be consistent with the initial state of the FPGA finite state machine. Interrupts in general are also enabled at this time, although no specific interrupts are yet active.

Idling Control

This section of code will simply run an unproductive loop while the game is in progress. While the game is running all of the game's data will be handled by the real time interrupt. Whenever a score signal goes high, this code will disable the real time interrupt until the FPGA sends its start signal. At that point the code will set all of the values for the game to start again, the interrupt will be re-enabled, and then it will return to the loop.

A/D Converter Code

This code configures and runs the A/D converter to grab the values from pins E0 and E1. It shifts down the top 6 bits to create a number in the range [0:63]. It then edits them into the range [5:58] so that that the paddle will always remain entirely on the screen. To edit the numbers into this range it simply takes the bits outside of this range and moves them to the closest value inside the range.

Horizontal Method of Ball Movement

This method controls the horizontal movement of the ball by changing its position periodically based on its speed. It also checks to see when the ball collides with a wall and changes the direction of the ball's velocity, while keeping the same magnitude.

Vertical Method of Ball Movement

This controls the vertical movement of the ball. It periodically changes the position of the ball based on its velocity. Whenever the ball reaches one edge of the screen it checks to see if the paddle position overlaps with the position of the ball. If it does not overlap, then it will send a signal to the FPGA indicating that a player has scored. If they do overlap, then it will reverse the direction of the ball's vertical velocity while maintaining the same magnitude. The position of the ball on the paddle determines how much the horizontal velocity of the ball changes. A dead centre hit will produce no change, while a hit at the very edge of the paddle will create a large change in the direction of the ball relative to the centre of the paddle. The velocity of the ball is limited to a magnitude of 64 to prevent a strange physics engine as the result of overflow errors, and to accommodate a limitation of the algorithm that controls the movement of the ball.

Movement Algorithm

The same algorithm is used to control the movement of the ball for each direction. The velocity of the ball is a number in the range [-64:64]. During each cycle of a not yet determined length, the magnitude of the ball velocity is added to an accumulator. Whenever this accumulator reaches a number equal or greater than 64, the ball is moved over one pixel in the appropriate direction and the 64 is subtracted from the accumulator, which is equivalent to doing a mod 64 on its value.

Display On

In this routine, the LCD's display is turned on. When the LCD is first powered data is sent to the LCD display RAM but nothing is actually displayed on the screen. To turn on the LCD display, the reset signal is set high and all other instructions (CS1, CS2, R/W, D/I, E) are set low. Then $3F is sent to the data bus where the last bit indicates whether or not the display is on or off (1 being on, 0 being off).

LCD Write

In this routine Port C is firstly set up to take input. The control pattern is then masked so that only CS1, CS2, and Reset retain the value. Then the R/W is set high which now makes this pattern the status check pattern. This pattern is written to port B and then after a micro second (timing issues) the enable bit is set high which executes the pattern on the LCD. At this point the data on the data bus is read off port c. If the data bus is all zeroes, this indicates that the LCD isn't busy, the display is on and reset is low, all of which are necessary. If the data bus is not all zeroes, the function keeps looping until the data bus is all zeroes. Port C is now set up to output. The control pattern is put on port b and then the enable bit is set high after half a micro second. The data or instruction is also written to port C at this time and then the enable signal is dropped to indicate that the operation is finished. The idle state is then turned on.

Writing to the LCD / Clearing a Pixel

This function calls the LCD write routine to write a pixel to the screen. Firstly the control pattern is loaded so that the LCD knows whether to write to CS1 or CS2. This control pattern also has Reset high, D/I low, R/W low and enable low. The page number is loaded into the data field and the LCD Write function is called with these values of LCD control and Data. After this the Y coordinate of the ball is taken and then the 6th bit forced high to convert it to a set line instruction and stored into data. LCD write is once again called with LCD control (which hasn't changed from the last time LCD write was called) and the present value of data. Lastly, the 1st bit of the LCD control pattern is set high to indicate that the next value coming over the Data bus is the data to be written to the LCD. Then the pattern of 0s and 1s for the page earlier specified is loaded into data and so written to the LCD. A slight variation of this method is the clear pixel method. Instead of calculating the data to be written to the LCD, the data is automatically set to #$00 which clears each pixel in the current page.

Clear LCD / Clear Line

The Clear LCD function writes all zeroes to the LCD, effectively clearing the LCD. Basically the routine starts at line 127 and then for each page writes zeroes to that page. After this is finished the routine moves on to the next line and repeats the process. The Clear line subroutine is just a subset of the Clear LCD routine that takes in a line and writes zeroes to each page on that line.

Paddle

This function writes the 11 pixel wide paddle to the screen given the location of the centre of the paddle. The reason the ordinary write function can't be used to in a loop 11 times to write the 11 pixels to the screen is that one page of data has to be addressed at a time, and so if this was used within one page only one pixel would be written to. I decided that the paddles shouldn't be able to come partially off the screen so the routine first checks if that would normally happen and then sets the value of the centre of the paddle so that the paddle will stop at the edge of the screen. Even though this is also done in the A/D converter, I found for a bug free game this had to be repeated in the LCD engine as well. After this the x coordinate of the paddle is decremented by five to get the first pixel in the paddle. The page to be written to is found using the setpage subroutine as it was in the writing to the LCD subroutine. However, obtaining the data is a lot different. The first time in the loop the number of darkened pixels in that page was the last 3 bits of the x coordinate of the first pixel of the paddle which would indicate the page data. After this the number of darkened pixels for each page was found by loading in the number of pixels that were left to be written to the LCD. The page was initialized to being all darkened pixels and then shifted right (8 – num of darkened pixels) times. If this is the first part of the paddle this pattern needs to be flipped so that the first part of the paddle can connect with the rest of the paddle. As a result the number of darkened pixels needs to be flipped as well. This is done by subtracting the data and dark values from #$FF and #$08 respectively. In the case where the total number of pixels left to be written to the LCD is more than 8 this value if forced to 8.

Idle Pattern

Thanks to Aaron Stratton (who also used this graphical LCD last year), I know that an idle pattern is a very good idea between instructions so that random garbage doesn't get written to / read from the LCD. After every instruction, an idle pattern of $38 is loaded onto port B which means that reset, CS1 and CS2 are all set high and R/W, D/I and E are all low to make sure that nothing can get read from or written to the LCD at an inappropriate time and also as an added timing precaution.

