Vector-based Pong on an Oscilloscope

Edmond Lau
6.115 Final Project
May 13, 2004

Table of Contents

1 Introduction 3

2 Hardware Design Description 3
  2.1 Using the Oscilloscope Display in XY Mode 4
  2.2 Interfacing the D/A Converters with the Oscilloscope Display 4
  2.3 Creating Potentiometer Controllers to Move the Paddles 5
  2.4 Integrating a Sound System into Pong 7

3 Software Design Description 7
  3.1 Refreshing the Oscilloscope Display 9
  3.2 Loading and Drawing Vector-based Graphics 9
  3.3 Dealing with Resolutions in Ball Velocity 11

4 Possible Design Extensions 11

5 Conclusion 12

6 Appendix 13
  6.1 Hardware Schematics – System Core 13
  6.2 Hardware Schematics – Auxiliary Sound System 14
  6.3 Assembly Code for pong.asm 15

List of Figures

1 Screenshot of Oscilloscope Pong 3
2 Block Diagram of System Hardware 4
3 Coordinate System of the Oscilloscope Display 5
4 Potentiometer Controller for the Paddle 6
5 Audio Amplifier Circuit 7
6 Flowchart of Software Control Logic 8

1 Introduction

In 1972, the cofounder of Atari, Nolan Bushnell, launched the video-game revolution with the arcade game Pong and set out on the path to become the father of the video-game industry. Although he developed the original Pong game for television consoles, the world's first video game was actually constructed using a laboratory oscilloscope as a display medium. Inspired by Bushnell's attempts, I recreated the original Pong, but on an oscilloscope rather than on a TV for my final project.

Building Pong using vector-based graphics on an oscilloscope provided a fun and enjoyable project with both visible and playable results. Moreover, it enabled me to explore a new and exciting use of the oscilloscope and introduced me to the world of vector-based graphics. The traditional Pong video game involves two players controlling paddles on opposite sides of the screen, trying to score a goal by bouncing a ball past the opponent's paddle. The ball accelerates on each subsequent bounce against a paddle, and the first player to score seven points wins. Figure 1 shows a screenshot of my Pong game. My original goal had simply been to build a functioning version of Pong, but I succeeded in integrating collision and goal sounds as well as a scoreboard on the oscilloscope as well.

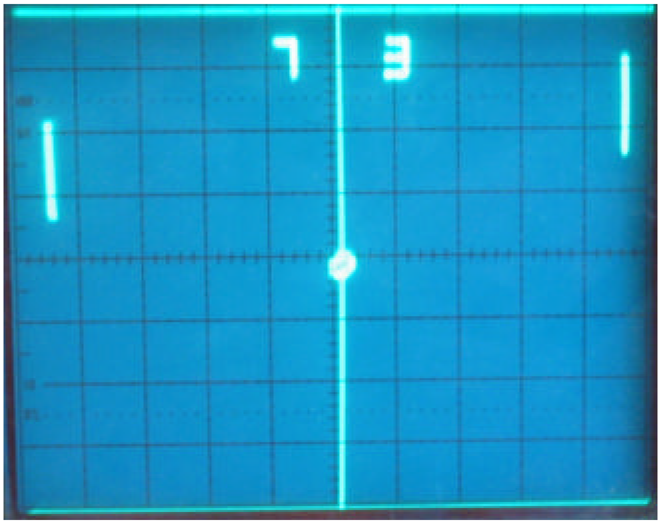


Figure 1: Screenshot of Oscilloscope Pong.

2 Hardware Design Description

In this section, I describe the hardware used for the core system components, including the configuration for the oscilloscope, the D/A converters used to draw graphics, and the potentiometers used as game controllers. Figure 2 shows the block diagram for the hardware structure. The lab kit's two built-in potentiometers connect to separate A/D converters and function as the paddle controllers. Two D/A converters connect to channels 1 and 2 of the analog oscilloscope and serve as the drawing tools. A third D/A converter drives an audio amplifier circuit to play collision and goal sounds.

The full schematic for the hardware is included in Appendix 6.1.

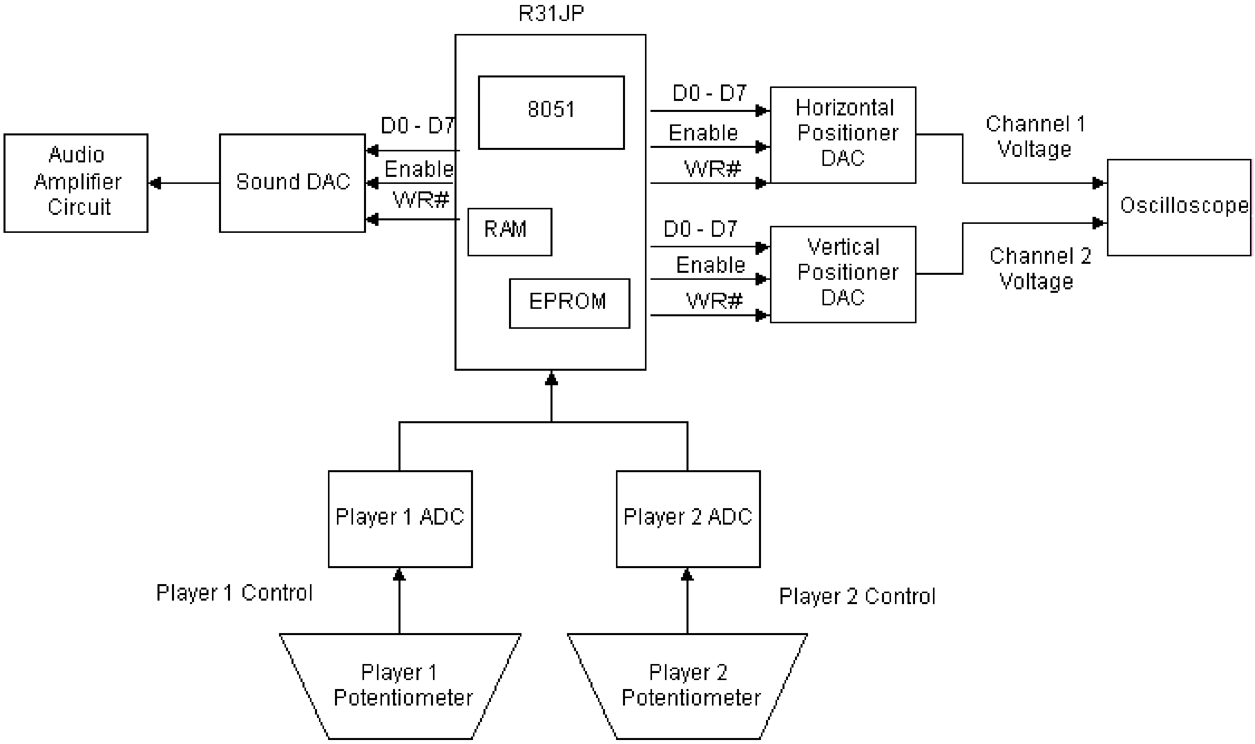


Figure 2: Block Diagram of System Hardware.

2.1 Using the Oscilloscope Display in XY Mode

Conventional usage of the oscilloscope runs in YT mode, where each voltage signal detected on the various probes is displayed as a function of time. However, most oscilloscopes also have an XY mode, which allows inputs to be plotted as functions of each other; the voltage reading on one probe determines the horizontal component and the reading on the other determines the vertical component. According to common oscilloscope manuals, the XY mode is used primarily for measuring the phase shift of two input waveforms.

For my video game system, I used the XY mode to generate graphics on an analog oscilloscope. For example, by generating two 90 degree off-phase sinusoids with the 8051, I traced out the image of a circular ball on the screen. By varying the offsets, the 8051 can then change the position of the ball around the screen to simulate movement. The major challenge in implementing the system was to generate a sufficiently fast refresh rate so that the oscilloscope screen did not appear to flicker.

2.2 Interfacing the D/A Converters with the Oscilloscope Display

For the graphics display, I used two AD558 D/A converters to generate voltages to the oscilloscope probes; the voltage output of one chip determined the horizontal X component and the voltage output of the other determined the vertical Y component.

The first major design consideration involved determining how to configure the D/A converters and the resolution and offset settings of the oscilloscope to create the video game display's coordinate system. Since the D/A converters could only generate positive voltages, I configured the offset settings to place the XY mode origin at the lower left corner of the screen. Solving the problem of how to configure the D/A converters and the oscilloscope resolution settings entailed balancing three constraints:

1. The AD558 maps a digital input range of 00h to FFh to either an analog output range of 0 – 2.56V or 0 – 10V. This output range needs to cover the entire visible portion of the oscilloscope screen.
2. The analog oscilloscopes provide eight vertical divisions and ten horizontal divisions, and the resolutions of interest are limited to values of 50 mV, 100 mV, 200 mV, 500 mV, and 1V.
3. The higher the screen resolution, the sharper the graphics will be.

