4COM1042 [Computing Platforms]

Co-design Group Project B

“The Game of TV-Tennis”

Notes

Alex Shafarenko & Steve Hunt, 2016

Overview

For this project you will implement a TV-Tennis game.

- You will implement a 32x32 video display, and a kinematic controller chip that will move the bats and the ball around the display, and you will use a set of 7-segment displays to show the current scores for the two players.
- You will also write a program that predicts where the ball will travel and positions the computer's bat to hit it back.

TV-tennis is one of the names of the first video game ever produced (the Game of Pong). Watch the following video of the inventor Ralph Baering playing

`https://youtu.be/XNRx5hc4gYc`

and some more history of the game from the Smithsonian

`https://youtu.be/-I73oK9q-jk`

In this project we are trying to reproduce some features of TV-Tennis as well as getting the computer to play it with a human in as engaging a manner as possible.

We are of course armed by our superior knowledge of computing compared to the 1960s prototype but also disadvantaged by the rather crude simulation technology of Logisim, which does not permit us to use multiple bat controllers at the same time and limits the video resolution to a 32x32.

Nevertheless it is perfectly possible to achieve realistic gameplay; what's more, since the CPU simulation is equally basic (and slow), it is – unusually – rather easy to build a system in which the computer and the human player are evenly matched. This is further helped by the unwieldiness of the simulated joystick: it is not a straightforward affair to bring the bat to the desired location on the board, especially if the joystick has been driven beyond the sensor limit.

The robotic player

The new element in our game compared to the classic version is the robotic player: a computer that controls one of the bats.

The computer always plays the right bat, with the left bat being controlled by a human via a simulated joystick. To level the game a little we have constrained the robotic player:

- The computer can only move its bat while the ball is in the left-hand half of the screen (the human player's half). As soon as the ball has entered the right-hand half, the game controller (a piece of hardware) blocks the right bat.

  So the computer must make its predictions and move the right bat before the ball is in its half of the screen. Since the simulation is at a logic gate level, the computer is slow. Sometimes it will not be quick enough, and it will miss the ball.

Possible extensions

Even though we have nobbled the computer, it is still better equipped to make predictions than a human player, and this could result in dull play. To make it interesting, two advanced features have been introduced in our *reference implementation*, which you might consider adding as extensions to the basic system once you have got it working:

1. To make it more interesting, the controller applies a *diffusion force* vertically, i.e. randomly knocks the ball off target a little up or down. So the trajectory cannot be predicted by plotting a single straight line but by identifying a narrow range of different directions in which the ball might travel. This means the program must work by dead reckoning.

   Sometimes, when the calculation does not correctly predict the direction in which the ball will bounce, the computer may need to change the bat position more than once during the flight of the ball.

   In a situation where the angle of incidence is rather steep, the program needs to do more work to account for the predicted reflections, and the human can win a point much more easily.