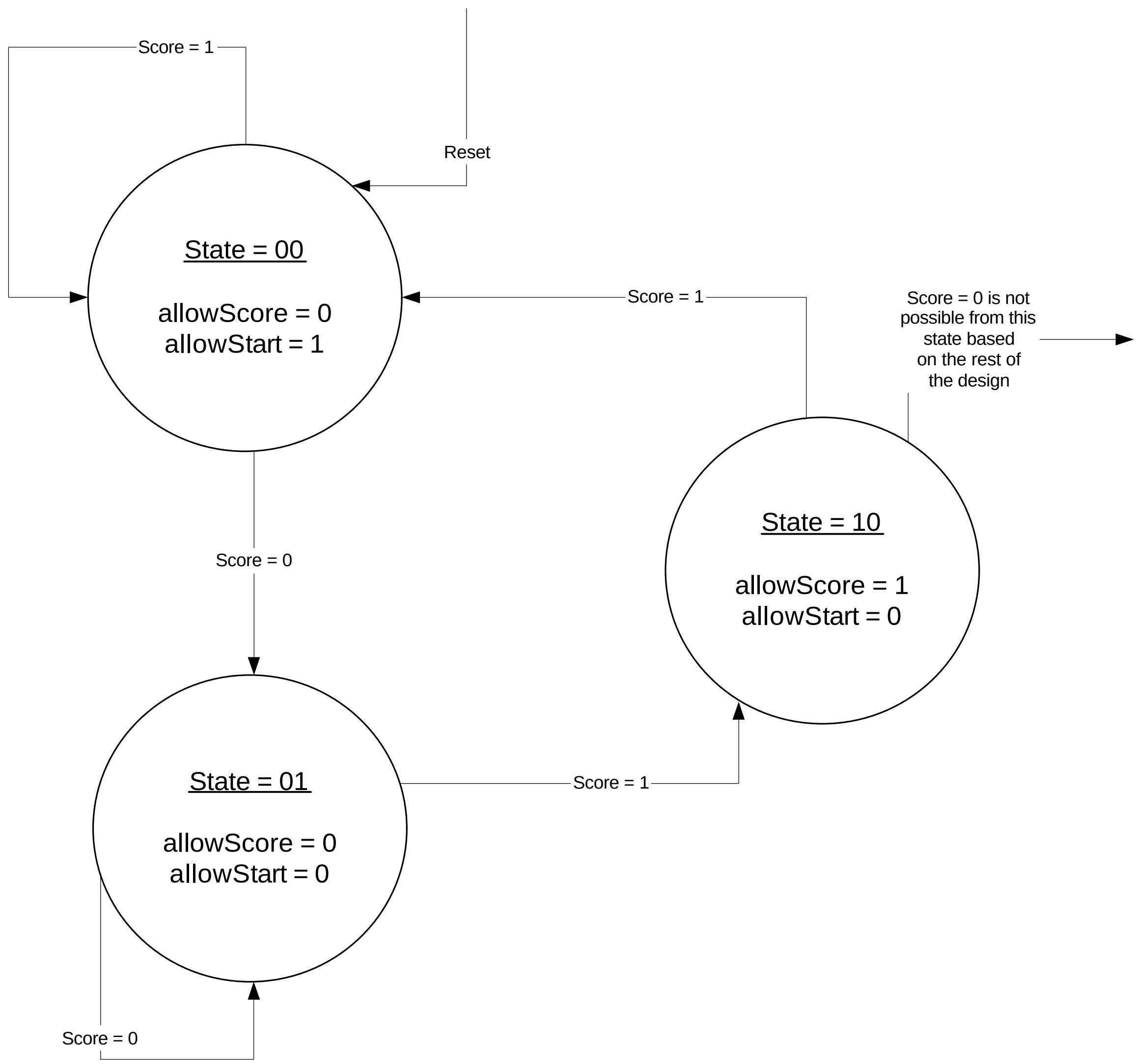
FPGA Design

HC11 & FPGA Handshake

The score controlling interface between the HC11 and the FPGA is done as a handshake. When the HC11 physics engine detects a score, it outputs one of two signals to the FPGA indicating which player scored. The HC11 then pauses the game by disabling the interrupt used to time the game. When the FPGA receives a score signal it updates the score and then sends its handshake signal back to the HC11 telling it to start again. This signal is not sent when the score of one player reaches 7 points (i.e. when a player has won). The HC11, upon receiving this start signal, lowers all scoring signals and re-enables the real time interrupt so that play will resume. At this point the FPGA lowers its start signal and starts waiting for another score to occur.

FPGA

The dual seven segment display driver on the FPGA was taken from lab 3, and was edited to remove the LED outputs. The score controlling logic involves the before mentioned handshake with the HC11. The FPGA uses the following three state finite state machine to keep track of which part of the handshake it is in.



The input signal "score" is an OR of both players' score signals, and is therefore true when either player scores. The output signal "allowScore" tells other logic that it can add the current scoring signals to the old scores, thus updating the score. The other output, "allowStart," sends a start signal back to the FPGA unless the score of one player is seven.

In the next higher module above this finite state machine the outputs of the FSM are refined a bit more and executed upon. The allowStart signal is anded with the player scores both being non-seven and is then outputted to the HC11 as the start signal. Each cycle the current score is updated by adding to it the players scoring signals anded with allowScore. Since allowScore is only held high for a single cycle, this insures that the score is not double added. Even though the scores only require three bits to represent numbers zero to seven, four bits are used for the convenience of using the already designed four bit seven segment display driver. At this level the two player scoring signals are also ored to create the single scoring signal for the FSM.

At the top level of the FPGA circuit, the scores for the two players are sent to be displayed on the seven segment displays. The proper inputs and outputs to the FPGA are also set as such. All of the Verilog code can be found in the appendix.

Results

Our project resulted in a pong game that functions nearly entirely in the spec that we set out. The only problem that we have noticed is that the first time we try to start up the system and run the game the SB6108 chip that drove the part of the LCD furthest from pin 1 would display pixels on the LCD one pixel more to the right than it should. This problem was always solved by resetting the HC11 and reloading the file.We are not sure if this is a strange bug in the HC11s that we have been using, or a bug in the LCD hardware. In either case, once the game is started and the file reloaded it runs flawlessly.

The biggest challenges in this project involved actually getting the LCD display to perform correctly. Everything from turning the LCD on, to drawing a single pixel, to drawing an actual pong screen as a group of pixels proved to be a challenge. This project also involved interfacing together far more pieces of complex hardware than had ever been done in class.

The back and forth communication involved in the handshake was not something we had previously done in class. This aspect of the project, however, actually worked correctly without any real problems to debug. This is likely the result of careful planning prior to the actual coding of the hardware.

If given more time to expand upon and improve this project a few changes we would look into would be fixing the start up conditions so that it starts up correctly on the first try every time. We would look into using the backlighting built into the LCD to improve the visibility, and therefore playability, of the game. There is also a slight bug in the physics engine that potentially allows the ball to move one pixel into a wall and then bouncing back out under very specific conditions. Since this is only a rare bug, and likely wouldn't be noticeable when it did happen, we decided not to fix it for our final project.

