Total Immersion Pong

Chris Laas Jacob Strauss

6.111 Final Project

May 13, 1999

Abstract

This report documents the design and construction of a Full Immersion Pong system. Interfacing with the user entirely through video input and output streams from a camera and to a monitor, and a couple of LEDs, this device allows two players to “be” the paddles in the traditional computer game Pong. The device consists of a video I/O subsystem, a time-domain frame differencer, a game rules module, and a bitmap overlay module. The input/output subsystem feeds a stream of video data and synchronizing timing pulses on to the frame differencer, which detects motion in the field of view. This information is passed on to the rules module, which uses it to determine whether and when the “ball” will bounce. Finally, the current ball position is fed to the bitmap overlay module, which modifies the output stream to include an image of a rotating cube at that location.

Contents

List of Figures

1 Introduction

1.1 Pong

Pong was one of the first electronic games to achieve great popularity, largely because it is simple enough to implement cheaply, yet fast-paced enough to entertain. In the simplest form of the game, a ball moves on a monitor, bouncing off of the ceiling and floor and off of "paddles" — rectangles near the left and right edges of the screen — whose vertical positions are controlled by the players' joysticks. If a player allows the ball past his paddle, he loses a point.

This project implements a more entertaining version of this game: instead of using joysticks to control paddles, the players stand in front of a video camera and bounce the ball by moving their arms, legs, and body. The digital system detects the movement and reflects the ball when it touches a region of motion. The effect is one of "immersion" of the player into the game; instead of interacting with a computer's keyboard or joystick, the player interacts as directly as possible with the game.

1.2 Modules Overview

1.2.1 Overall Organization

The Full Immersion Pong game consists of four main modules. The analog I/O module converts a video stream to a sequence of digital samples for processing, and converts the processed digital signal into an analog signal suitable for display on a monitor. The frame differencer detects motion in the video stream, and passes this information on to the rules