After some calculations, I determined that by using the 0 – 2.56V configuration of the D/A converters and by setting the voltage resolutions on channels 1 and 2 to 200mV per division, I could create a low power 160x200 display. Figure 3 illustrates the coordinate map that I used for the oscilloscope; because each of the D/A converters provided a 10 mV resolution at the output range 0 – 2.56V, I could essentially output 20 discrete points for each 200 mV division.

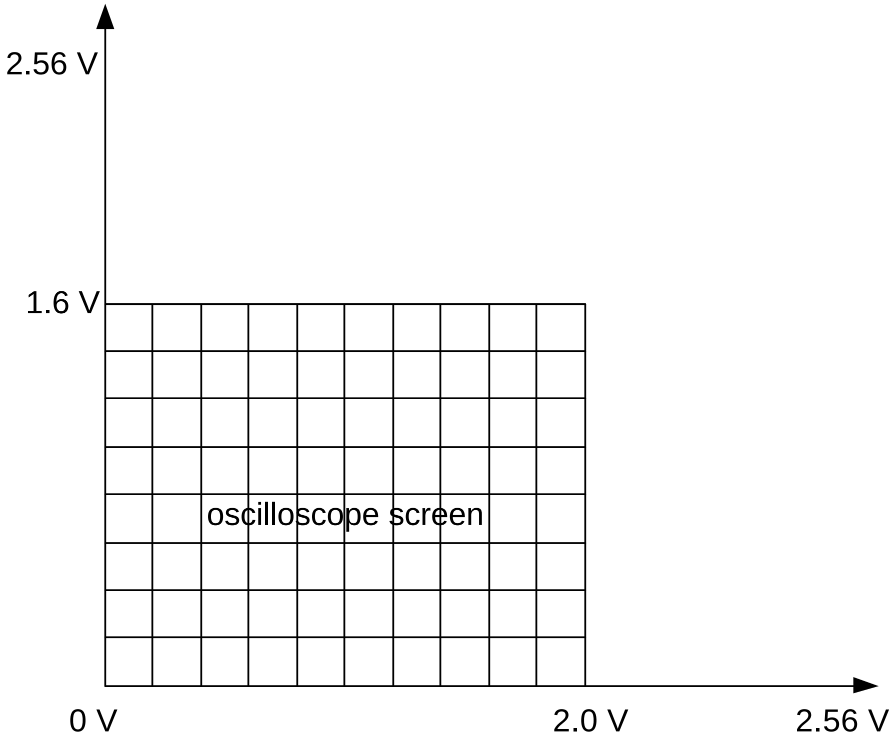


Figure 3: The Coordinate System of the Oscilloscope Display.

2.3 Creating Potentiometer Controllers to Move the Paddles

The design of the paddle controllers involved another major design decision. Conventional video game experiences suggest using two buttons on a game controller, keyboard, or keypad; a player holds down one of the buttons to move his paddle either up or down. This would involve either triggering external interrupts to notify the 8051 that a button has been pushed or using a polling strategy to check for button presses.

This approach has fundamental drawbacks from the perspectives of software implementation and the user's gaming experience. A strategy using external interrupts introduces the additional programming complexity of have to deal with the special case where a player drains too much CPU time by holding a button down too long; the software must turn the external interrupt off periodically so that it can continue running the control logic. A polling strategy avoids this issue but makes the speed of paddle movement constant: the paddle speed is constrained to be proportional to the polling rate.

To remedy this problem, I used a little creativity to design controllers that would not drain precious CPU cycles and that would moreover provide variable speeds for paddle movement. The solution, illustrated in Figure 4, involved using the lab kit's two built-in potentiometer knobs as the paddle controllers. By twiddling the potentiometer knob, a variable voltage in the range of 0 – 5V is input to an A/D converter that converts it to a digital value in the output range of 00h – FFh to determine the paddle position; I duplicated this idea to make two paddles. A polling strategy is used to update the paddle's position, but because the speed of paddle movement is only determined by the speed with which a player turns the knob, game improves substantially.

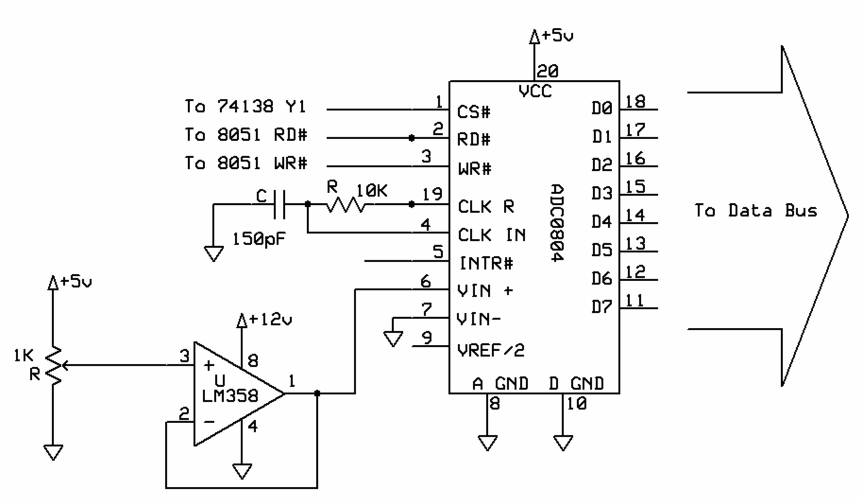


Figure 4: Potentiometer Controller for the Paddle.

The other half of this solution involves the software interface to the A/D converters. Denoting the paddle length as `P_LENGTH`, the 8051 can then perform the following mathematical calculation to convert the digital output reading to a value in the coordinate system denoting the y-coordinate of the bottom of the paddle:

$$y - coord = \frac{A/D\ output * (160 - P\_LENGTH)}{256}$$

This calculation can be performed simply by multiplying the A/D output by (160-`P_LENGTH`) and taking the high-order byte, as shown in the following `adcToVal` routine that interfaces with the potentiometer controllers:

```
;==============================================================
; ADCTOVAL: reads the ADC at dptr and outputs the coordinate value
;       in Pong coordinates to acc
; input: dptr (ADC), SCALING_FACTOR (Y_MAX – P_LENGTH)
; destroys: dptr, a, b, r4
; outputs: acc (val)
;==============================================================
adcToVal:
    movx @dptr, a               ; fire up the adc
    mov r4, #08
_waitADC:
    djnz r4, _waitADC
    movx a, @dptr
    mov b, #SCALING_FACTOR      ; convert from [0-255] to [0-SCALING_FACTOR]
    mul ab                      ; val = ADC_out*SCALING_FACTOR/256 = b
    mov a, b
    ret
```

2.4 Integrating a Sound System into Pong

No video gaming system is complete without integrated sound. An integrated sound system was one of the additional features I implemented for my video game system (with the scoreboard being the other). To play sounds, I connected a third D/A converter to an audio amplifier circuit as shown in Figure 5.

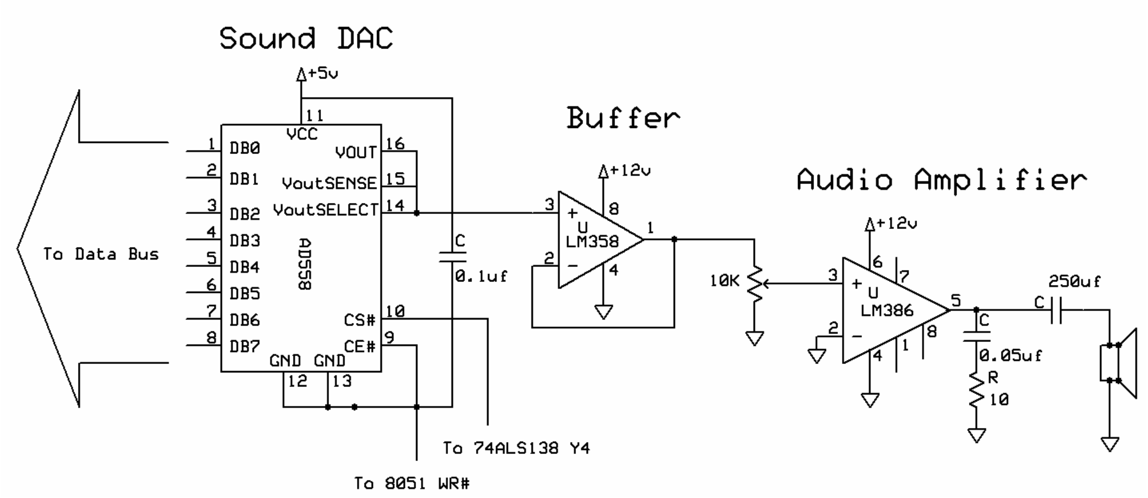


Figure 5: Audio Amplifier Circuit.