References

1. CrystalFontz Graphical LCD Products,
http://www.crystalfontz.com/products/12864b/CFAG12864BWGHV.pdf
http://www.crystalfontz.com/products/DS_S6B0108_V00.pdf

Parts List

Part	Source	Vendor Part #	Price
LCD Display	Crystalfontz.com	CFAG12864B-WGH-V	37.03
Knobs	Stockroom		

Appendix

HC11 Code

```
* main.asm
* Written by Philip Vegdahl - pvegdahl@hmc.edu
*            Reneé Logan - rlogan@hmc.edu
*
* Started on November 23, 2002
* Finished on December 9, 2002
*
*This file codes the movement of the paddles and ball on the screen and
outputs this information to the LCD.


**********************************
*Constants
**********************************
PORTB EQU   $1004       *Port B Register

PORTC EQU   $1003       *Port C Register
DDRC  EQU   $1007       *Port C Data Direction Register
PORTD EQU   $1008
DDRD  EQU   $1009
TMSK2 EQU   $1024
TFLG2 EQU   $1025
PACTL EQU   $1026
SCCR2 EQU   $102D
ADCTL EQU   $1030
ADR1  EQU   $1031
ADR2  EQU   $1032
OPTION      EQU   $1039

**********************************
*Variables
**********************************
      ORG   $0000
LCDCTRL     EQU   $0001       *Control Pattern for LCD
DATA  EQU   $0002       *Data to be written to LCD
PAGENUM     EQU   $0004 *Max value of page number
LINENUM     EQU   $0005       *Max value of line number
CURLINE     EQU   $0006       *Current Line
CURPAGE     EQU   $0007       *Current Page
KNOB1 EQU   $DFF0       *Current knob x - coordinate values
KNOB2 EQU   $DFF1
HVEL  EQU   $DFF2       *Horizontal velocity of ball
VVEL  EQU   $DFF3       *Vertical velocity of ball
XCOOR EQU   $0008       *Horizontal position of ball
YCOOR EQU   $0009       *Vertical position of ball
OLDX  EQU   $000A       *Old horizontal position of ball
OLDY  EQU   $000B       *Old vertical position of ball
KNOBX EQU   $000C       *Horizontal position of centre of paddle
KNOBY EQU   $000D       *Vertical position of paddle
```

```
DARK  EQU   $000E       *Number of pixels of the paddle that have been
written to the screen
TOTAL EQU   $000F       *Total number of paddle pixels to be written
TIMES EQU   $0010       *Number of consecutive times the pwrite loop
(in paddle) has been executed
COUNT EQU   $0011       *Number of consecutive times the clear loop (in
clrline) has been executed
HCOUNT      EQU   $DFF6       * counters for moving ball
VCOUNT      EQU   $DFF7

START EQU   $D000       * Program ORG location
RTI   EQU   $D100       * RT Interrupt ORG location



      ORG   $00EB       * RTI Jump vector location

      JMP   rStart



      ORG   START

* This first section of the code is only run once to initialize the
* HC11 and the LCD display to the correct initial states.

Start BSET  OPTION #$80 * Turn on AD Converter
      LDAA  #$40
Delay DECA              * Count down on A to let A/D warm up
      BNE   Delay
      STAA  YCOOR       * Initialize YCOOR to 0
      STAA  SCCR2       * Needs to be cleared for PortD to work

*LCD initialization

      LDAA  #$38        *Idle Control Pattern
      STAA  PORTB       *Write it to the LCD control lines

*End LCD initialization


*Turn On LCD

      LDAA  #$20        *Generate LCD Write Control Pattern
      LDAB  #$3F        *Generate Display On instruction

      STAA  PORTB       *Write control to Port B
      NOP               *Timing Issues
      NOP
      ORAA  #$01        *Set E bit in the Control Pattern High
      STAA  PORTB       *so that things can be written to the LCD
      STAB  PORTC       *Write the Instruction to Port C

      LDAA  #$38        *Control Pattern Idle State
      STAA  PORTB       *Write it to the LCD
      LDAB  #$00        *Put empty data on to the LCD control lines
      STAB  PORTC
```

```
*End LCD on Routine

      LDAA  PACTL        * Choose speed for RTI to be ~8ms
      ORAA  #$01
      STAA  PACTL
      LDAA  #$03
      STAA  DDRD         * Pins 0 & 1 set as outputs
      STAA  PORTD        * Reset state for port D
      JSR   CLRLCD             *Clear LCD
      CLI                * Enable interrupts

* This section of the code essentially idles while the interrupts
* are controlling the game. Whenever a player scores it disables
* interrupts until the FPGA tells it to start again. When that
* signal comes it re-enables interrupts and goes back to idling

mWait LDAA   PORTD
      ANDA  #$03         * Mask for point score bits
      CMPA  #$00
      BEQ   mWait        * Game currently in progress
      LDAA  TMSK2        * set mask bit 6 low to disable RTI
      ANDA  #$BF
      STAA  TMSK2        * Disable real time interrupts
      LDAA  PORTD
      ANDA  #$20         * Mask for ready to start bit
      CMPA  #$20
      BNE   mWait        * Not ready to start again yet
      STAA  PORTD        * Clear score bits
      LDAA  #$20
      STAA  XCOOR        * Initialize XCOOR to 32
      LDAA  YCOOR
      CMPA  #$00
      BNE   mTop         * Ball at top of screen
      LDAB  #$10         * VVEL will be +16
      BRA   mSkip

mTop  LDAB  #$F0         * VVEL will be -16

mSkip STAB  VVEL         * Initialize VVEL
      LDAA  #$00
      STAA  HVEL         * Initialize HVEL to 0
      STAA  VCOUNT       * Reset movement counters
      STAA  HCOUNT
      LDAA  TMSK2        * set mask bit 6 high to enable RTI
      ORAA  #$40
      STAA  TMSK2        * Enable Real time interrupts
      BRA   mWait


      ORG   RTI

rStart      LDAA  #$40         * Clear interrupt flag
      STAA  TFLG2
      JSR   aStart             * A/D subroutine
      JSR   hStart             * Horizontal ball movement subroutine
      JSR   vStart             * Vertical ball movement subrountine
```

```
      JSR   UPDATE
      RTI



* This subroutine grabs the knob values from the A/D ports
* E0 and E1. It then converts these values to 6 bit paddle
* positions representing the center of the paddles.

aStart      LDAA  #$10        * Configure ADCTL to start
      STAA  ADCTL

aSpin LDAB  ADCTL       * Load ADCTL to check for done
      ANDB  #$80
      CMPB  #$80        * Is the Convertion Complete flag set?
      BNE   aSpin       * Not done, keep waiting.
      LDAA  ADR1
      LDAB  ADR2
      LSRA              * Shift down to lowest 6 bits for
      LSRA              * position of the center of the paddle
      LSRB              * (ie a number in the range [0:63])
      LSRB
      CMPA  #$05        * Hold paddle on the screen
      BGE   aSkip1
      LDAA  #$05        * Paddle off on low end, bring up
      BRA   aSkip2
aSkip1      CMPA  #$3A
      BLE   aSkip2
      LDAA  #$3A        * Paddle off high end, bring down
aSkip2      CMPB  #$05
      BGE   aSkip3
      LDAB  #$05        * Paddle off on low end, bring up
      BRA   aSkip4
aSkip3      CMPB  #$3A
      BLE   aSkip4
      LDAB  #$3A        * Paddle off high end, bring down
aSkip4      STAA  KNOB1       * Store paddle positions
      STAB  KNOB2
      RTS



* This subroutine handles all of the horizontal motion of
* the ball. It changes its position based on velocity and
* changes its velocity on wall collisions.

hStart      LDAA  HVEL
      CMPA  #$00        * Check horizontal direction of ball
      BEQ   hDone       * Ball not moving
      BLT   hNeg        * Ball movement negative
      LDAB  HCOUNT
      ADDB  HVEL        * Update count
      CMPB  #$40        * Check count for ready to move ball
      BLT   hWait       * keep waiting
      SUBB  #$40
      STAB  HCOUNT            * Mod the count by 64 and save
      LDAA  XCOOR
```

```
      INCA                 * Move ball right by one
      STAA  XCOOR
      CMPA  #$3F           * right wall
      BGE   wall
      BRA   hDone

hNeg  LDAB  HCOUNT
      SUBB   HVEL          * Update count
      CMPB  #$40           * Check count for ready to move ball
      BLT   hWait          * keep waiting
      SUBB  #$40
      STAB  HCOUNT               * Mod the count by 64 and save