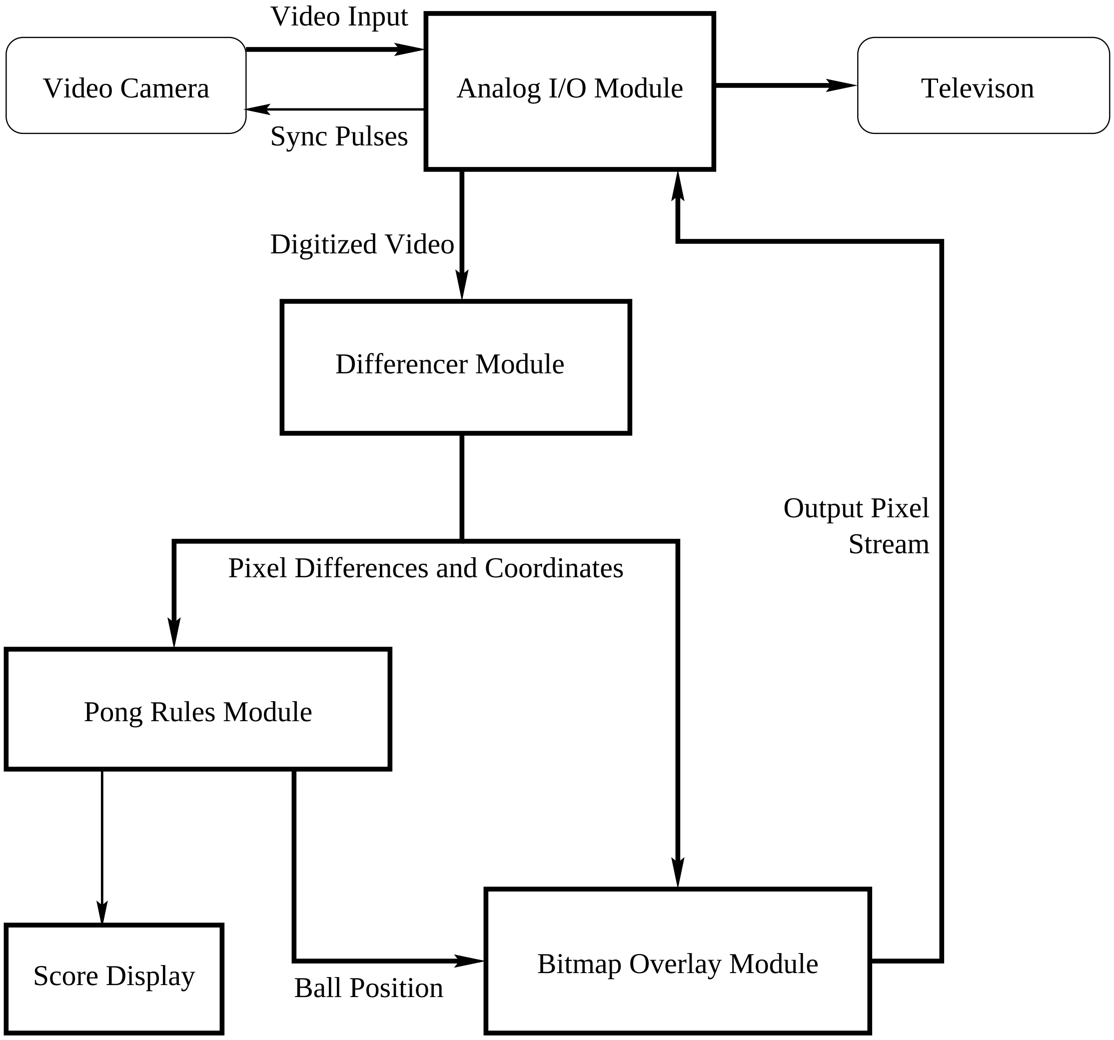


Figure 1: System Block Diagram

module, which keeps track of the state of the game. Finally, the bitmap overlay module superimposes an image on the difference stream to provide visual feedback on the current ball position to the user, and outputs this stream to the I/O module for display.

1.2.2 Analog I/O Module

The input to the system is an analog NTSC video stream from a greyscale camera. This camera is synchronized with regular horizontal sync and vertical sync pulses from an LM1882 video sync generator, which also provides synchronization control signals to the digital logic. The video data arrives at a rate of 30 frames per second, with two interlaced fields per frame. There are 525 lines per frame, giving a line rate of 15.73 kHz. Lines are separated by low-voltage "blanking" pulses and even lower-voltage "sync" pulses; fields are separated by similar pulses of longer duration. Between blanking pulses, the AD775 flash A2D samples the input stream at a rate of 4.77 MHz (210 ns per pixel), giving 180 pixels per line. The eight-bit pixel samples are sent to the digital logic, which processes this data and produces an output stream of pixels at the same rate. This data is fed to the inputs of a DAC0800 D2A, whose current output is processed by an op amp and combined with sync signals to produce an NTSC video signal appropriate for a greyscale display screen.

1.2.3 Frame Differencer Module

To locate regions of motion, the device performs a time-domain first difference on each pixel of the frame. In order to do this, it stores an entire frame in a 256 kilobyte DRAM by cycling through all the pixel row and column coordinates in the fast page mode of the DRAM. The previous value and the current value of the pixel brightness are given to a combinatorial circuit which calculates the absolute value of the difference. A block diagram of the Differencer and the input portion of the Analog I/O module is shown in Figure 2.

1.2.4 Pong Rules Module

This module, built from programmable logic, keeps track of the current position and velocity of the ball and of the current score. After every frame, the x and y coordinates of the ball are incremented by the velocity values, causing the ball to move across the screen. When the