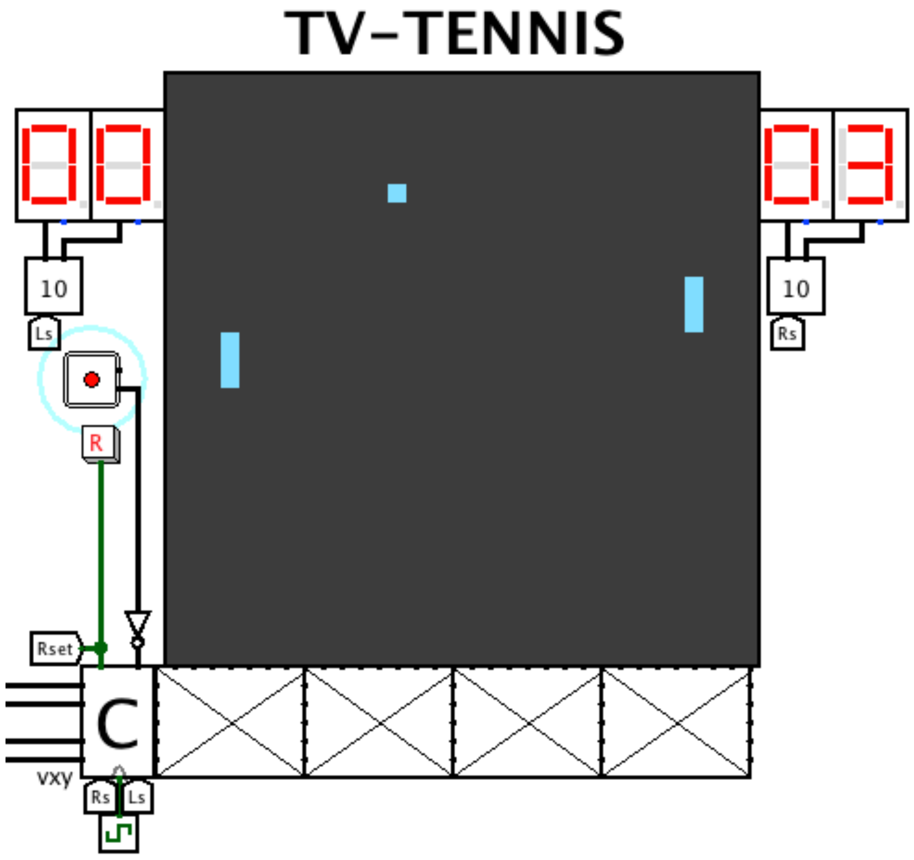
2. Even further, the reference system includes a velocity transfer circuit, which carries some of the momentum of the bat over to the ball, making the ball "spin" off the bat. This changes the angle of reflection, and makes it possible, although quite difficult, for a human to attack the robot with steep spinning shots.

Naturally, there is no expectation that you will develop either of these advanced features, but it is useful to known that they can be added within the constraints of the given simulation technology and processor core.

You are not limited in how you extend the system once you have got it working. You may change its functionality, or add other advanced features of your own devising.

Hardware design

This project is principally concerned with the hardware side. The game pad has several components, connected up in a pretty straightforward manner. One of these components – the kinematic controller, labelled 'C' in the diagram – is quite complex. We will leave consideration of that component to last, and consider the others first.



The display panel, the bats and the ball

We have a general-purpose *display panel* made up of 1024 pixels, arranged into 32 columns of 32 pixels each. The columns are numbered 0..31 from left to right, and the pixels in each column are numbered 0..31 from bottom to top.

Each column has a 32-bit input pin, with one bit per pixel. The pattern displayed by a column is dictated by the 32-bit pattern that is asserted on this pin (the *highs* in the pattern switch pixels on and the *lows* switch pixels off). Given the right input signals this panel could display any 32x32 pixel pattern.

For TV-tennis we need to display 2 bats and a ball against a plain background. The ball is drawn as a single pixel, and each bat is drawn as a vertical line made up of 3 adjacent pixels in a single column. The ball can appear anywhere on the screen, but the bats can only appear in fixed columns (left bat in column 3 and right bat in column 28).

To display the ball we need to switch on 1 pixel out of 32 in a specific column on the screen. We need to know the column (x coordinate) and row (y coordinate) in which the ball should appear. To display one bat we switch on 3 adjacent pixels out of 32 in a fixed column, so we just need to specify the row (y coordinate) in which the bottom end of the bat should appear. So we need four 5-bit numbers to specify the positions of the ball (ballX, ballY) and the two bats (leftbatY, rightbatY). These will come from the kinematic controller.

The video subsystem

The video subsystem has four 5-bit inputs: ballX, ballY, leftbatY, rightbatY. The display panel is driven by a set of 32 *video chips*, and all four of these inputs are connected to every video chip. However, in order to reduce the amount of wiring the video chips are *chained*. The chips are arranged in a chain from West to East; the West-most chip receives all of the input data, and passes it on to the next chip in the chain.

Instead of chaining 32 individual video chips we employ a *cellular design*. 8 chips are chained together to make a *video section*. 4 of these sections are then chained. Cellular design is a standard technique: design a single cell, add it to the library of circuits, then build a bigger circuit by wiring together a number of identical cells in a regular pattern. Each section is indicated in the circuit diagram by a rectangle containing a diagonal cross.

The video chip

Every video chip has ballX, ballY, leftbatY, rightbatY *input* pins on its West side and ballX, ballY, leftbatY, rightbatY *output* pins on its East side, and the bit patterns on each of them pass through each video chip unchanged.