      LDAA  XCOOR
      DECA                 * Move ball left by one
      STAA  XCOOR
      CMPA  #$00           * left wall
      BLE   wall
      BRA   hDone

wall  LDAB  #$00
      SUBB   HVEL          * flip horizontal velocity direction
      STAB   HVEL
      BRA   hDone

hWait STAB  HCOUNT               * save new count and keep waiting

hDone RTS



* The subroutine handles the vertical motion of the ball.
* Whenever the ball reaches the end of the screen, it
* checks to see if there is a paddle collision or a score,
* then either changes the ball velocities, or asserts the
* appropriate player's scoring signal.

vStart      LDAA  VVEL
      CMPA  #$00
      BLT   vNeg           * Ball movement negative
      LDAB  VCOUNT
      ADDB  VVEL           * Update count
      CMPB  #$40           * Check count for ready to move ball
      BGE   jSkip1               * Branch out of range, must use jump
      JMP   vWait          * keep waiting
jSkip1      SUBB  #$40
      STAB  VCOUNT               * Mod the count by 64 and save
      LDAA  YCOOR
      INCA                 * Move ball up by one
      STAA  YCOOR
      CMPA  #$7F           * Check for ball at edge of table
      BEQ   jSkip2
      JMP   vDone          * Branch out of range, must jump

jSkip2      LDAA  XCOOR          * Check for hitting paddle
      LDAB  KNOB2
      ADDB  #$05           * Right side of paddle
      CBA
```

```
      BGT   score1          * branch if a point is scored
      SUBB  #$0A        * Left side of paddle
      CBA
      BLT   score1          * branch if a point is scored

      LDAB  KNOB2       * Calculate velocity to add to ball
      LDAA  XCOOR       * as distance from ball to center of
      SBA               * the paddle times 4
      LSLA
      LSLA
      LDAB  HVEL
      ABA               * New horizontal ball speed
      CMPA  #$40        * Max speed 64
      BLE   vSkip1
      LDAA  #$40        * Set speed down to max
      BRA   vSHV

vSkip1      CMPA  #$C0      * Min speed -64
      BGE   vSHV
      LDAA  #$C0        * Set speed up to max
      BRA   vSHV


score1      LDAA  #$01      * Set port D bit 0 high to signal
      STAA  PORTD       * FPGA that player 1 scored
      BRA   vDone

vNeg  LDAB  VCOUNT
      SUBB  VVEL        * Update count
      CMPB  #$40        * Check count for ready to move ball
      BLT   vWait       * keep waiting
      SUBB  #$40
      STAB  VCOUNT          * Mod the count by 64 and save
      LDAA  YCOOR
      DECA              * Move ball down by one
      STAA  YCOOR
      CMPA  #$00        * Check for ball at edge of table
      BNE   vDone

      LDAA  XCOOR       * Check for hitting paddle
      LDAB  KNOB1
      ADDB  #$05        * Right side of paddle
      CBA
      BGT   score2          * branch if a point is scored
      SUBB  #$0A        * Left side of paddle
      CBA
      BLT   score2          * branch if a point is scored

      LDAB  KNOB1       * Calculate velocity to add to ball
      LDAA  XCOOR       * as distance from ball to center of
      SBA               * the paddle times 4
      LSLA
      LSLA
      LDAB  HVEL
      ABA               * New horizontal ball speed
      CMPA  #$40        * Max speed 64
      BLE   vSkip2
```

```
      LDAA   #$40         * Set speed down to max
      BRA    vSHV

vSkip2       CMPA  #$C0         * Min speed -64
      BGE    vSHV
      LDAA   #$C0         * Set speed up to min
      BRA     vSHV


score2       LDAA  #$02         * Set port D bit 1 high to signal
      STAA   PORTD        * FPGA that player 2 scored
      BRA    vDone

vSHV  STAA   HVEL         * Store horizontal velocity
      LDAA   #$00
      SUBA   VVEL         * flip vertical velocity
      STAA   VVEL
      BRA    vDone

vWait STAB   VCOUNT             * Save count and keep waiting

vDone RTS

UPDATE       JSR   CLRPIX            *Clear old position of ball

      LDAA   XCOOR        *Store current positions of ball which on the
next loop
      STAA   OLDX         *will be the old positions of the ball
      LDAA   YCOOR
      STAA   OLDY

      JSR    WRITE        *Write the position of the ball to the screen

      LDAA   YCOOR        *Preserve the value of y coordinate
      PSHA

      LDAA   #$7F         *Knobs are always either at 00 or 7f
      STAA   YCOOR
      JSR    CLRLINE            *Clear the old position of the paddle

      LDAA   #$00
      STAA   YCOOR
      JSR    CLRLINE            *Clear the old position of the paddle

      PULA
      STAA   YCOOR        *Restore old value of y coordinate

      LDAA   KNOB1        *X coordinate value of the centre of the first
knob
      STAA   KNOBX
      LDAA   #$00         *Place this paddle at the bottom of the screen
      STAA   KNOBY
      JSR    PADDLE             *Write the paddle to the screen

      LDAA   KNOB2        *X coordinate value of the centre of the second
knob
      STAA   KNOBX
```

```
      LDAA  #$7F        *Place this paddle at the top of the screen
      STAA  KNOBY
      JSR   PADDLE            *Write the paddle to the screen
      RTS

WRITE JSR   SETPAGE           *Sets values for the LCD Control and the
data
      JSR   LCDW        *Write these values to the LCD

      LDAB  YCOOR
      ORAB  #$40        *Change to set line instruction
      STAB  DATA
      JSR   LCDW        *Write these values to the LCD

      LDAA  LCDCTRL
      ORAA  #$02        *Changing from instruction to data
      STAA  LCDCTRL
      JSR   SETDATA           *Sets data value
      JSR   LCDW        *Write actual pixel to the LCD
      RTS

CLRPIX      LDAA  XCOOR       *Preserve values of x and y coordinates
      PSHA
      LDAA  YCOOR
      PSHA
      LDAA  OLDX        *Load in position that is to be cleared
      STAA  XCOOR
      LDAA  OLDY
      STAA  YCOOR
      JSR   SETPAGE           *Sets values for the LCD Control and the
data
      JSR   LCDW        *Write these values to the LCD

      LDAB  YCOOR
      ORAB  #$40        *Change to set line instruction
      STAB  DATA
      JSR   LCDW        *Write these values to the LCD

      LDAA  LCDCTRL
      ORAA  #$02        *Changing from instruction to data
      STAA  LCDCTRL
      LDAA  #$00        *Load in 0s which clears the page that being
written to
      STAA  DATA        *Sets data value
      JSR   LCDW        *Write these values to the LCD
      PULA              *Restore values of x and y coordinates
      STAA  YCOOR
      PULA
      STAA  XCOOR
      RTS

LCDW  LDAA  #$00        *Setup Port C to input
      STAA  DDRC

CHKW  LDAA  LCDCTRL           *Get LCD Control Pattern
      ANDA  #$38        *Alter it into the status check pattern
      ORAA  #$04
```

```
      STAA  PORTB       *Write the pattern to port B
      NOP
      NOP               *Wait one micro second
      ORAA  #$01        *Set E in the control bit high
      STAA  PORTB       *Write pattern to port B

      LDAB  PORTC       *Read the results
      BNE   CHKW        *Check until not busy anymore

      LDAA  #$FF        *Setup Port C to output
      STAA  DDRC

      LDAA  LCDCTRL           *Get LCD control pattern
      STAA  PORTB       *Write the pattern to Port B

      LDAB  DATA        *Get Instruction/Data
      NOP
      ORAA  #$01        *Set E in the control pattern high
      STAA  PORTB       *Write LCD control to port B
      STAB  PORTC       *Write the instruction/data to Port C

      ANDA  #$FE        *Drop the enable signal
      STAA  PORTB       *Write control to port B

      LDAA  #$38        *Control pattern idle state
      STAA  PORTB       *Write it to the LCD
      LDAB  #$00        *Put empty data on to the LCD lines
      STAB  PORTC
      PULA
      STAA  TMSK2
      RTS

CLRLCD      LDAA  #$80        *Max value of line number
      STAA  LINENUM