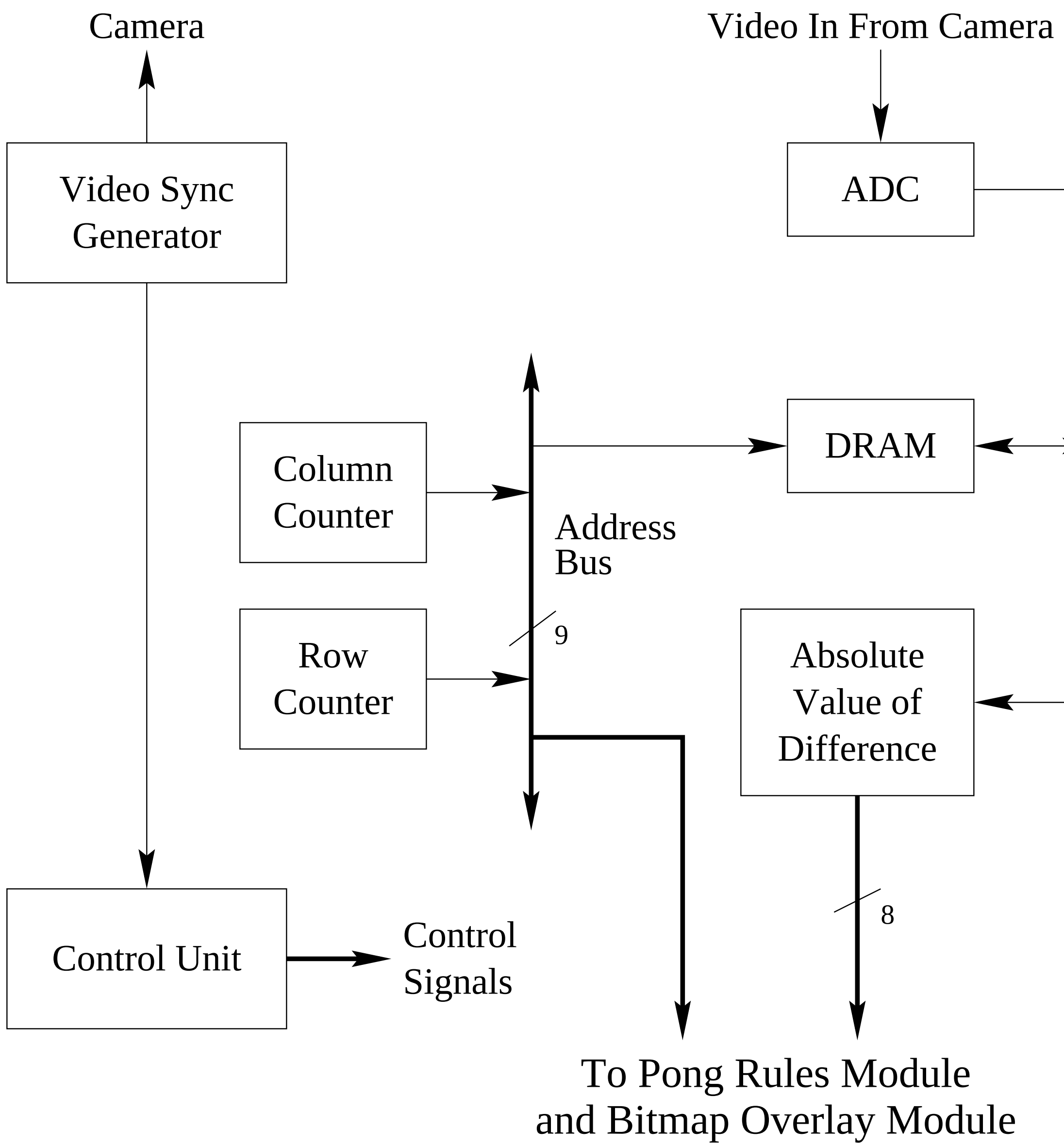


Figure 2: Differencer Block Diagram
All of the components of the differencer are shown. The input portions of the Analog I/O module, the ADC and the Video Sync generator, are also shown.

ball position and current pixel coordinates are equal, a bit of the difference stream is tested, to determine if there is motion in the video-in stream at the location of the ball. If so, the ball then bounces back and randomizes its velocity. If the ball falls off either horizontal edge of the screen, the appropriate player loses a point.

1.2.5 Bitmap Overlay Module

Finally, the ball's coordinates, the current pixel coordinates, and the eight-bit difference stream are sent to the bitmap overlay module. The pixel's coordinates are subtracted from the ball's by discrete logic, and the result is fed into the address lines of a PROM lookup table and to the selector lines of a multiplexor, to produce a single bit of the bitmap image programmed into the PROM. This bit is XORed with every bit of the difference stream, and the result goes to the D2A and, finally, to the display. The two high bits of the PROM address lines are attached to a '393 which cycles roughly twice a second; they select one of four images in an animation. The effect is that the display shows regions of brightness where the camera input is changing, and that there is an image of a rotating cube superimposed on the screen at the position of the ball.

2 Description

2.1 Input and Ouput

2.1.1 Analog to Digital Converter

The video signal from the camera is fed directly into an AD 775 analog to digital converter, U1 on Figure 3. This chip samples video data on the falling edge of the clock, and two and a half clock cycles later places the value corresponding to that sample on the outputs, which connect to the differencer. The output enable for the chip is asserted for half of a pixel cycle, during which the dram stores the intensity of the pixel. The pipeline delay associated with the samples is ignored by the system. The input range of the ADC as configured here is 0 to 2.4 volts.