Each video chip drives a specific column of the display panel, and each chip needs to 'know' which column it is driving, so the chips are numbered 0..31 like the columns. Rather than store a different column number in each chip individually we use a little trick.

We give each video chip an additional 5-bit input pin labelled chipID on its West side, and add a corresponding output pin on its East side, but this time we add 1 to the number before outputting it again. Then when we connect the chips in a chain we make sure to input 0b00000 on the chipID pin of the West-most chip (= column 0), which adds 1 and so sends the signal 0b00001 to the next chip in the chain. This way all 32 chips number themselves automatically.

West side inputs (5-bit input pins)	**East side outputs (5-bit output pins)**
chipID	chipID+1
ballX	ballX (unchanged)
ballY	ballY (unchanged)
leftbatY	leftbatY (unchanged)
rightbatY	rightbatY (unchanged)

It is important to understand that the video chip is purely combinational. A video chip has no internal state and produces a 32-bit output pattern that drives one column of the display panel. That pattern is output on the single North pin. So how is this pattern created?

Each chip includes circuitry to display a ball, and circuitry to display a single bat.

The ball's position is received on input pins ballX and ballY. Each chip compares its chipID to ballX, and if they are equal it lights up pixel number ballY in its column of the display. This is a straightforward piece of combinational logic.

Because the bats are in fixed columns we can 'hard-wire' their column numbers (3 and 28) into the video subsystem. Every video chip contains two 5-bit constants: 0b00011 and 0b11100. These two constants are compared with the chipID, so the video chip that has 00011 as its chipID displays the left bat with its bottom pixel in row leftbatY, and the video chip that has 11100 as its chipID displays the right bat with its bottom pixel in row rightbatY. This too is achieved by combinational logic.

NOTE: No chip will ever need to display both *bats*, but it is possible that the ball could be in the same column as one of the bats, so the design should allow for this.

Kinematic controller

The job of the kinematic controller is to move the ball by updating its coordinates, to block the right bat when the ball is on the right-hand side of the screen, and to report the positions of the ball and the two bats to other parts of the system (the program and the video subsystem).

The rest is done by other pieces of hardware and software: the left bat is controlled directly by a (simulated) joystick and the right bat is controlled (via an I/O interface) by the CdM-8 core running the robot player program.

The controller updates the coordinates and velocity of the ball, and reports them to the program, which uses these values to predict the flight path of the ball and to move its bat accordingly. The kinematic controller needs to know the position of both bats to determine whether the ball has hit either of them, and is also responsible for 'blocking' the right bat when the ball is in the right-hand half of the screen

Position of the ball

The ball moves around a square space that is divided up into cells by a 256 x 256 grid, with the cells numbered 0..255 from left to right and 0..255 from bottom to top, so the bottom left hand cell has the coordinates 0,0. The x and y coordinates of the ball are held as 8-bit unsigned numbers, and all movement calculations and position updates are performed on those 8-bit values.

Driving the display

As the controller has the position of the ball and the y coordinates of the two bats it is logical to use it to communicate them to the video subsystem for display purposes.

Unfortunately the display panel cannot show the ball's location with such great precision, because it is only 32 pixels across and 32 pixels high. We make each pixel on the display correspond to many grid cells in the space (64 grid cells to each pixel in fact). For example, the pixel at column 2 and row 5 on the screen will be used to display the ball when its actual x-coordinate is between 16 and 23, and its y coordinate is between 40 and 47.

The translation from controller coordinates to screen column and row numbers is easy to achieve by sending just the 5 most significant bits of the ball's x and y coordinates to the video subsystem. The bats' y coordinates are held as 5-bit values throughout, so no translation is required.

Representing ball velocity

The velocity, vxy, of the ball has a horizontal (x) component, vx, and a vertical (y) component, vy. Each of these is represented by a 3-bit 2's complement (signed) number, meaning that vx and vy can each take on values in the range -4..+3. This is a very coarse scale, but it is good enough for our purposes.

A positive x velocity means that the ball is travelling from left to right across the space, and a positive y velocity means that the ball is travelling from bottom to top. Negative velocities of course mean that the ball is travelling in the opposite direction. Any combination of vx and vy is possible.