      LDAA  #$C0        *Max value of page number + 1
      STAA  PAGENUM
      LDAA  #$7F
      STAA  CURLINE           *Current Line number

OUTER LDAA  #$B8
      STAA  CURPAGE           *Current page number

INNER LDAA  CURLINE
      ANDA  #$40        *Get the value of the 6th bit
      BNE   UPPER       *Branch to upper which indicates that the CS1
lines should be set

      LDAA  #$28        *Set the CS2 lines
      STAA  LCDCTRL
      BRA   CONT

UPPER LDAA  #$30        *Set the CS1 lines
      STAA  LCDCTRL

CONT  LDAB  CURPAGE           *Current Page
```

```
      STAB  DATA        *Write the current page to the LCD so that LCD
knows where to write
      JSR   LCDW        *to when it gets the data

      LDAB  CURLINE           *Set line that should be written to
      ORAB  #$40        *Change to set line data
      STAB  DATA
      JSR   LCDW

      LDAA  LCDCTRL
      ORAA  #$02        *Changing from instruction to data
      STAA  LCDCTRL

      LDAB  #$00        *Write nothing to every pixel which effectively
clears LCD
      STAB  DATA
      JSR   LCDW
      INC   CURPAGE           *Increment the page
      LDAA  CURPAGE
      CMPA  PAGENUM     *See if the current page is the last page that
needs to be written to
      BNE   INNER       *Finished with that line

      LDAA  CURLINE           *Current line
      CMPA  #$00
      BEQ   END         *Gone through all the lines
      DEC   CURLINE           *Move the line that is being cleared
      JMP   OUTER
END   RTS

PADDLE      LDAA  XCOOR       *Preserve values of x and y coordinates
      PSHA
      LDAA  YCOOR
      PSHA
      LDAA  #$0B        *Paddle is 11 pixels long
      STAA  TOTAL       *Total number of pixels left to be written in
paddle
      LDAA  #$00        *No pixels are dark as yet
      STAA  DARK        *Number of paddle pixels that have been drawn
      LDAA  KNOBY       *Y coordinate of paddle
      STAA  YCOOR

      SUBA  #$05        *X coordinate of first pixel in paddle
      STAA  XCOOR

PNEXT LDAA  #$01        *Start number of times throught the routine off
at 1
      STAA  TIMES       *Number of consecutive times through the pwrite
routine

PWRITE      JSR   SETPAGE           *Sets values for the LCD Control
and the data
      LDAA  TIMES       *Amount of times this routine has looped
      DECA
      ADDA  DATA        *Increment pages according to the number of
times through the routine
```

```
      STAA  DATA        *so that each time this loop is executed the
next page over is being written
      JSR   LCDW        *to which is essential to draw the 11 pixels
which has to stretch over at least 2 pages

      LDAB  YCOOR
      ORAB  #$40        *Change to set line instruction
      STAB  DATA
      JSR   LCDW        *Write these values to the LCD

      LDAA  LCDCTRL
      ORAA  #$02        *Changing from instruction to data
      STAA  LCDCTRL

      LDAA  TOTAL       *Write the pixels that have not yet been
written
      STAA  DARK

      LDAA  TIMES
      CMPA  #$01        *Check if first time through the loop
      BEQ   FTIME

      LDAA  TOTAL
      CMPA  #$08        *See if the whole page needs to be written to
      BGE   TOOBIG
      JMP   GETD

FTIME LDAA  XCOOR
      ANDA  #$07        *Just want page data
      STAA  DARK

      JSR   PADDATA           *Get data to be written to the LCD
      LDAA  #$FF
      SUBA  DATA        *Invert data so that the paddle can be
connected in the two pages
      STAA  DATA
      JSR   LCDW        *Write these values to the LCD

      LDAA  #$08
      SUBA  DARK
      STAA  DARK        *Data and Dark both need to be inverted

      JMP   PCONT

TOOBIG      LDAA  #$08        *Write to the whole page
      STAA  DARK

GETD  JSR   PADDATA           *Sets data value
      JSR   LCDW

PCONT INC   TIMES       *Increment the number of times the loop has
executed
      LDAA  TOTAL       *Update the number of pixels left to be written
by subtracting
      SUBA  DARK        *the pixels that have just been written
      BEQ   PDONE       *No more pixels to be written
      STAA  TOTAL
```

```
      JMP   PWRITE

PDONE PULA              *Restore values of x and y coordinate
      STAA  YCOOR
      PULA
      STAA  XCOOR
      RTS



SETPAGE     LDAA  YCOOR
      ANDA  #$40        *Obtain value of 6th bit
      BNE   SECOND            *Branch to second which indicates that
the CS1 lines should be set

      LDAA  #$28        *Set to CS2 lines
      STAA  LCDCTRL
      JMP   NEXT

SECOND      LDAA  #$30        *Set to CS1 lines
      STAA  LCDCTRL

NEXT  LDAA  XCOOR
      ANDA  #$38        *The first 3 bits of the X coordinate indicate
the page since they're 8 pages per line
      LSRA              *Shift these 3 bits down to the end so they can
be manipulated
      LSRA
      LSRA
      ANDA  #$07        *Only want last 3 bits
      ORAA  #$B8        *Change it to page pattern
      STAA  DATA        *Write this data pattern to the LCD
      RTS

SETDATA     LDAA  XCOOR
      ANDA  #$07        *Just want page data

      LDAB  #$80        *Darken the pixel to the utmost right of the
current page

*In this loop the pixel that we want darkened is found by shifting the
pixel that is darkened to the left.
*This is done by rightshifting the value in accumulator value which
amounts to doing what's stated above.
*To get the right data within the page then the amount of shifts will
be the difference of 7 and the last
*3 bits of the X - coordinate.
LOOP1 CMPA  #$07
      BEQ   GOOD        *Shift until the accumulator has gotten to 7
      LSRB              *Shift the pixel that's darkened to the right
      INCA
      BRA   LOOP1
GOOD  STAB  DATA        *Store the pattern to be written to the LCD
      RTS

PADDATA     LDAA  DARK
```

```
      LDAB  #$FF        *Darken all the pixels in the current page

*In this loop the pixel that we want darkened is found by shifting the
pixel that is darkened to the left.
*This is done by rightshifting the value in accumulator value which
amounts to doing what's stated above.
*To get the right data within the page then the amount of shifts will
be the difference of 7 and the last
*3 bits of the X - coordinate.
PLOOP1      CMPA  #$08
      BEQ   PGOOD       *Shift until the accumulator has gotten to 8
      LSRB              *Shift the pixels that are darkened to the
right
      INCA
      BRA   PLOOP1
PGOOD STAB  DATA        *Store the pattern to be written to the LCD
      RTS

CLRLINE     LDAA  XCOOR       *Preserve values of x and y coordinates
      PSHA
      LDAA  YCOOR
      PSHA

      LDAA  #$01        *Start the counter at 1
      STAA  COUNT       *Number of consecutive times the Clear loop has
been executed

      LDAA  #$00        *Set the x coordinate to 0 which sets the page
to be the one to the utmost left
      STAA  XCOOR

CLEAR JSR   SETPAGE           *Sets values for the LCD Control and the
data
      JSR   LCDW        *Write these values to the LCD

      LDAB  YCOOR
      ORAB  #$40        *Change to set line instruction
      STAB  DATA
      JSR   LCDW        *Write these values to the LCD

      LDAA  LCDCTRL
      ORAA  #$02        *Changing from instruction to data
      STAA  LCDCTRL
      LDAA  #$00        *Write 0s to that page on the LCD which clears
that page on the LCD
      STAA  DATA
      JSR   LCDW        *Write these values to the LCD

      LDAA  XCOOR
      ADDA  #$08        *Advance to next page
      STAA  XCOOR
      LDAA  COUNT
      CMPA  #$08        *See if loop has executed 8 times (for the 8
pages that need to be written to)
      BEQ   CDONE
      INC   COUNT       *Increase the number of times the loop has
executed
```

```
      JMP   CLEAR

CDONE PULA                *Restore the values of the x and y coordinates
      STAA  YCOOR
      PULA
      STAA  XCOOR
                                   RTS
```