2.1.2 Sync Generator

The LM1882, U4 on Figure 3, provides vertical and horizontal sync pulses for both the camera and the rest of the controller, providing information about the start and end of both lines and frames. The data input signals are present to turn on the output enable bit in the chip's status register. The chip's default values create pules for standard NTSC when clocked at 14.31818 MHz, except for the output enable. To start up, the `clr` and `/load` inputs must be pulsed sequentially to enable the ouput.

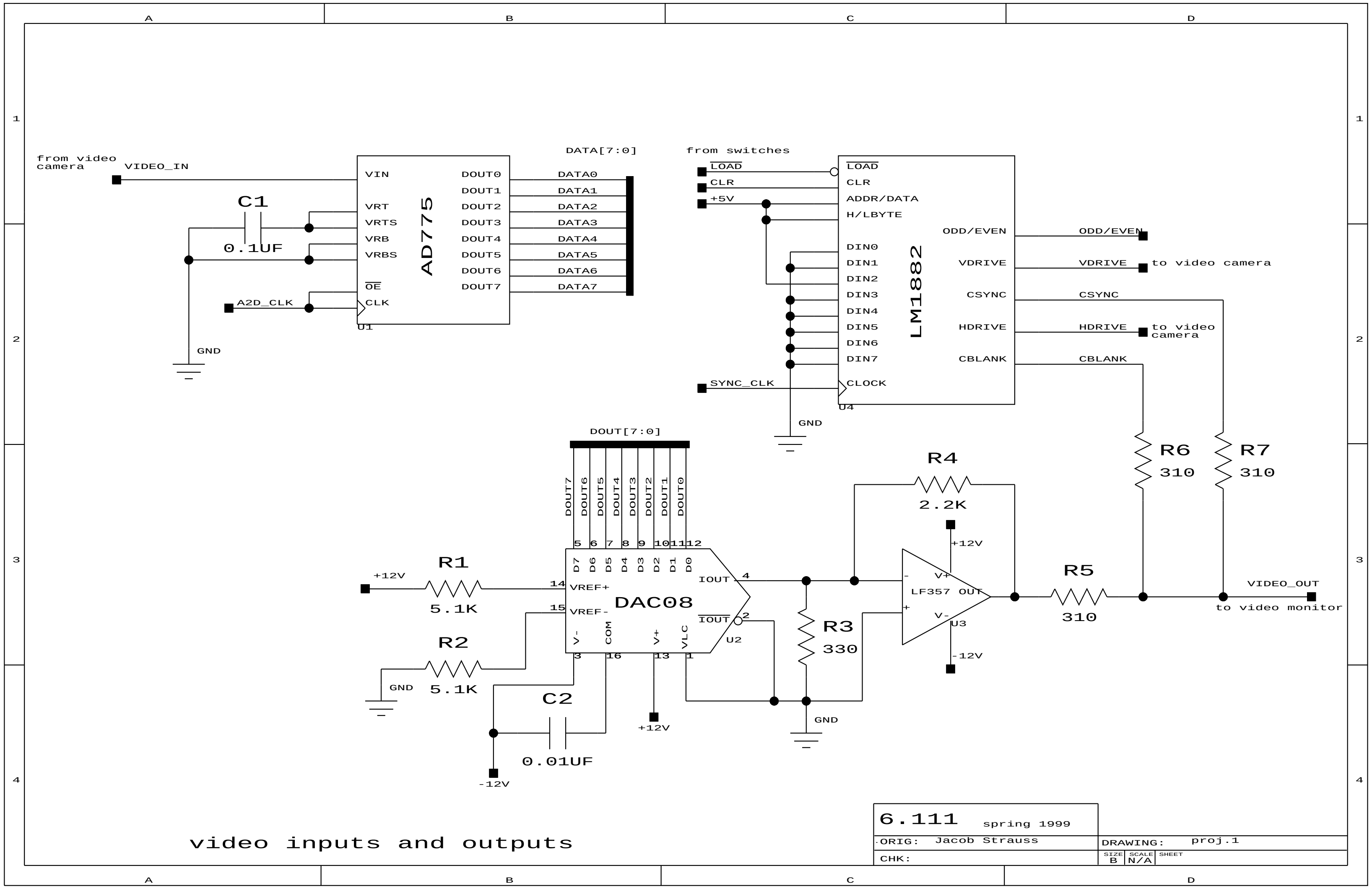


Figure 3: Analog Video I/O Module

2.1.3 Digital to Analog Converter

The outputs from the frame differencer and bitmap overlay module combine to produce a single stream of video data, which is converted to analog and displayed on a greyscale monitor. The converter consists of two chips, a DAC0800, U2 on Figure 3, and an LF357 op amp, which is necessary because the DAC0800 acts as a variable current source rather than a variable voltage source. R3, the 330Ω resistor across the two inputs to the op amp, is an uncommon location for a gain control on an op amp. The LF357 requires that the feedback gain be within a much more restricted range than most op amps, to prevent a pole from transforming the desired negative feedback to unstable positive feedback.