The Pong gaming system generated an approximately 900 Hz beep on the kit's speaker during collisions and a 1.2 kHz beep whenever a goal was scored. Noise in the audio amplifier circuit added additional frequencies to the otherwise monotone sounds and actually improved the sound quality.

3 Software Design Description

In this section, I describe the software control loop that brings the gaming system to life in addition to some salient features of my code. Figure 6 illustrates the software control logic used to execute Pong. When the R31JP is reset (or first switched to RUN mode), the gaming system resets both scores to zero and waits for the start button to be pressed, constantly refreshing the scores, the ball, and the paddles in the process.

Upon detecting a depressed start button, the control logic executes the following initialization algorithm to start a round:

1. Reset the ball location to the middle of the screen.
2. Check if either player has reached seven points; if so, end the game.
3. Load the players' score for the current round into RAM.
4. Serve the ball by initializing it with a default velocity in the direction of the previous round's loser. The ball travels to the right on the first round.

Next, the control logic iterates over the following loop until a goal has been scored:

1. Update the positions of the both paddles by reading the A/D converters connected to the potentiometer controllers.
2. Detect ball collisions with the walls; if a collision is detected, change the direction of the vertical component of the ball's velocity and play a low frequency collision sound.
3. Detect collisions with the paddles; if a collision is detected, increase the ball's velocity, change the direction of the horizontal component of velocity, and play a low frequency collision sound.
4. Update the location of the ball based on the current velocity.
5. Detect whether the ball has entered the goal (i.e. reached the edge of the screen). If so, play a higher frequency sound, increment the winner's score, and jump to step 1 of the initialization algorithm for starting a new round.
6. Load the graphics data for the updated paddles and the ball into the RAM.
7. Wait for the refresh interrupt to trigger and refresh the oscilloscope display before jumping back to step 1.

A `refreshISR` interrupt service routine, which is not shown in the diagram, draws the walls, the midfield line, the paddles, the ball, and the scores at a 70 Hz rate.

The Pong assembly code is included in Appendix 6.2.

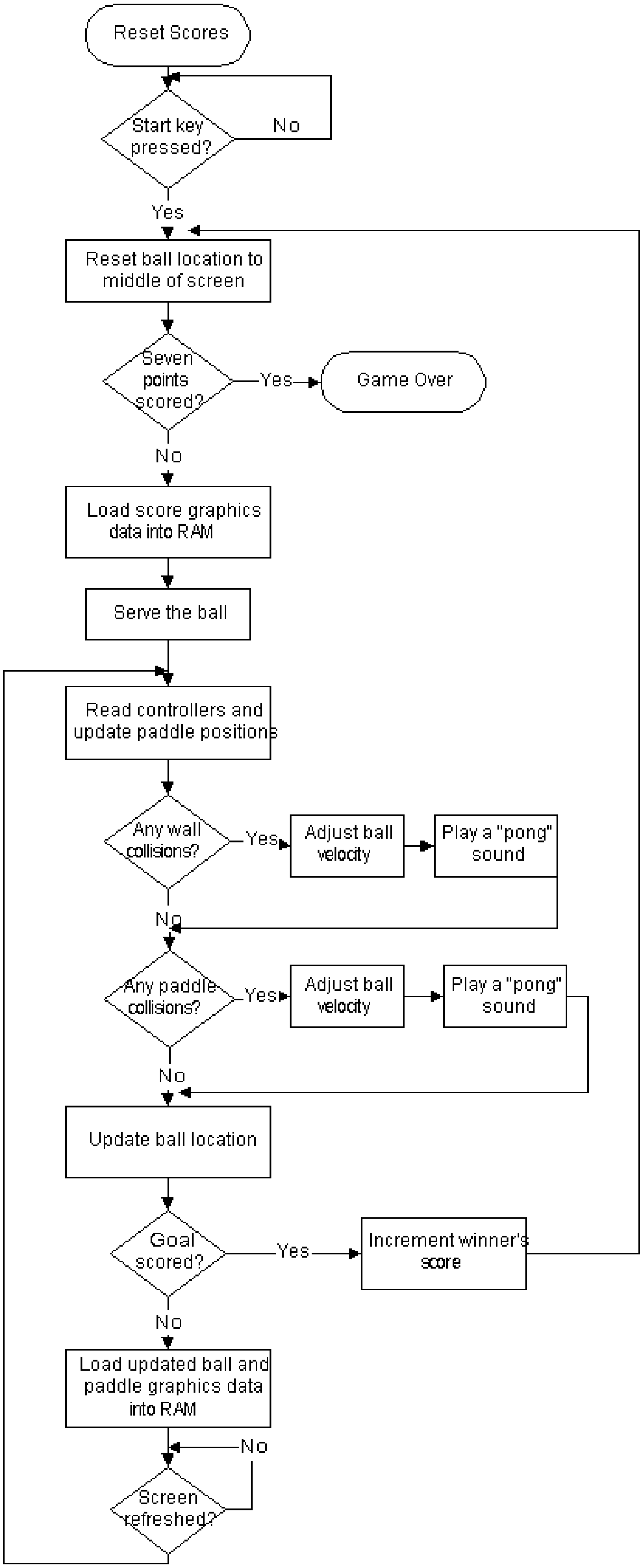


Figure 6: Flowchart of Software Control Logic.

In the following subsections, I highlight and explain some of the more details and design considerations that factored into the above control logic.

3.1 Refreshing the Oscilloscope Display

The major design risk and challenge in implementing the Pong gaming system was to generate a sufficiently fast refresh rate so that the oscilloscope screen did not appear to flicker. Using the 70Hz refresh rate of my LCD monitor as the basis, I set up the `refreshISR` interrupt, which redraws the screen, to fire 70 times per second. This constrained the amount of processing time for reading the controllers, detecting collisions, loading graphics data to the RAM, playing sounds, and sending the graphics data to the DACs to 13165 machine cycles. By using the potentiometer controllers, which did not require additional machine cycles to process external interrupts, and by limiting the sounds to simple square waves, I mitigated the effects of this limitation and ultimately succeeded in running both the control logic and the refresh logic within the limited number of machine cycles.

3.2 Loading and Drawing Vector-based Graphics

TV-based video games and most electronic images use raster images, where the image is determined by the colors of various pixels on the screen. Vector-based images, on the other hand, consist of mathematical descriptions of points and lines. Oscilloscope Pong uses vector-based images for the ball and the score. The data for the ball and score's representation is coded into the pong.asm file as program data. To simulate ball movement and to use one set of score data for both players, a horizontal and vertical offset is applied to the vector data before loading it into RAM.

To illustrate the concept of vector-based graphics, I describe, as an example, how the data for the ball is loaded into RAM and drawn onto the oscilloscope. The graphics data for the ball is represented in the following piece of assembly code:

```
ballX3:
db 00h, 00h, 01h, 02h, 03h, 04h, 05h, 06h, 06h
db 06h, 05h, 04h, 03h, 02h, 01h, 00h, 0ffh

ballY3:
db 03h, 04h, 05h, 06h, 06h, 06h, 05h, 04h, 03h
db 02h, 01h, 00h, 00h, 00h, 01h, 02h, 0ffh
```

`BallX3` denotes the sequence of x-coordinates and `ballY3` denotes the corresponding sequence of y-coordinates used for drawing a ball with radius 3 at the lower left corner of the screen. The 0ffh at the end of each data table is a discipline that I developed for framing the graphics data, i.e. encoding where the graphics data ends; the discipline removes the need to hardcode into the program the number of points that need to be loaded from the data table into the appropriate place in RAM.