Verilog Code

```
/*
For all of the following modules:
Written by Philip Vegdahl
December 4, 2002
pvegdahl@hmc.edu
*/

module scoreTop(clk,reset,player1,player2,start,myseg,power);

/*
This is the top level module that sends all of the output
signals and receives all of the input signals. It uses lower
level modules to do all of the data crunching. The overall
module receives scoring signals from the HC11, updates the
score, and then returns a start signal to the HC11 so it will
know to start the game again. This signal will not be sent when
a player has reached 7 points, thus winning the game.
*/

    input clk;
    input reset;
    input player1;            // player 1 scores
    input player2;            // player 2 scores
    output start;             // tell HC11 to start game again
    output [6:0] myseg; //    7-seg output
    output [1:0] power; //    power switcher for 7-seg
    wire [3:0] score1, score2;


      scoreMem theScore(clk,reset,player1,player2,score1,score2,start);
      Lab3 theseg(clk,reset,score1,score2,myseg,power);

endmodule





module scoreMem(clk,reset,player1,player2,score1,score2,start);

/*
This module keeps track of, and updates the players scores.
It will also choose whether or not a start signal can be sent
based on whether or not a player has already won.
*/

    input clk;
    input reset;
    input player1;                  // Player 1 scores
    input player2;                  // Player 2 scores
    output [3:0] score1;      // Player 1's current score
    output [3:0] score2;      // Player 2's current score
    output start;                   // Tells HC11 to start game
```

```
    reg [3:0] score1;
    reg [3:0] score2;
    wire win;                              //  One  player  has  won  (7
points)
    wire allowScore;                 //    Score can be updated
    wire allowStart;                 //    FSM ready for game to start
    wire score;                      // A player has scored


      scoreFSM theScore(clk,reset,score,allowScore,allowStart);
      assign win = (score1==4'd7)|(score2==4'd7);
      assign start = allowStart&(~win);   // Don't start when a player
has won
      assign score = player1|player2;

      always@(posedge clk or posedge reset)
            if(reset) begin
                  score1 <= 0;
                  score2 <= 0;
            end
            else begin
                  score1 <= score1 + (player1&allowScore);
                  score2 <= score2 + (player2&allowScore);
            end

endmodule





module scoreFSM(clk,reset,score,allowScore,allowStart);

/*
This is a finite state machine that controls the scoring
handshake with the HC11. The states are as follows.

State 00:   Score has been updated and start is being sent.
                  Start will be lowered whenever the score signal
                  is lowered.

State 01:   Waiting for a score signal from the HC11.

State 10:   Score signal recieved from HC11, update score.
*/

    input clk;
    input reset;
    input score;              // A player score signal is high
    output allowScore;  // Allow the score to be changed
    output allowStart;  // Score has been updated and game can resume
    reg [1:0] state;          // Current state of FSM


      assign allowStart = (state==2'b00);
      assign allowScore = (state==2'b10);
```

```
      always@(posedge clk or posedge reset)
            if(reset) state <= 2'b00;
            else begin
                  state[0] <= ~score;
                  state[1] <= state[0]&score;
            end

endmodule





module Lab3(clk,reset,s0,s1,myseg, power);

/*
This module takes 2 4-bit binary numbers as user inputs and
outputs signals to display both numbers on different
7-segment displays using only one piece of decoding hardware.
*/

    input clk, reset;
    input [3:0] s0, s1; // the 2 binary input signals
    output [6:0] myseg; // output to 7-segment displays
    output [1:0] power; // controlers for which display to power
    wire [3:0] s2;            // wire carying the binary input in use
    wire sel;                       //  selector  for  the  mux  and  power
signals

      slow_clk theclk(clk, reset, sel);   // slower clock used to

      // switch between displays
      mux2_4 themux(s0,s1,sel,s2);              //  mux  to  select  input
signal
      seg theseg(s2, myseg);                          //       7-segment
output
      assign power = {~sel, sel};               //    power    selection
signals


endmodule





module seg(s,seg);

/*
This is the hardware to decode a 4-bit binary number into a
single digit hexadecimal output.
*/

    input [3:0] s;            // Binary input
    output [6:0] seg;   // 7-segment output
```

```
//These are the different segment selection signals

assign seg[0] = ((~s[3] & ~s[2] & ~s[1] & s[0]) |     // 1
                                    (~s[3] & s[2] & ~s[1] &
~s[0]) |    // 4
                                    (s[3] & ~s[2] & s[1] &
s[0]) |     // B
                                    (s[3] & s[2] & ~s[1] &
s[0]));     // D

assign seg[1] = ((~s[3] & s[2] & ~s[1] & s[0]) |      // 5
                                    (~s[3] & s[2] & s[1] &
~s[0]) |    // 6
                                    (s[3] & ~s[2] & s[1] &
s[0]) |     // B
                                    (s[3] & s[2] & ~s[1] &
~s[0]) |    // C
                                    (s[3] & s[2] & s[1] &
~s[0]) |    // E
                                    (s[3] & s[2] & s[1] &
s[0]));           // F

assign seg[2] = ((~s[3] & ~s[2] & s[1] & ~s[0]) |     // 2
                                    (s[3] & s[2] & ~s[1] &
~s[0]) |    // C
                                    (s[3] & s[2] & s[1] &
~s[0]) |    // E
                                    (s[3] & s[2] & s[1] &
s[0]));           // F

assign seg[3] = ((~s[3] & ~s[2] & ~s[1] & s[0]) |     // 1
                                    (~s[3] & s[2] & ~s[1] &
~s[0]) |    // 4
                                    (~s[3] & s[2] & s[1] &
s[0]) |     // 7
                                    (s[3] & ~s[2] & ~s[1] &
s[0]) |     // 9
                                    (s[3] & ~s[2] & s[1] &
~s[0]) |    // A
                                    (s[3] & s[2] & s[1] &
s[0]));           // F

assign seg[4] = ((~s[3] & ~s[2] & ~s[1] & s[0]) |     // 1
                                    (~s[3] & ~s[2] & s[1] &
s[0]) |     // 3
                                    (~s[3] & s[2] & ~s[1] &
~s[0]) |    // 4
                                    (~s[3] & s[2] & ~s[1] &
s[0]) |     // 5
                                    (~s[3] & s[2] & s[1] &
s[0]) |     // 7
                                    (s[3] & ~s[2] & ~s[1] &
s[0]));     // 9

assign seg[5] = ((~s[3] & ~s[2] & ~s[1] & s[0]) |     // 1
```

```
                                                            (~s[3] & ~s[2] & s[1] &
~s[0]) |    // 2
                                                            (~s[3] & ~s[2] & s[1] &
s[0]) |     // 3
                                                            (~s[3] & s[2] & s[1] &
s[0]) |     // 7
                                                            (s[3] & s[2] & ~s[1] &
s[0]));     // D

            assign seg[6] = ((~s[3] & ~s[2] & ~s[1] & ~s[0]) |    // 0
                                                            (~s[3] & ~s[2] & ~s[1]
& s[0]) |   // 1
                                                            (~s[3] & s[2] & s[1] &
s[0]) |     // 7
                                                            (s[3] & s[2] & ~s[1] &
~s[0]));    //    C

endmodule
```

```
module slow_clk(clk,reset,new_clk);

/*
This hardware creates a new clock signal that runs 1024 times slower
than
the original clock. This prevents timing problems from switching back
and
and forth quickly.
*/

    input clk, reset;   // basic clock and reset
    output new_clk;           // outputed slower clock
    reg [9:0] counter;  // counter to keep track of timing on new clock

      always@(posedge clk or posedge reset)
            if(reset) counter = 0;  // reset the counter to zero
            else counter = (counter+1) % 1024;  // add to the clock and
wrap around

            // to 0 whenever it hits 1024

      assign new_clk = counter[9];  //  Output  signal  is  MSB  of  the
counter.

endmodule
```

```
module mux2_4(d0,d1,sel,y);

/*
This is just a two-way, 4-bit multiplexor.
```