Every data item read by the CdM-8 core must be represented as an 8-bit string, because CdM-8 is an 8-bit platform. Rather than pad each of the two velocity components to 8 bits we combine them into one 8-bit string as follows: bits 0..2 carry vx, bits 3..5 carry vy, and bits 6..7 are always set to 0.

Reflecting the ball

When the ball hits a bat or reaches one of the vertical edges of the space it is made to bounce (reflected) by negating vx. When it hits the top or bottom edge of the spaceit is reflected by negating vy. It is only when the ball exactly hits a corner that both vx and vy will need to be negated to make it bounce.

It should be noted that -4 cannot be negated correctly, because there is no 3-bit 2's complement representation for +4. In fact, the 3-bit 2's complement of –4 is also –4, so the ball's velocity in that direction will not change at all. In this circumstance the reflected value of the velocity is set to +3 instead.

Internal storage

The controller needs two pairs of registers to start with: a pair of 8-bit ones for the ball's x and y coordinates and a pair of 3-bit ones for the x and y components of its velocity.

It also includes two 8-bit registers to keep the two players' scores, and a 5-bit register into which is latched the right bat position. This latter register is used to implement blocking the computer from moving the bat when the ball is on the right hand half of the screen. A new right bat position can only be latched into the register when the ball is in a display column whose number does not exceed 15.

That is all the state that needs to be managed. The rest of the circuitry is purely combinational.

Motion calculation

The trajectory of the ball is computed iteratively, step-by-step, using a *local* clock. The clock in question has no relationship with the processor clock, even though Logisim simulation ties all the clocks used in a design together. In a way the existence of a local clock means that the controller is a tiny computer in its own right, although it does not have memory and computes exclusively using registers. Nor does it have a program as such, since it repeats the same hard-wired "instruction" at every clock cycle.

The calculation is a two-phase process:

Phase 1: step movement of the ball based on the current velocity
Phase 2: velocity alterations based on collisions detected in Phase 1

Phase 1 is simple to compute

x := x + vx

and

y := y + vy

The x and y coordinates are used to refer to cells in the ball's virtual movement space. The x and y components of velocity are measured in units of cells-moved per clock-cycle, represented as 8-bit strings, with the sign bit extended, as it should be, to the left. The new position of the ball can then be calculated using two 8-bit adders.

Detecting when the ball reaches an edge

It is very simple to detect when the ball has collided with one of the edges of the movement space: when vx<0, adding it to the x coordinate must produce a carry (of 1); if it does not, the ball has just hit the left wall. When vx>0 adding it to the x coordinate must ***not*** produce a carry; if it does, the ball has just hit the right wall. Similarly, when adding vy to y we may use the carry-out to determine whether the ball has collided with the ceiling or the floor.

In actual fact the kinematic controller does not detect a collision with an edge. What it *actually* detects is the fact that the ball would be *outside* the movement space if its position was updated by the required amount. This is obviously illegal, so when it is detected the new coordinate is *not* latched in the coordinate register. Instead, the ball's velocity in this dimension is negated to make it 'bounce'.

Notice that there are two kinds of collisions: horizontal and vertical. Only one coordinate is involved in a collision, the other does not participate. Of course if the ball is played directly into a corner, it may hit both walls at the same time; nonetheless each collision direction will be processed independently.

In Phase 2, the velocity update takes place. If there was no collision in either the horizontal (x) or the vertical (y) dimension the ball's velocity remains the same. If there was a collision, the velocity has to be negated in that dimension, reversing the direction of travel.

How do we implement the above two-phase process? A simple solution is to make use of both halves of the clock cycle. When the clock is high, the coordinates get recalculated based on the current velocity; when the clock is low, the speed gets reflected depending on whether the movement has resulted in a collision. This means that the coordinate latches must be triggered by the falling edge of the clock and the velocity latches by the rising edge.

Bouncing the ball off a bat

The bats can only affect horizontal reflections, since they are themselves vertical. The reflection procedure is almost exactly the same as that for reflecting the ball off a wall. In the latter case we assume that the walls are at x=0 and at x=255, and use the carry to detect when the ball is attempting to cross that boundary

When detecting contact with a bat we perform a slightly different calculation, but use the same detection technique of examining the carry out. The left bat is in column 3 on the display, which means it spans x coordinates in the range 24..31. If we assume that the ball will hit the middle of the bat that means it will have 28 as its x coordinate at the point of contact.