The output of the LF357 and the CBLANK and CSYNC outputs of the LM1882 tie, through resistors, to the input of the display monitor. This linear network ensures that the voltage levels are within acceptable levels for the monitor, and that the sync pulses are placed on the output, as the digital output stream is set to all zeroes during blanking periods.

2.2 Frame Differencer

2.2.1 Overview

The main cycle of the frame differencer consists of a sequence of operations, performed on each pixel in order. First, the DRAM's /OE falls low to read the old pixel intensity out of DRAM; this value is stored in a register. The ADC then outputs the new intensity of the same pixel, and the new and old values are fed to the combinational circuit which calculates the absolute value of the difference of the two values, and the result is latched into a final register. The nine-bit column and row counters serve as the address to DRAM;

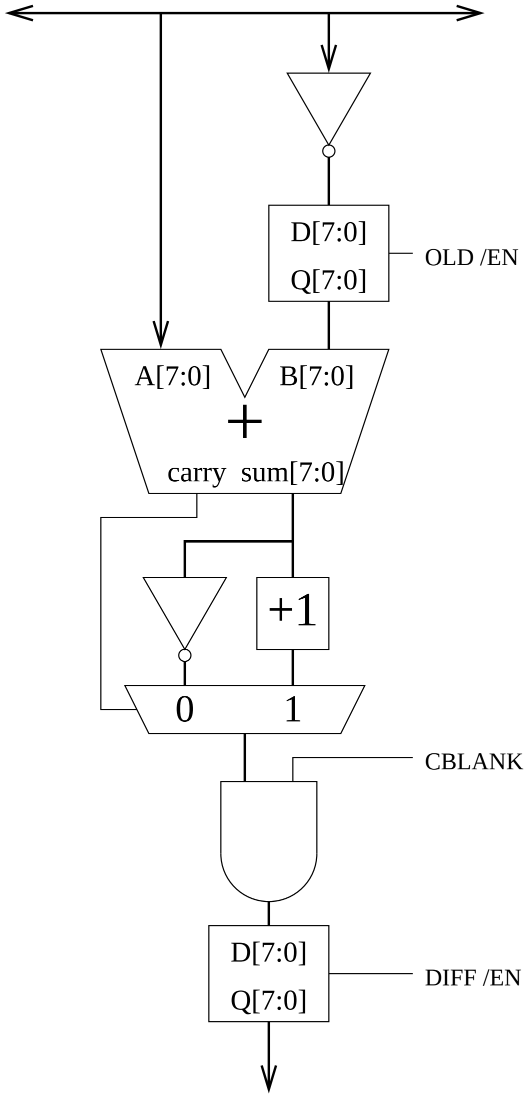


Figure 4: Absolute Difference Block Diagram

the column counter increments after every pixel, and the row counter increments after every row. Figure 6, the timing diagram for the differencer, shows the evolution of the control signals over the pixel cycles, each of which last eight clock cycles. Eight cycles are necessary in order to allow the address and data busses to stabilize before they are read.