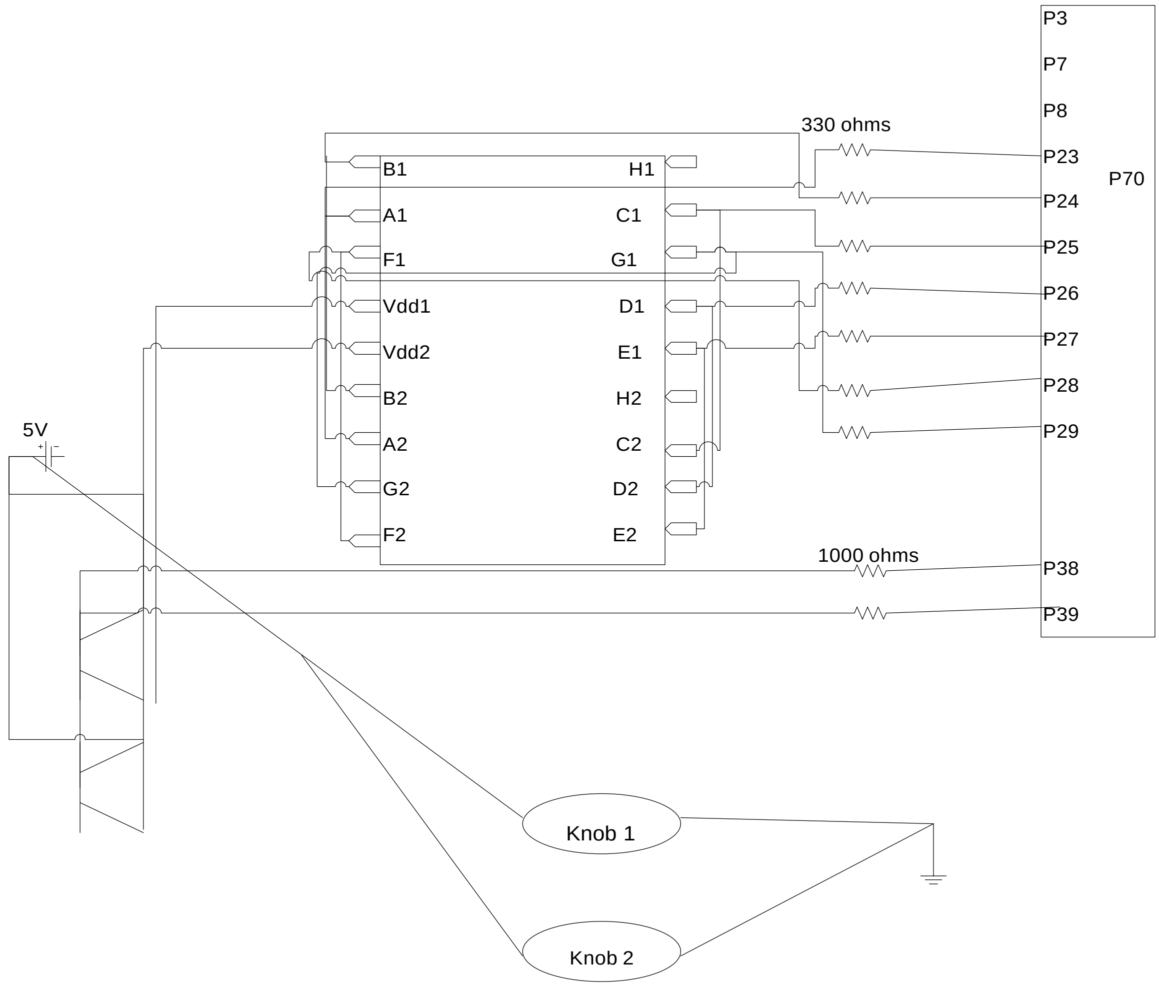
```
*/

    input [3:0] d0;     // Mux input 1
    input [3:0] d1;     // Mux input 2
    input sel;                // Selection signal
    output [3:0] y;     // Output signal

      assign y = sel?d1:d0;   // Chooses the correct output from the
                                                       //       selection
signal.

Endmodule
```