Using the graphics data, the main control loop then executes the `loadBall` subroutine to copy the data into RAM locations `BALL_X_VECTOR` and `BALL_Y_VECTOR`. The data from the above tables are first offset by the horizontal and vertical coordinates of the ball's current position, `M_BALL_X` and `M_BALL_Y`, respectively, prior to being loaded to the RAM. For instance, the actual x-values loaded to RAM are calculated as follows:

```
x-value = x-value from ballX3 + ball’s x-location – ball’s radius
```

The following code snippet performs the `loadBall` subroutine:

```
;===============================================================
; LOADBALL: loads the vector data for the ball into RAM
; destroys: a, dptr, P2, r0, r4, r5, b, c
;===============================================================
loadBall:
    mov r0, #0h                ; initialize input offset r0 to 0
    mov dptr, #ballX3          ; set input pointer to ballX table
    mov P2, #BALL_X_VECTOR_HI  ; set output pointer to vector
_loadBallXLoop:
    mov a, r0                  ; load the r0-th point from table
    movc a, @a+dptr
    cjne a, #0ffh, _loadBallXOK ; terminating value found?
    sjmp _endLoadBallX
_loadBallXOK:
    add a, M_BALL_X            ; offset the ball's location
    clr c                      ; x-value = val + offset - radius
    subb a, #BALL_RADIUS

    movx @r0, a
    inc r0
    sjmp _loadBallXLoop        ; keep loading
_endLoadBallX:
    mov a, #0ffh               ; copy terminating character over
    movx @r0, a

    mov r0, #0h                ; initialize input offset r0 to 0
    mov dptr, #ballY3          ; set input pointer to ballY table
    mov P2, #BALL_Y_VECTOR_HI  ; set output pointer to vector
_loadBallYLoop:
    mov a, r0                  ; load the r0-th point from table
    movc a, @a+dptr
    cjne a, #0ffh, _loadBallYOK ; terminating value found?
    sjmp _endLoadBallY
_loadBallYOK:
    add a, M_BALL_Y            ; offset the ball's location
    clr c                      ; x-value = val + offset - radius
    subb a, #BALL_RADIUS

    movx @r0, a
    inc r0
    sjmp _loadBallYLoop        ; keep loading
_endLoadBallY:
    mov a, #0ffh               ; copy terminating character over
    movx @r0, a
    ret
```

In the `refreshISR` interrupt service routine, the 8051 then alternates between sending a value to the *x*-coordinate DAC and a value to the *y*-coordinate DAC using the data loaded into RAM:

```
;===============================================================
; DRAWBALL: draws the ball to the scope
; destroys: P2, r0, r1, r3, dptr, a, dph2, dpl2
;===============================================================
drawBall:
    mov dptr, #BALL_X_VECTOR
    mov DPH2, #BALL_Y_VECTOR_HI
    mov DPL2, #00
    lcall drawXY
    ret
```

```
;============================================================
; DRAWXY: draws the values at dptr and dph2:dpl2
; destroys: P2, r0, r1, r3, a
;============================================================
drawXY:
    mov P2, #0FEh
    mov r0, #X_DAC_LO          ; set up pointers to X_DAC and Y_DAC
    mov r1, #Y_DAC_LO
    mov r3, #0                 ; initialize offset to 0
_drawXYLoop:
    mov a, r3                  ; set acc to offset
    movc a, @a+dptr            ; get the next x-value
    cjne a, #0ffh, _drawXYOK   ; terminating character?
    sjmp _endDrawXY
_drawXYOK:
    movx @r0, a                ; send the x-value to the X_DAC

    mov a, r3                  ; set acc to offset
    push dph
    push dpl
    mov dph, DPH2
    mov dpl, DPL2
    movc a, @a+dptr            ; get the next y-value
    pop dpl
    pop dph
    movx @r1, a                ; send the y-value to the Y_DAC
    inc r3
    sjmp _drawXYLoop

_endDrawXY:
    lcall clearCursor
    ret
```

3.3 Dealing with Resolutions in Ball Velocity

One interesting design challenge involved determining how to represent ball velocity. I had already straightforwardly decided to represent ball position using two 8-bit numbers, one for the horizontal position and one for the vertical position, in the 160x200 coordinate space. Inspired by the vector-related ideas for the graphics, the obvious choice would have been to also use two 8-bit numbers for the velocity, one for the horizontal component and the other for the vertical component. For each unit of velocity, the ball would then move that many units in the 160x200 grid for every update. However, the problem with this representation was that the resolution was too coarse; a speed of four would already be extremely fast for a Pong game.

To solve this problem, I instead used two 16-bit numbers to represent ball position. At each update, I added the 8-bit speed values to the 16-bit numbers, and used the high bytes to determine the actual ball position. By itself, this would only allow speeds ranging from 0 to 1. To support higher speeds, I added a loop to the control logic that iterated over the collision detection code and ball update code multiple times before finally loading the new ball positions into RAM.

4 Possible Design Extensions

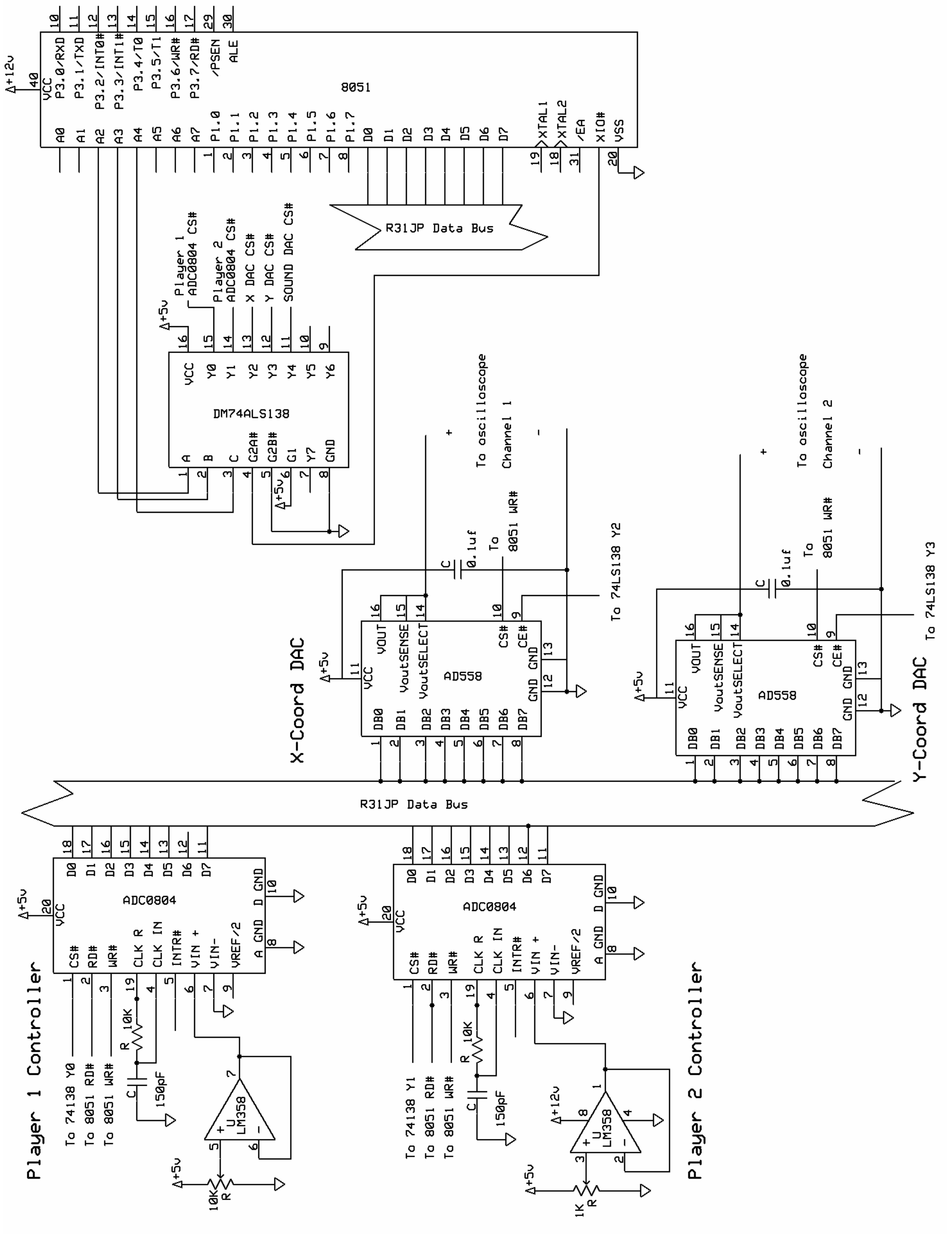
The major feature that I would have loved to add to the Pong game would have been support for ball spin. The current version of Pong assumes a physics model in which the paddles apply no frictional force to the ball upon collision. Constructing a more complicated physics model in which the paddles could indeed apply frictional forces would enable players to change the trajectory and velocity of the ball. This extension would improve game play by incorporating an additional dimension of different techniques of hitting the ball.