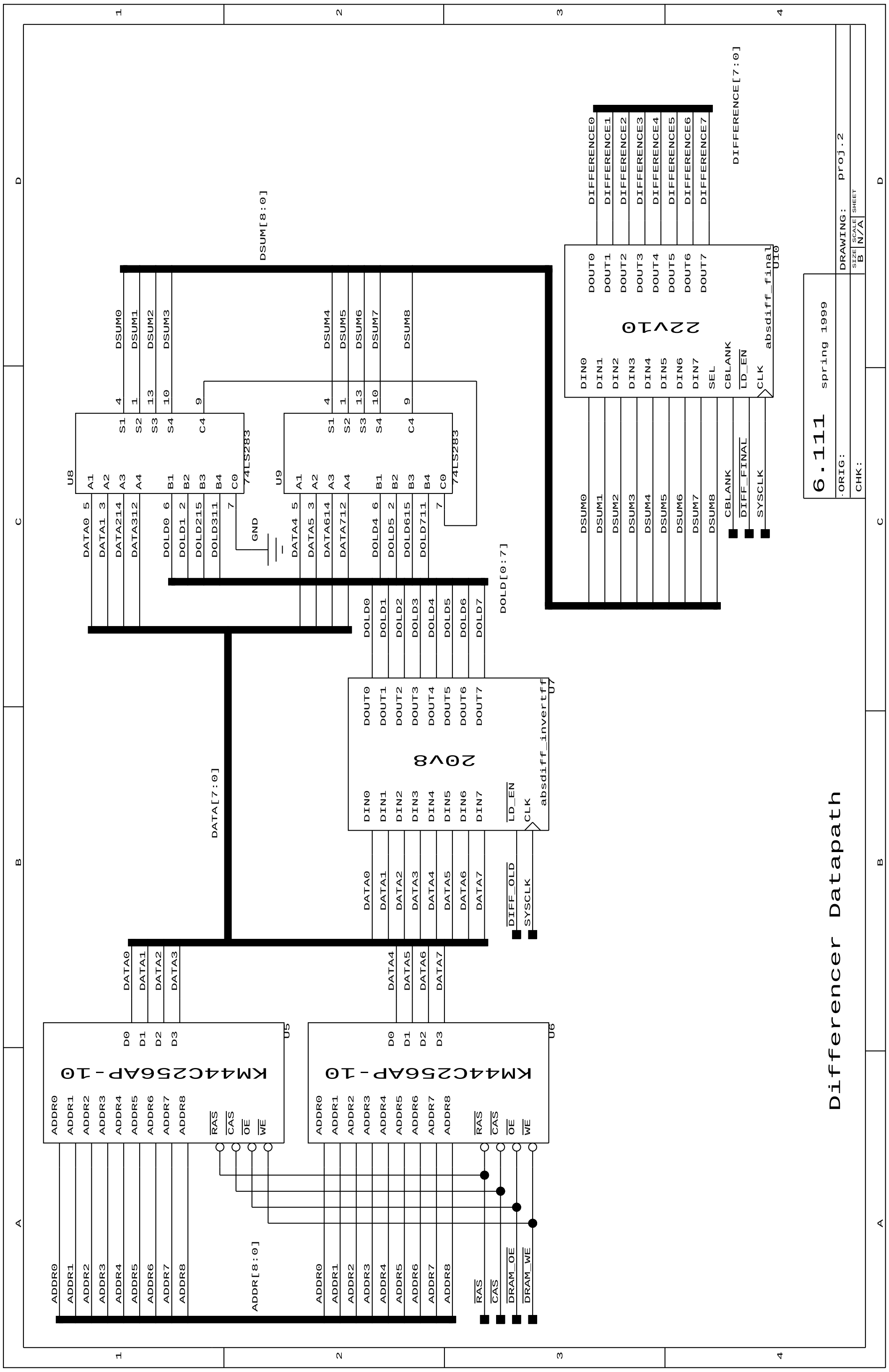


Figure 5: Absolute Differencer Datapath

2.2.2 Absolute Difference Datapath

The differencer computes the absolute value of the difference between the current pixel's intensity and that of the same pixel of the previous frame. Once the two eight-bit input values are on the bus, this is a purely combinational process. The algorithm proceeds as follows:

1. Bitwise invert one of the eight-bit values.
2. Add the inverted signal to the other original eight-bit signal.
3. Use the high carry out of the adder as the select bit to a 16 to 8 multiplexor.
   - If the carry bit is 1, then output the sum plus 1.
   - If the carry bit is 0, then output the bitwise inversion of the sum.