Now when we use negative vx to update the ball's position (i.e. it is travelling right to left) we first subtract 28 from the old x value, then add vx, and if there is a carry-out of 0 the ball may have hit the bat (but only if the ball has a y coordinate that places it within the length of the bat).

–28 is 0xe4 when represented as an 8-bit 2's complement number. The distance for the right bat is the same, 28 units, only now the ball is travelling in the opposite direction, so the distance is positive: 0x1c.

Scoring the game

A player's score in the game is the number of ball reflections off the vertical wall in the other player's half of the movement area. The controller includes two counters for keeping score. (In this instance each counter is a register with an increment in its feedback: we use a standard Logisim component for it).

The trigger for the score counters comes from the adder that implements x+vx in Phase 1, or rather from its wall-collision circuit. Every time a wall collision is detected (recall that this is done by detecting that the carry-out of the adder is opposite to the sign of vx), this acts as an enable signal for the score counter update in the following (low clock) Phase 2 cycle. Which player gets the point depends solely on the sign of vx at the point of collision.

Design summary

To summarise, the design for the kinematic controller consists of six uncomplicated parts of the circuit:

- X update
- Y update
- vx reflection
- vy reflection
- bat collision detection
- score keeping

The relationship between the above parts is as follows: X- and Y-updates drive vx- and vy-reflections, with vx-reflection being also driven by bat collision detection. Finally the score keeping is entirely under the control of X-update.

The specifications of the kinematic controller and the two subsidiary chips upon which it depends are in the appendix at the end of this document

Interfacing the game with the CdM-8 platform

The program running on the processor needs to know the coordinates and velocity of the ball in order to make predictions. When it has worked out the desirable position of the right bat this has to be made available to the kinematic controller.

Consequently the interface should include one I/O register in which the program will write the right bat's y-coordinate. This is a 5-bit register that should utilise the *most* significant bits of the **`I/Odat`** bunch.

Let us connect the game pad to IO-3 (address = 0xf3) for this game. The same address can be used to read the ball velocity vxy packed, as the controller does it, into an 8-bit word with the 2 MSBs unused. The notes titled *Working with a full-core CDM-8 system*, available separately, explain how to share IO addresses.