5 Conclusion

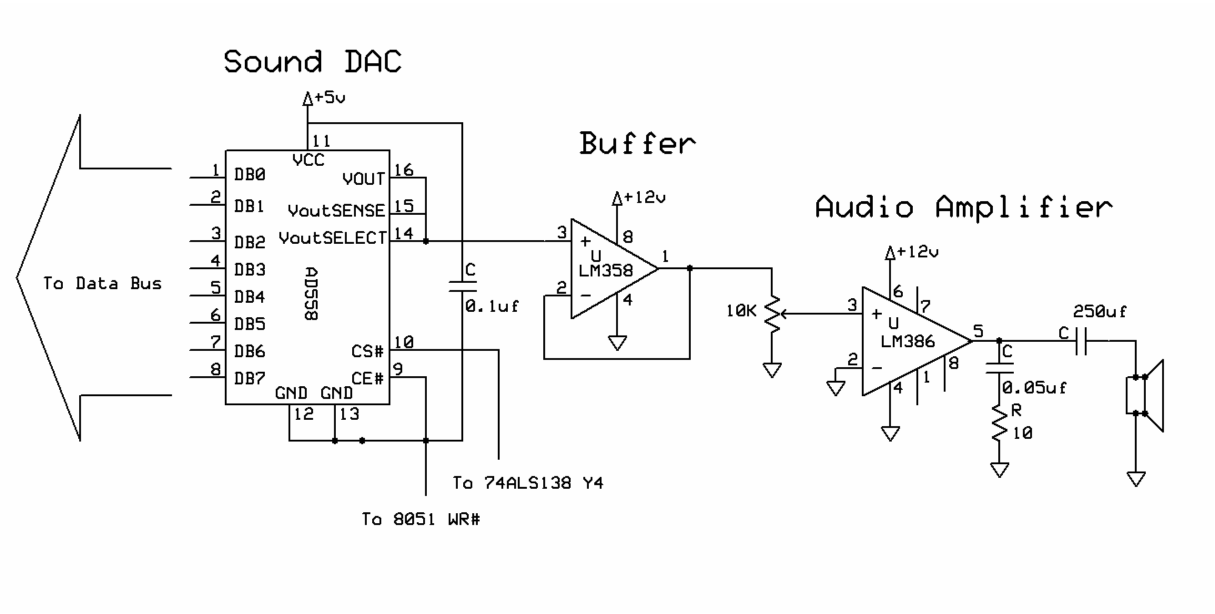
In this project, I developed a Pong video game system with integrated sound and scoreboard. I explored additional functionality on the analog oscilloscopes and discovered an infrequently used application of the oscilloscope display. I gained an introduction to the world of vector-based graphics in drawing the various paddles, balls, and scores. Most importantly, I successfully built a complete system that works.