Breadboard Schematic



Crystalfontz Graphical LCD

Overview

For this project we used a 128x64 pixel LCD, part number #CFAG12864B-WGH-V, which is available from www.crystalfontz.com. This LCD contains two 64x64 Samsung S6B0108 chips placed side by side which drive the display. As a result there are 128 lines on the LCD, with line 0 being the edge of the screen nearest to pin 1 and line 64 (line 0 of the 2nd chip) being in the middle of the screen. Each line contains 8 pages of data with each page containing 8 pixels each. A page of data must be addressed all at once. A logic value of '1' being written to a certain pixel on the screen means that the pixel is darkened. The pixels all retain their value until another value is explicitly written to them. When the LCD is first powered up, the display RAM in the chips have the value 1 for every pixel. However, even though the data is in the display RAM this pattern isn't seen on the LCD until the LCD is given the write control pattern and the display on instruction.

PinOut

Pin 1: +5V

Pin 2: GND

Pin 3: -2.5V to -4.5V, where -2.5V is a light background and -4.5V is a fully darkened background.

Pin 4-11: Data Bus Bits 0-7: Sends instructions or data to be written to the LCD. Returns status flags or data read from the LCD.

Pin 12: CS1 – Column Select 1: Active low control signal that selects the first S6B108 device and so writes to lines 0 to 63.

Pin 13: CS2 – Column Select 2: Active low control signal that selects the second S6B108 device and so writes to lines 64 to 127.

Pin 14: R – reset: an asynchronously low reset signal that turns off the screen and resets the line scroll register.

Pin 15: R/W – Read or Write. 1 indicates read from LCD data bus, 0 indicates write to LCD data bus.

Pin 16: D/I – Data or Instruction. 1 indicates data is being sent the LCD, 0 indicates that an instruction is being sent to the LCD.

Pin 17: E – enable: acts as a clocking signal, that is the signal needs to be high for anything to happen to the LCD.

Pin 18: Negative Voltage Output (Not used in this project)

Pin 19: Positive Power for LED Backlight (Not used in this project)

Pin 20: Negative Power for LED Backlight (Not used in this project)

General Operations

There are two main operations that can be done with this LCD – reads and writes. However, reading from the LCD is something that wasn't used in this project and as a result I'm not able to provide much information on this function of the LCD. The E signal acts as the clock for the device. As a result, the E signal should idle low and only be changed when an instruction is ready to be started. On a suggestion from Aaron Stratton, an idle pattern of {R = 1, CS1 = 1, CS2 = 1, R/W = 0, D/I = 0, E = 0} was written to the LCD between instructions. This idle pattern ensures that nothing can be written to the LCD as neither column is selected. Each instruction done on the LCD followed this basic format:

1. Set the Control Pattern (R, CS1, CS2, R/W, D/I, E). Reset should be high and enable should be low at this point. The other values will depend on what the user is trying to get the LCD to do.
2. Raise the Enable signal.
3. Send the Data/Instruction that needs to be written to the LCD.
4. Lower the Enable signal.

Status Checks

When executing instructions that affect the LCD display, a status check needs to be made, as if the LCD is busy when an instruction is sent then that instruction is ignored. The only place in the Pong game that a status check is used is when trying to write to LCD (in this context, writing to LCD includes writing instructions, not only writing pixels to the screen). This status check keeps on executing until the LCD is no longer has the busy flag set indicating that its busy. Aaron Stratton found an error in the S6B0108 documentation in Version 0.0 on page 16. In the documentation it was stated that the busy flag was set on the falling edge of E but in actuality its set on the rising edge of E.