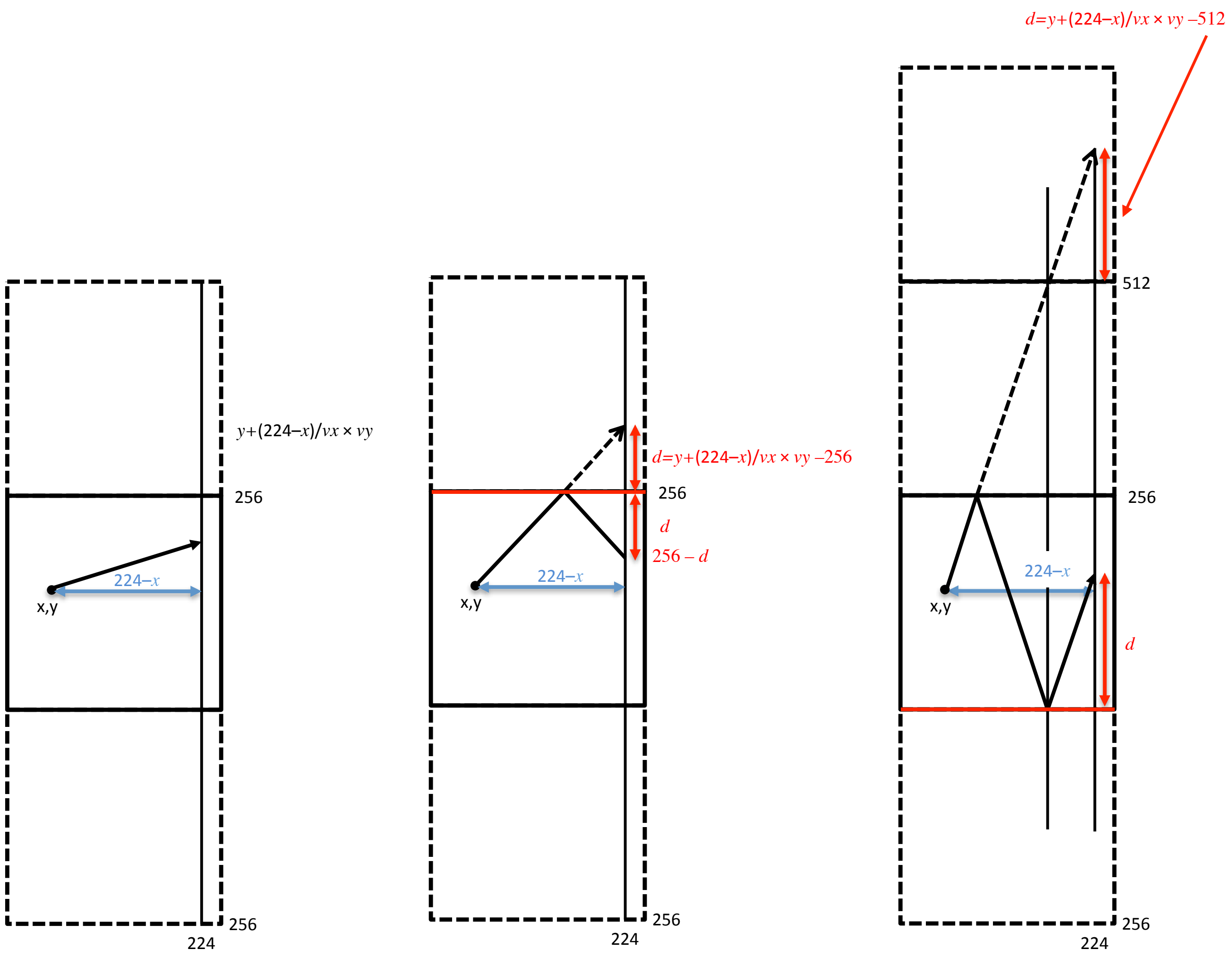
Two more registers should be employed to enable the program to read 8-bit x- and y-coordinates of the ball, say IO-4 and IO-5. Neither the velocity nor the coordinates require registers in the interface since the controller latches them at every clock cycle and asserts them on east side pins. All the interface needs to do is decode the I/O address/direction of transfer and assert those values on the **`I/Odat`** bunch.

Software

The program has to predict the ball's trajectory given its current position and velocity. It should determine the point at which the ball will cross the 28th vertical of the display (counting from 0) and write its y-coordinate to the IO-3 register. It must do so reasonably quickly because if in the meantime the ball travels as far as the median point of the display, the controller will not copy the content of IO-3 to its eatsern pin, and as a result the right bat will not move and the ball will likely hit the right wall.

Having to work fast means that trying to follow the hardware algorithm in software will not deliver satisfactory results since the processor is too slow compared to the controller. Instead of imitating the controller by adding small increments to the coordinates and working out the next location until we approach the bat, we must compute the final location of the ball at the point of crossing in *one* step. The diagram on the next page indicates how this may be done.

The distance between the ball and the right bat along the horizontal axis is $224 - x$ (here and below we will measure any distances in sub-pixels to scale the board up to the size 256x256). If the horizontal velocity of the ball is vx, it will



take the ball $(224 - x)/vx$ controller clock cycles to reach the bat line. Its vertical displacement over this time will be $(224 - x)/vx \times vy$. This displacement, when added to the current y-coordinate defines the intersection point on the bat line at which the ball will arrive. Or rather *would* arrive if we could guarantee that it would not collide with the horizontal wall.

In the current design no object other than the ball ever moves in the *x*-direction; this means that vx has a constant magnitude. The controller sets vx to –2 (110 in 3-bit 2's complement) upon reset, so the absolute value of the horizontal speed stays at 2. On the other hand, vy is a 3-bit number as well, so its absolute value could be at most 4.

This means that we can easily both divide and multiply in the program. We divide by 2 by executing the right shift, and we multiply by 2 and 4 by doing left shifts; finally multiplication by 3 boils down to a shift and an add.