6 Appendix

6.1 Hardware Schematics – System Core



6.2 Hardware Schematics – Auxiliary Sound System



6.3 Assembly Code for pong.asm

```
;   ************************************************
;   *                                              *
;   *  Vector-based Pong on an Oscilloscope        *
;   *  6.115 - Final Project                       *
;   *                                              *
;   *  Massachusetts Institute of Technology       *
;   *  Edmond Lau,  May 2004                       *
;   *                                              *
;   ************************************************

;; Conventions:
;; r0-r3 are used by interrupts
;; r0, r1 and r4-r7 are used by main program
;; 0ffh is special terminating character

;================================================================
;   PERIPHERALS
;================================================================

P1_ADC equ 0FE00h
P2_ADC equ 0FE04h

X_DAC equ 0FE08h
X_DAC_LO equ 08h
Y_DAC equ 0FE0Ch
Y_DAC_LO equ 0Ch
SOUND_DAC equ 0FE10h

;================================================================
;   STATIC CONSTANTS
;================================================================

X_MAX equ 200
X_MIN equ 0
Y_MAX equ 160
Y_MIN equ 0

P1_X equ 10                     ; X location of paddle 1
P2_X equ 190                    ; X location of paddle 2
P_LENGTH equ 30                 ; length of paddle
SCALING_FACTOR equ 130          ; set to Y_MAX - P_LENGTH

P1_SCORE_X equ 80
P1_SCORE_Y equ 140
P2_SCORE_X equ 115
P2_SCORE_Y equ 140

BALL_RADIUS equ 3
DEFAULT_SPEED_X equ 90
DEFAULT_SPEED_Y equ 90
N_BALL_POINTS equ 12

;================================================================
;   VARIABLES, MEMORY LOCATIONS
;================================================================

;; M's should never have #'s preceding them

M_BALL_X equ 60h
M_BALL_Y equ 61h
M_BALL_SPEED_X equ 62h
M_BALL_SPEED_Y equ 63h
M_BALL_MOVE_X equ 64h
M_BALL_MOVE_Y equ 65h

RIGHT_F equ P1.1                ; flag: ball moving to the right
UP_F equ P1.2                   ; flag: ball moving up
SERVING_RIGHT_F equ P1.3        ; flag: serving ball to the right
GOAL_F equ P1.4                 ; flag: goal?

M_P1_Y_PREV equ 66h             ; previous bottom Y locations
```

```
M_P2_Y_PREV equ 67h
M_P1_Y equ 68h                  ; bottom Y location of paddles
M_P2_Y equ 69h

M_P1_SCORE equ 6ah
M_P2_SCORE equ 6bh

DPH2 equ 6ch
DPL2 equ 6dh

REFRESHED_F equ P1.0            ; flag: has current data been refreshed yet?

START_BUTTON equ P3.2
SOUND_HIGH_F equ P1.5           ; flag: next beep, high or low
GOAL_BEEP_F equ P1.6            ; flag: a goal beep?

P1_Y_VECTOR equ 7000h           ; RAM locations of vector data
P2_Y_VECTOR equ 7100h
BALL_X_VECTOR equ 7200h
BALL_X_VECTOR_HI equ 72h
BALL_Y_VECTOR equ 7300h
BALL_Y_VECTOR_HI equ 73h

P1_SCORE_X_VECTOR equ 7400h
P1_SCORE_X_VECTOR_HI equ 74h
P1_SCORE_Y_VECTOR equ 7500h
P1_SCORE_Y_VECTOR_HI equ 75h
P2_SCORE_X_VECTOR equ 7600h
P2_SCORE_X_VECTOR_HI equ 76h
P2_SCORE_Y_VECTOR equ 7700h
P2_SCORE_Y_VECTOR_HI equ 77h

;================================================================
;   CONTROL LOGIC
;================================================================

org 00h
ljmp main
org 0bh
ljmp refreshISR
org 1bh
ljmp beepISR

org 100h
main:
    mov TMOD, #11h              ; initializes serial port
                                ; set up timer 0 for 16-bit mode 1
    ;mov IE, #82h               ; enable timer 0 interrupt
    mov IE, #8ah
    mov TH0, #0CCh              ; set up 13166 counts (70Hz)
    mov TL0, #92h

    mov dptr, #SOUND_DAC
    mov a, #0ffh
    movx @dptr, a

    mov P1, #0h                 ; turn off all flags

    mov M_P1_SCORE, #0
    mov M_P2_SCORE, #0

    setb SERVING_RIGHT_F
    setb TR0
    lcall clearCursor

    mov M_BALL_X, #100          ; initialize the ball location
    mov M_BALL_Y, #80
    lcall loadScores
    lcall loadBall

_waitForStartButton:
    lcall updatePaddles
    lcall loadPaddles
```

```
    clr REFRESHED_F             ; keep refreshing until start pressed
_waitForRefresh1:
    jnb REFRESHED_F, _waitForRefresh1
    jnb START_BUTTON, _waitForStartButton

_start:
    mov M_BALL_X, #100          ; initialize the ball location
    mov M_BALL_Y, #80

    ;; check if anyone won yet
    mov a, M_P1_SCORE
    clr c
    subb a, #07
    jz _gameOver
    mov a, M_P2_SCORE
    clr c
    subb a, #07
    jz _gameOver

    clr GOAL_F

    lcall loadScores
    ;lcall loadPaddles
    ;lcall loadBall

    lcall serveBall
_controlLoop:
    mov r6, #04
_runLoop:
    lcall updatePaddles
    lcall collideWalls
    lcall collidePaddles
    lcall moveBall
    lcall detectGoal

    jb GOAL_F, _start           ; goal scored

    djnz r6, _runLoop

    lcall loadPaddles
    lcall loadBall

    clr REFRESHED_F             ; clear the refreshed flag
_waitForRefresh2:
    jnb REFRESHED_F, _waitForRefresh2

    ljmp _controlLoop

_gameOver:
    lcall loadScores
    lcall loadPaddles
    lcall loadBall
    clr REFRESHED_F             ; clear the refreshed flag
_waitForRefresh3:
    jnb REFRESHED_F, _waitForRefresh3
    sjmp _gameOver

;================================================================
; REFRESHISR: refreshes the scope screen by sending vector data
;               to the scopes
;================================================================
refreshISR:
    mov TH0, #0CCh              ; set up 13166 counts (70Hz)
    mov TL0, #92h
    jb REFRESHED_F, _skipISR    ; not finished calculating yet
    setb REFRESHED_F

    ;; pushing values not necessary because main has finished calculating

    lcall drawWalls
    lcall drawMidField
    lcall drawPaddles
    lcall drawBall
```

```
    lcall drawScores


    sjmp _endISR
_skipISR:
    setb P1.7                   ; set flag
_endISR:
    setb REFRESHED_F
    reti

;================================================================
; BEEPISR: switches for square waves
;================================================================
beepISR:
    jb GOAL_BEEP_F, _goal
    mov TH1, #0feh              ; 900Hz beep (1024 counts)
    mov TL1, #00h
    ;mov TH1, #0fch             ; 493Hz beep (934 counts)
    ;mov TL1, #64h
    sjmp _beep
_goal:
    ;mov TH1, #0f8h             ; 229.8 Hz beep
    ;mov TL1, #30h
    mov TH1, #0feh              ; 1.2kHz beep (384 counts)
    mov TL1, #080h
    ;mov TH1, #0ffh             ; 10 kHz beep
    ;mov TL1, #0156
_beep:
    push dph
    push dpl
    push acc
    mov dptr, #SOUND_DAC
    djnz r7, _continueBeep
    clr TR1
    mov a, #0ffh
    sjmp _sendSound
_continueBeep:
    jb SOUND_HIGH_F, _high

    clr a
    sjmp _sendSound
_high:
    mov a, #040h
_sendSound:
    movx @dptr, a
    cpl SOUND_HIGH_F
_endBeep:
    pop acc
    pop dpl
    pop dph
    reti

;================================================================
; BEEP: beep for collision
;================================================================
beep:
    mov r7, #50
    clr GOAL_BEEP_F
    setb TR1
    setb TF1
    ret

goalBeep:
    mov r7, #50
    setb GOAL_BEEP_F
    setb TR1
    setb TF1
    ret

;================================================================
; UPDATEPADDLES: updates the paddle Y state based on the ADC
;                   output values from the potentiometer controls
; destroys: dptr, a, b, r4
```

```
; outputs: M_P1_Y, M_P2_Y
;===============================================================
updatePaddles:
    mov a, M_P1_Y
    mov M_P1_Y_PREV, a
    mov a, M_P2_Y
    mov M_P2_Y_PREV, a

    mov dptr, #P1_ADC           ; get player 1 paddle position from ADC
    lcall adcToVal
    mov M_P1_Y, a

    mov dptr, #P2_ADC           ; get player 2 paddle position from ADC
    lcall adcToVal
    mov M_P2_Y, a
    ret

;===============================================================
; ADCTOVAL: reads the ADC at dptr and outputs the coordinate value
;       in Pong coordinates to acc
; input: dptr (ADC)
; destroys: dptr, a, b, r4
; outputs: acc (val)
;===============================================================
adcToVal:
    movx @dptr, a               ; fire up the adc
    mov r4, #08
_waitADC:
    djnz r4, _waitADC
    movx a, @dptr
    mov b, #SCALING_FACTOR      ; convert from [0-255] to [0-120]
    mul ab                      ; val = ADC_out*120/256 = b
    mov a, b
    ret

;===============================================================
; SERVEBALL: serves the ball from the winner,
;             i.e. gives ball initial velocity
; inputs: SERVING_RIGHT_F
; outputs: RIGHT_F, M_BALL_SPEED_X, M_BALL_SPEED_Y
;          M_BALL_MOVE_X, M_BALL_MOVE_Y
;===============================================================
serveBall:
    jb SERVING_RIGHT_F, _serveRight
    clr RIGHT_F
    sjmp _initBall
_serveRight:
    setb RIGHT_F
_initBall:
    mov M_BALL_SPEED_X, #DEFAULT_SPEED_X
    mov M_BALL_SPEED_Y, #DEFAULT_SPEED_Y
    mov M_BALL_MOVE_X, #0
    mov M_BALL_MOVE_Y, #0
    ret

;===============================================================
; COLLIDEWALLS: detects collisions with top and bottom walls
; destroys: r4
; output: UP_F
;===============================================================
collideWalls:
    jb UP_F, _collideUp
    mov a, #BALL_RADIUS         ; ball's moving down
    clr c                       ; if ball is less than radius away
                                ;   from bottom, then collide
    subb a, M_BALL_Y            ; diff = radius - ball_y
    jnc _flipUpDown             ; if radius >= ball_y, collide
    sjmp _endCollideWalls
_collideUp:
    mov r4, #Y_MAX

    mov a, M_BALL_Y
    add a, #BALL_RADIUS
```

```
    clr c
    subb a, r4                  ; acc = ball_y + radius - y_max
    jnc _flipUpDown             ; if acc !< 0, collide
    sjmp _endCollideWalls

_flipUpDown:
    cpl UP_F
    lcall beep
_endCollideWalls
    ret

;================================================================
; COLLIDEPADDLES: detects collisions with paddles
;   To collide, a ball must touch the paddle and the center of
;     ball must be along the paddle
; destroys: r4
;================================================================
collidePaddles:
    jb RIGHT_F, _collideRight
    mov a, #P1_X
    add a, #BALL_RADIUS
    subb a, M_BALL_X            ; P1_X + ball_radius == ball_x
    jnz _endCollide

    mov a, M_BALL_Y
    clr c
    subb a, M_P1_Y              ; ball_y >? P1_y
    jc _endCollide

    mov a, M_P1_Y
    add a, #P_LENGTH
    subb a, M_BALL_Y            ; P1_Y + P_length >? ball_y
    jc _endCollide

    sjmp _collide
_collideRight:
    mov a, #P2_X
    clr c
    subb a, #BALL_RADIUS
    subb a, M_BALL_X            ; ball_x + radius == P2_X
    jnz _endCollide

    mov a, M_BALL_Y
    clr c
    subb a, M_P2_Y              ; ball_y >? P2_y
    jc _endCollide

    mov a, M_P2_Y
    add a, #P_LENGTH
    subb a, M_BALL_Y            ; P2_Y + P_length >? ball_y
    jc _endCollide

_collide:
    cpl RIGHT_F
    lcall beep
    mov a, M_BALL_SPEED_X
    cjne a, #250, _increaseSpeed
    sjmp _endCollide

_increaseSpeed:
    mov a, M_BALL_SPEED_X
    add a, #10
    mov M_BALL_SPEED_X, a

    mov a, M_BALL_SPEED_Y
    add a, #10
    mov M_BALL_SPEED_Y, a

_endCollide:
    ret

;================================================================
; DETECTGOAL: detects if a goal has been scored
```

```
;              updates the scoreboard
; destroys: r4
;=================================================================
detectGoal:
    jb RIGHT_F, _detectGoalRight
    mov a, #BALL_RADIUS         ; ball moving left
    clr c
    subb a, M_BALL_X            ; radius - ball_x <= 0 ?
    jnc _goalP2
_detectGoalRight:
    mov r4, #X_MAX

    mov a, M_BALL_X
    add a, #BALL_RADIUS
    clr c
    subb a, r4                  ; acc = ball_x + radius - x_max
    jnc _goalP1                 ; acc <= 0?
    sjmp _endDetectGoal

_goalP1:
    inc M_P1_SCORE
    clr SERVING_RIGHT_F
    sjmp _scored
_goalP2:
    inc M_P2_SCORE
    setb SERVING_RIGHT_F
_scored:
    setb GOAL_F
    lcall goalBeep
_endDetectGoal:
    ret

;=================================================================
; MOVEBALL: moves the ball assuming no collisions
; destroys: a, c
; outputs: M_BALL_MOVE_X, M_BALL_MOVE_Y, M_BALL_X, M_BALL_Y
;=================================================================
moveBall:
    mov a, M_BALL_MOVE_X        ; load the ball move counter
    clr c
    add a, M_BALL_SPEED_X       ; update x counter with speed
    mov M_BALL_MOVE_X, a
    jnc _moveBallY              ; if carry, update ball x-loc, else move Y
    jb RIGHT_F, _moveBallRight
    dec M_BALL_X                ; move ball left
    sjmp _moveBallY
_moveBallRight:
    inc M_BALL_X                ; move ball left
_moveBallY:
    mov a, M_BALL_MOVE_Y
    clr c
    add a, M_BALL_SPEED_Y       ; update y counter with speed
    mov M_BALL_MOVE_Y, a
    jnc _endMoveBall            ; if carry, update ball y-loc, else end
    jb UP_F, _moveBallUp
    dec M_BALL_Y                ; move ball down
    sjmp _endMoveBall
_moveBallUp:
    inc M_BALL_Y                ; move ball up
_endMoveBall:
    ret

;=================================================================
; LOADPADDLES: loads the vector data for the 2 paddles into RAM
; destroys: a, dptr, r4
;=================================================================
loadPaddles:
    mov dptr, #P1_Y_VECTOR      ; set data pointer to beginning of P1 vector
    mov a, M_P1_Y               ; set bottom y-value for P1_Y
    lcall loadPaddle

    mov dptr, #P2_Y_VECTOR
    mov a, M_P2_Y
```

```
    lcall loadPaddle
    ret

;================================================================
; LOADPADDLE: loads the vector data for the a paddles into RAM
; input: dptr (paddle vector), acc (bottom y-value)
; destroys: a, dptr, r4
;================================================================
loadPaddle:
    mov r4, #P_LENGTH           ; number of points to write
_loadPaddleLoop:
    movx @dptr, a
    inc dptr
    add a, #01                  ; increment at resolution of 1
    djnz r4, _loadPaddleLoop
    ret

;================================================================
; LOADSCORES: loads the player scores
; destroys: P2, dptr, r0, r4, r5, acc
;================================================================
loadScores:
    mov dptr, #scoreTableX
    mov r4, M_P1_SCORE
    lcall setScorePointer       ; set dptr to P1's score x
    mov P2, #P1_SCORE_X_VECTOR_HI
    mov r5, #P1_SCORE_X
    lcall loadScoreVector       ; load P1's X scores

    mov dptr, #scoreTableY
    lcall setScorePointer       ; set dptr to P1's score y
    mov P2, #P1_SCORE_Y_VECTOR_HI
    mov r5, #P1_SCORE_Y
    lcall loadScoreVector       ; load P2's Y scores

    mov dptr, #scoreTableX
    mov r4, M_P2_SCORE
    lcall setScorePointer       ; set dptr to P2's score x
    mov P2, #P2_SCORE_X_VECTOR_HI
    mov r5, #P2_SCORE_X
    lcall loadScoreVector       ; load P2's X scores

    mov dptr, #scoreTableY
    lcall setScorePointer       ; set dptr to P2's score y
    mov P2, #P2_SCORE_Y_VECTOR_HI
    mov r5, #P2_SCORE_Y
    lcall loadScoreVector       ; load P2's Y scores
    ret

;================================================================
; LOADSCOREVECTOR: loads a score vector
; inputs: P2 (output vector high byte)
;         dptr (input vector)
;         r5 (output value offset)
; destroys: r0, acc
;================================================================
loadScoreVector:
    mov r0, #0h
_loadScoreLoop:
    mov a, r0
    movc a, @a+dptr
    cjne a, #0ffh, _loadScoreOK
    sjmp _endLoadScoreVector
_loadScoreOK:
    add a, r5
    movx @r0, a
    inc r0
    sjmp _loadScoreLoop
_endLoadScoreVector:
    mov a, #0ffh
    movx @r0, a
    ret
```

```
;================================================================
; SETSCOREPOINTER: sets the dptr to data for score in r4 (0-7)
; inputs: r4 (score 0-7), dptr (pointer to scoretable x/y)
; output: dptr (pointer to score data x/y)
;================================================================
setScorePointer:
    mov   a, r4               ; load acc with score
    rl    a                   ; multiply by two.
    inc   a                   ; load first vector onto stack
    movc  a, @a+dptr          ;         "          "
    push  acc                 ;         "          "
    mov   a, r4               ; load acc with monitor routine number
    rl    a                   ; multiply by two
    movc  a, @a+dptr          ; load second vector onto stack
    push  acc
    pop dph
    pop dpl
    ret


;================================================================
; LOADBALL: loads the vector data for the ball into RAM
; destroys: a, dptr, P2, r0, r4, r5, b, c
;================================================================
loadBall:
    mov r0, #0h               ; initialize input offset r0 to 0
    mov dptr, #ballX3         ; set input pointer to ballX table
    mov P2, #BALL_X_VECTOR_HI ; set output pointer to vector

_loadBallXLoop:
    mov a, r0                 ; load the r0-th point from table
    movc a, @a+dptr
    cjne a, #0ffh, _loadBallXOK ; terminating value found?
    sjmp _endLoadBallX
_loadBallXOK:
    add a, M_BALL_X           ; offset the ball's location
    clr c                     ; x-value = val + offset - radius
    subb a, #BALL_RADIUS

    movx @r0, a
    inc r0
    sjmp _loadBallXLoop       ; keep loading
_endLoadBallX:
    mov a, #0ffh              ; copy terminating character over
    movx @r0, a

    mov r0, #0h               ; initialize input offset r0 to 0
    mov dptr, #ballY3         ; set input pointer to ballY table
    mov P2, #BALL_Y_VECTOR_HI ; set output pointer to vector
_loadBallYLoop:
    mov a, r0                 ; load the r0-th point from table
    movc a, @a+dptr
    cjne a, #0ffh, _loadBallYOK ; terminating value found?
    sjmp _endLoadBallY
_loadBallYOK:
    add a, M_BALL_Y           ; offset the ball's location
    clr c                     ; x-value = val + offset - radius
    subb a, #BALL_RADIUS

    movx @r0, a