In the process of "multiplying" and adding to get the displacement and the eventual crossing point, only the low-order 8 bits will be kept; we need to pay attention to the carry bit. Indeed as the diagrams above suggest, if the result *d* is obtained without a carry-out of 1 (left), or with a carry-out of 1 occurring an even number of times (right), *d* is the y-coordinate of the crossing point. However if the carry bit has been produced an odd number of times (e.g., it is produced once in the middle figure), *d* needs to be subtracted from 256. (But that is simply 8-bit 2's complement.)

In the above analysis we assumed that the vertical velocity vy was positive. Nothing much changes when vy<0. The difference is that instead of counting the number of times a carry of 1 has been produced in doing the calculation, we count the number of times a carry of 0 has occurred. Indeed, when adding a negative to a positive results in a positive (in terms of the above figure: when the horizontal line is not crossed) the negative will produce a carry of 1 (since it has a 1 in the sign bit already and it can only become 0 together with a carry out of 1).

At the very least the program should be able to deal with one reflection. This is easy enough (involving only a few arithmetic manipulations and tests) and is completely sufficient if the basic controller is not extended to include random velocity fluctuations and/or bat-to-ball velocity transfer. However, if those extensions are attempted, the incident angle may become rather steep, in which case it may be possible for a ball to bounce off the walls twice (as in the right diagram above).

Suggested design progression

1. Use the **cocoemu** emulator for debugging your code thoroughly before interfacing it with the game pad. You should use I/O addresses 0xf3 to 0xf5 as normal memory addresses and simulate the controller by placing constants in them. This way it is perfectly possible to completely debug the trajectory prediction program before the controller and other circuitry are built.

2. Start your hardware work in parallel with the software. Begin by building a single vertical row chip that renders one column of the 32x32 display. The chip should produce a 32-bit raster of zeros, or an image of a one-pixel ball at a certain height, or an image of a bat (3 consecutive pixels at some height) or both the ball and the bat. All will depend on the east side inputs. Use pins to debug your circuit thoroughly. Then stack 32 copies of the chip together and see if you can manipulate the ball and the bats using pins. You are ready to proceed to building the controller.

3. Follow the controller chip specification *very carefully*. Check everything many times. Focus on the X,Y, vx and vy latches and build a working prototype that can move the ball around correctly. Leave out the bat collision parts and the score registers. Attach pins to the controller and set some y-coordinates for both bats. By clicking on the clock see if you can induce the ball to fly and get reflected. Then connect a joystick to the left-bat input pin LEFTYIN, set a reasonably low tick frequency and give it a go playing against the walls and an immovable right bat.

4. If you have reached this point, your controller basically works. Quickly build the computer interface and get your software person to run the game. Develop your own little tests though, which show that the right bat works correctly under program control, so when the software person insinuates that it is the hardware that does not work you may have a convincing proof of how wrong they are.

GOOD LUCK!

Appendix: Chip specifications

The following two chips are used by the kinematic controller. Each of them is purely combinational. In other words each of them acts as if it were a mathematical or logical *function*

The first is used to reverse the direction of travel of the ball in either the horizontal or the vertical direction. The old and new velocities are always of opposite sign, but the chip does not perform a mathematically correct change of sign, since it assumes that –(–4) = 3, which leads to a slightly incorrect angle of reflection for the ball. This just makes the game more interesting.

```
Chip Specification
   Name: reflect
I/O pins
   Input:  oldv(3)
   Output: newv(3)
Combinational
   newv =
      if oldv = 0b100 then oldv’
      else oldv’+1
End Chip Specification
```

The second chip works out whether the ball is in one of the screen rows occupied by one of the bats. This is used to determine whether the bat and ball have come into contact with one another.

```
Chip Specification
   Name: inrange
I/O pins
   Input:  Yball(5), Ybat(5)
   Output: shareArow
Combinational
   shareArow =    Yball = Ybat
                ∨ Yball = (Ybat+1)
                ∨ Yball = (Ybat+2)
End Chip Specification
```