    inc r0
    sjmp _loadBallYLoop       ; keep loading
_endLoadBallY:
    mov a, #0ffh              ; copy terminating character over
    movx @r0, a
    ret

;================================================================
; DRAWPADDLES: draws the 2 paddles to the scope
; destroys: r2, dptr, a
;================================================================
drawPaddles:
    mov dptr, #X_DAC          ; send the x-value of paddle 1 to X_DAC
```

```
    mov a, #P1_X
    movx @dptr, a

    mov r2, #P_LENGTH           ; set up number of points for drawYLine
    mov dptr, #P1_Y_VECTOR
    lcall drawYLine
    lcall clearCursor           ; hide the cursor

    mov dptr, #X_DAC            ; send the x-value of paddle 1 to X_DAC
    mov a, #P1_X
    clr c
    subb a, #01
    movx @dptr, a

    mov r2, #P_LENGTH           ; set up number of points for drawYLine
    mov dptr, #P1_Y_VECTOR
    lcall drawYLine
    lcall clearCursor           ; hide the cursor


    mov dptr, #X_DAC            ; send the x-value of paddle 2 to X_DAC
    mov a, #P2_X
    movx @dptr, a

    mov r2, #P_LENGTH           ; set up number of points for drawYLine
    mov dptr, #P2_Y_VECTOR
    lcall drawYLine
    lcall clearCursor

    mov dptr, #X_DAC            ; send the x-value of paddle 1 to X_DAC
    mov a, #P2_X
    clr c
    subb a, #01
    movx @dptr, a

    mov r2, #P_LENGTH           ; set up number of points for drawYLine
    mov dptr, #P2_Y_VECTOR
    lcall drawYLine
    lcall clearCursor           ; hide the cursor
    ret

;================================================================
; DRAWYLINE: sequentially sends n (r2) values to the Y_DAC from the
;              memory location dptr.
; inputs: r2, dptr
; destroys: P2, r0, r2, dptr, a
;================================================================
drawYLine:
    mov P2, #0FEh               ; point P2:r0 to Y_DAC
    mov r0, #Y_DAC_LO
_sendYValue:
    movx a, @dptr               ; read the y-value from RAM
    movx @r0, a                 ; send y-value to DAC
    inc dptr
    djnz r2, _sendYValue
    ret

;================================================================
; DRAWWALLS: draws the 2 walls to the scope
; destroys: r2, dptr, a
;================================================================
drawWalls:
    mov dptr, #Y_DAC            ; send the y-value of wall 1 to X_DAC
    mov a, #Y_MIN
    movx @dptr, a

    mov dptr, #X_DAC
    mov r2, #X_MAX
_drawWall1Loop:
    mov a, r2
    movx @dptr, a
    djnz r2, _drawWall1Loop
```

```
    lcall clearCursor

    mov dptr, #Y_DAC            ; send the y-value of wall 1 to X_DAC
    mov a, #Y_MAX
    movx @dptr, a

    mov dptr, #X_DAC
    mov r2, #X_MAX
_drawWall2Loop:
    mov a, r2
    movx @dptr, a
    djnz r2, _drawWall2Loop
    lcall clearCursor
    ret

;================================================================
; DRAWMIDFIELD: draws the mid-field line to the scope
; destroys: r2, dptr, a
;================================================================
drawMidField:
    mov a, #X_MAX
    rr a                        ; acc = x_max/2
    mov dptr, #X_DAC
    movx @dptr, a               ; send x-value of mid-field to X_DAC

    mov dptr, #Y_DAC
    mov r2, #Y_MAX
_drawMidFieldLoop:
    mov a, r2
    movx @dptr, a
    djnz r2, _drawMidFieldLoop

    lcall clearCursor
    ret


;================================================================
; DRAWBALL: draws the ball to the scope
; destroys: P2, r0, r1, r3, dptr, a, dph2, dpl2
;================================================================
drawBall:
    mov dptr, #BALL_X_VECTOR
    mov DPH2, #BALL_Y_VECTOR_HI
    mov DPL2, #00
    lcall drawXY
    ret

;================================================================
; DRAWSCORES: draws the score to the scope
; destroys: P2, r0, r1, r3, dptr, a, dph2, dpl2
;================================================================
drawScores:
    mov dptr, #P1_SCORE_X_VECTOR
    mov DPH2, #P1_SCORE_Y_VECTOR_HI
    mov DPL2, #00
    lcall drawXY
    lcall clearCursor

    mov dptr, #P2_SCORE_X_VECTOR
    mov DPH2, #P2_SCORE_Y_VECTOR_HI
    mov DPL2, #00
    lcall drawXY
    lcall clearCursor
    ret

;================================================================
; DRAWXY: draws the values at dptr and dph2:dpl2
; destroys: P2, r0, r1, r3, a
;================================================================
drawXY:
    mov P2, #0FEh
    mov r0, #X_DAC_LO           ; set up pointers to X_DAC and Y_DAC
    mov r1, #Y_DAC_LO
```

```
    mov r3, #0                  ; initialize offset to 0
_drawXYLoop:
    mov a, r3                   ; set acc to offset
    movc a, @a+dptr             ; get the next x-value
    cjne a, #0ffh, _drawXYOK    ; terminating character?
    sjmp _endDrawXY
_drawXYOK:
    movx @r0, a                 ; send the x-value to the X_DAC

    mov a, r3                   ; set acc to offset
    push dph
    push dpl
    mov dph, DPH2
    mov dpl, DPL2
    movc a, @a+dptr             ; get the next y-value
    pop dpl
    pop dph
    movx @r1, a                 ; send the y-value to the Y_DAC
    inc r3
    sjmp _drawXYLoop

_endDrawXY:
    lcall clearCursor
    ret

;================================================================
; CLEARCURSOR: sends cursor offscreen
; destroys: a, dptr
;================================================================
clearCursor:
    mov a, #0ffh
    mov dptr, #X_DAC
    movx @dptr, a
    mov dptr, #Y_DAC
    movx @dptr, a
    ret

scoreTableX:
    dw X0
    dw X1
    dw X2
    dw X3
    dw X4
    dw X5
    dw X6
    dw X7

scoreTableY:
    dw Y0
    dw Y1
    dw Y2
    dw Y3
    dw Y4
    dw Y5
    dw Y6
    dw Y7

ballX2:
db 00h, 01h, 02h, 03h, 04h, 05h
db 05h, 04h, 03h, 02h, 01h, 00h, 0ffh

ballY2:
db 03h, 04h, 05h, 05h, 04h, 03h
db 02h, 01h, 00h, 00h, 01h, 02h, 0ffh

ballX3:
db 00h, 00h, 01h, 02h, 03h, 04h, 05h, 06h, 06h
db 06h, 05h, 04h, 03h, 02h, 01h, 00h, 0ffh

ballY3:
db 03h, 04h, 05h, 06h, 06h, 06h, 05h, 04h, 03h
db 02h, 01h, 00h, 00h, 00h, 01h, 02h, 0ffh
```

```
ballX3Filled:
db 00h, 00h, 01h, 02h, 03h, 04h, 05h, 06h, 06h
db 05h, 05h, 04h, 03h, 02h, 01h, 01h
db 02h, 02h, 03h, 04h, 04h
db 00h, 01h, 02h, 03h, 04h, 05h, 06h
db 05h, 04h, 03h, 02h, 01h
db 02h, 03h, 04h
db 0ffh

ballY3Filled:
db 03h, 04h, 05h, 06h, 06h, 06h, 05h, 04h, 03h
db 03h, 04h, 05h, 05h, 05h, 04h, 03h
db 03h, 04h, 04h, 04h, 03h
db 02h, 01h, 00h, 00h, 00h, 01h, 02h
db 02h, 01h, 01h, 01h, 2h
db 02h, 02h, 02h
db 0ffh


X0:
db 00h, 00h, 00h, 00h, 00h, 00h, 00h, 00h, 00h, 00h, 00h
db 01h, 02h, 03h, 04h, 05h
db 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h
db 05h, 04h, 03h, 02h, 01h
db 0ffh

Y0:
db 00h, 01h, 02h, 03h, 04h, 05h, 06h, 07h, 08h, 09h, 0ah
db 0ah, 0ah, 0ah, 0ah, 0ah
db 0ah, 09h, 08h, 07h, 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 00h, 00h, 00h, 00h, 00h
db 0ffh

X1:
db 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h
db 0ffh

Y1:
db 0ah, 09h, 08h, 07h, 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 0ffh

X2:
db 00h, 01h, 02h, 03h, 04h, 05h, 06h
db 06h, 06h, 06h, 06h
db 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 00h, 00h, 00h, 00h
db 00h, 01h, 02h, 03h, 04h, 05h, 06h
db 0ffh

Y2:
db 0ah, 0ah, 0ah, 0ah, 0ah, 0ah, 0ah
db 09h, 08h, 07h, 06h
db 05h, 05h, 05h, 05h, 05h, 05h, 05h
db 04h, 03h, 02h, 01h
db 00h, 00h, 00h, 00h, 00h, 00h, 00h
db 0ffh

X3:
db 00h, 01h, 02h, 03h, 04h, 05h, 06h
db 06h, 06h, 06h, 06h
db 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 06h, 06h, 06h, 06h
db 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 0ffh

Y3:
db 0ah, 0ah, 0ah, 0ah, 0ah, 0ah, 0ah
db 09h, 08h, 07h, 06h
db 05h, 05h, 05h, 05h, 05h, 05h, 05h
db 04h, 03h, 02h, 01h
db 00h, 00h, 00h, 00h, 00h, 00h, 00h
db 0ffh
```

```
X4:
db 00h, 00h, 00h, 00h, 00h, 00h
db 01h, 02h, 03h, 04h, 05h, 06h
db 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h
db 0ffh

Y4:
db 0ah, 09h, 08h, 07h, 06h, 05h
db 05h, 05h, 05h, 05h, 05h
db 0ah, 09h, 08h, 07h, 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 0ffh

X5:
db 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 00h, 00h, 00h, 00h
db 00h, 01h, 02h, 03h, 04h, 05h, 06h
db 06h, 06h, 06h, 06h
db 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 0ffh

Y5:
db 0ah, 0ah, 0ah, 0ah, 0ah, 0ah, 0ah
db 09h, 08h, 07h, 06h
db 05h, 05h, 05h, 05h, 05h, 05h, 05h
db 04h, 03h, 02h, 01h
db 00h, 00h, 00h, 00h, 00h, 00h, 00h
db 0ffh

X6:
db 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 00h, 00h, 00h, 00h
db 00h, 01h, 02h, 03h, 04h, 05h, 06h
db 06h, 06h, 06h, 06h
db 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 00h, 00h, 00h, 00h
db 0ffh

Y6:
db 0ah, 0ah, 0ah, 0ah, 0ah, 0ah, 0ah
db 09h, 08h, 07h, 06h
db 05h, 05h, 05h, 05h, 05h, 05h, 05h
db 04h, 03h, 02h, 01h
db 00h, 00h, 00h, 00h, 00h, 00h, 00h
db 01h, 02h, 03h, 04h
db 0ffh

X7:
db 00h, 01h, 02h, 03h, 04h, 05h
db 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h, 06h
db 0ffh

Y7:
db 0ah, 0ah, 0ah, 0ah, 0ah, 0ah
db 0ah, 09h, 08h, 07h, 06h, 05h, 04h, 03h, 02h, 01h, 00h
db 0ffh
```