The kinematic controller is specified on the next two pages

```
Chip Specification
   Name: kinematic_controller
   Uses combinational chips: reflect(3->3), inrange(5,5->1)

I/O Pins
   Input:   reset                           # North side
            # The bats' vertical coordinates (from joystick & program)
   Input:   leftYin(5)                      # North side
   Input:   rightYin(5)                     # West side

            # ball coordinates and velocity (to CdM-8 program)
   Output:  xball(8), yball(8), vxy(8)      # West side

            # scoreA, scoreB used to update scoreboard
   Output:  scoreA(8), scoreB(8)            # South side

            # number of first display column
   Output:  zero(5)                         # East side

            # ball coordinates (to display)
   Output:  ballX(5), ballY(5)              # East side

            # The bats' vertical coordinates (to display)
   Output:  rightYout(5), leftYout(5)       # East side

Internal
   Signals:  vxneg, vyneg, xrefl, yrefl
   Signals:  hitA, hitB              # one of these will be true when
                                      # ball passes bat of other player
   Signals:  Lbcontact, Rbcontact, Bcontact
   Signals:  longVX(8), longVY(8)

   Latches:
             register(8) xreg,yreg # hold current x and y coordinates
                                   # of the ball in 256x256 space
             register(3) vx,vy     # x and y components of ball velocity

             resgister(8) scA,scB   # scores of the two players
             resgister(5) rightYreg # holds right bat y coordinate

Combinational
   zero = 0b00000
   leftYout = leftYin                 # Left bat position passed through
                                      # without change
   rightYout = rightYreg              # Right bat position taken from
                                      # register
   ballX = xreg.split(3:7)            # 5 MSbs of ball coordinates
   ballY = yreg.split(3:7)            # used to locate ball on screen

   vxy(0:2) = vx                      # Velocity has a 3-bit x component
   vxy(3:5) = vy                      # and a 3-bit y component
   vxy(6:7) = 0b00                    # padded to 8 bits with two 0s

   # Extend each 3-bit velocity to an 8-bit 2's complement number,
   # setting the 5 MSb's of the extended version to the sign of the
   # 3-bit version. Needed so we can add vx to x and vy to y
   longVX = vx.sign_extend(3->8,vx(2))
   longVY = vy.sign_extend(3->8,vy(2))
   vxneg =  longVX.split(7)                 # Is x velocity negative?
   xrefl =  (xreg + longVX).carry ≠ vxneg   # has the ball hit a wall?

   vyneg =  longVY.split(7)                 # Is y velocity negative?
   yrefl =  (yreg + longVY).carry ≠ vyneg   # has the ball hit the floor
                                            # or ceiling?
   hitA  =  xrefl ∧ vxneg                   # has player A scored?
   hitB  =  xrefl ∧ vxneg'                  # has player B scored?

   # Has ball hit right bat?
   LBcontact =  ( ((xreg + 0xe4) + longVX).carry ≠ vxneg )
               ∧ inrange (y, leftYin)

   # Has ball hit left bat?
   RBcontact =  ( ((xreg + 0x1c) + longVX).carry ≠ vxneg )
               ∧ inrange(y, rightYin)

   Bcontact = Lbcontact ∨ RBcontact

Behaviour
   on falling_edge of clock do:
                  xreg :=                   # update x-coord of ball
                     if reset  then 0xc0    #  x at start of game
                     else xreg + longVX     #  add x-velocity to x
                  enabled by:
                     (xrefl' ∧ Bcontact') ∨ reset

                  yreg :=                   # update y-coord of ball
                     if reset then 0x80     #  y at start of game
                     else yreg + longVY     #  add y-velocity to y
                  enabled by:
                     yrefl' ∨ reset

   # right bat can only move when ball is not on
   # r.h.s. of screen (column number < 16):
                  rightYreg :=
                     if reset then 0x0f else RightYin
                  enabled by:
                     ballX.split(4)'

   on rising_edge of clock do:
                  vx  :=                    # update x-velocity
                     if reset  then 0b110   #  initially -2
                     else reflect(vx)       #  horizontal bounce
                  enabled by:
                     xrefl ∨ Bcontact ∨ reset

                  vy  :=                    # update y-velocity
                     if reset  then 0b001   #  initially +1
                     else reflect(vy)       #  vertical bounce
                  enabled by:
                     yrefl ∨ reset

                  scA  := scA + 1 enabled by hitA
                  scB  := scB + 1 enabled by hitB

End Chip Specification
```