This combinational datapath is implemented in one 22v10, one 20v8, and 2 74LS283's. The first 20v8, U7 on Figure 5, registers the old pixel intensity and inverts it. The two '283's, U8 and U9, add the result to the value on the data bus, which is the output of the A2D. This sum is fed into the 22v10, which uses the carry output of the adder to select whether to output `dsum` plus 1 or `dsum` inverted. This multiplexor is combined with a register in U10, which latches the value until the next pixel value arrives.

2.2.3 Controller

The differencer controller consists of three 22v10 PALs. Two of them, U11 and U12 on Figure 7, are nine-bit counters which index the current pixel position. The other PAL, U13,

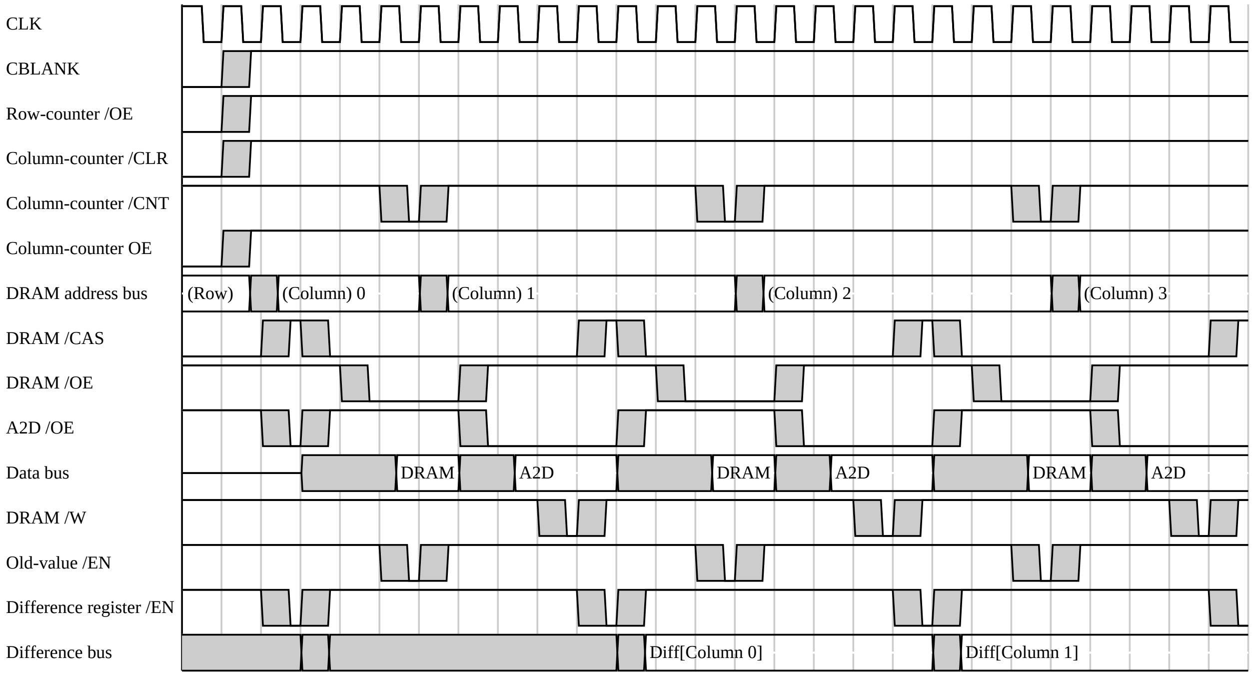


Figure 6: ADC and DRAM controller cycle
The beginning of a line, followed by the first three pixels on that line.

is the main controller fsm, which consists of a three-bit counter and decoding logic to assert and deassert certain signals during the various phases of the pixel cycle, which lasts for eight system clock periods. (See Figure 6) All other control signals are taken from the LM1882 (See Figure 8), with the exception of the `/RAS` row-address-strobe on the DRAM, which is inverted and time-delayed from the HDRIVE in order to meet the propagation delay timing constraint of the row-address counter and the constraint that /RAS be low while accessing the dram.

The row and column address counters are similar, but a few differences must be noted. Their output enables have opposite polarity (negative true for the row, positive true for the column) so that, while these inputs both come from CBLANK, nevertheless only one chip drives the bus at any point. In addition, while the differencer stores entire frames in the DRAM, the input is in the form of interlaced odd and even fields; thus, the ODD/EVEN

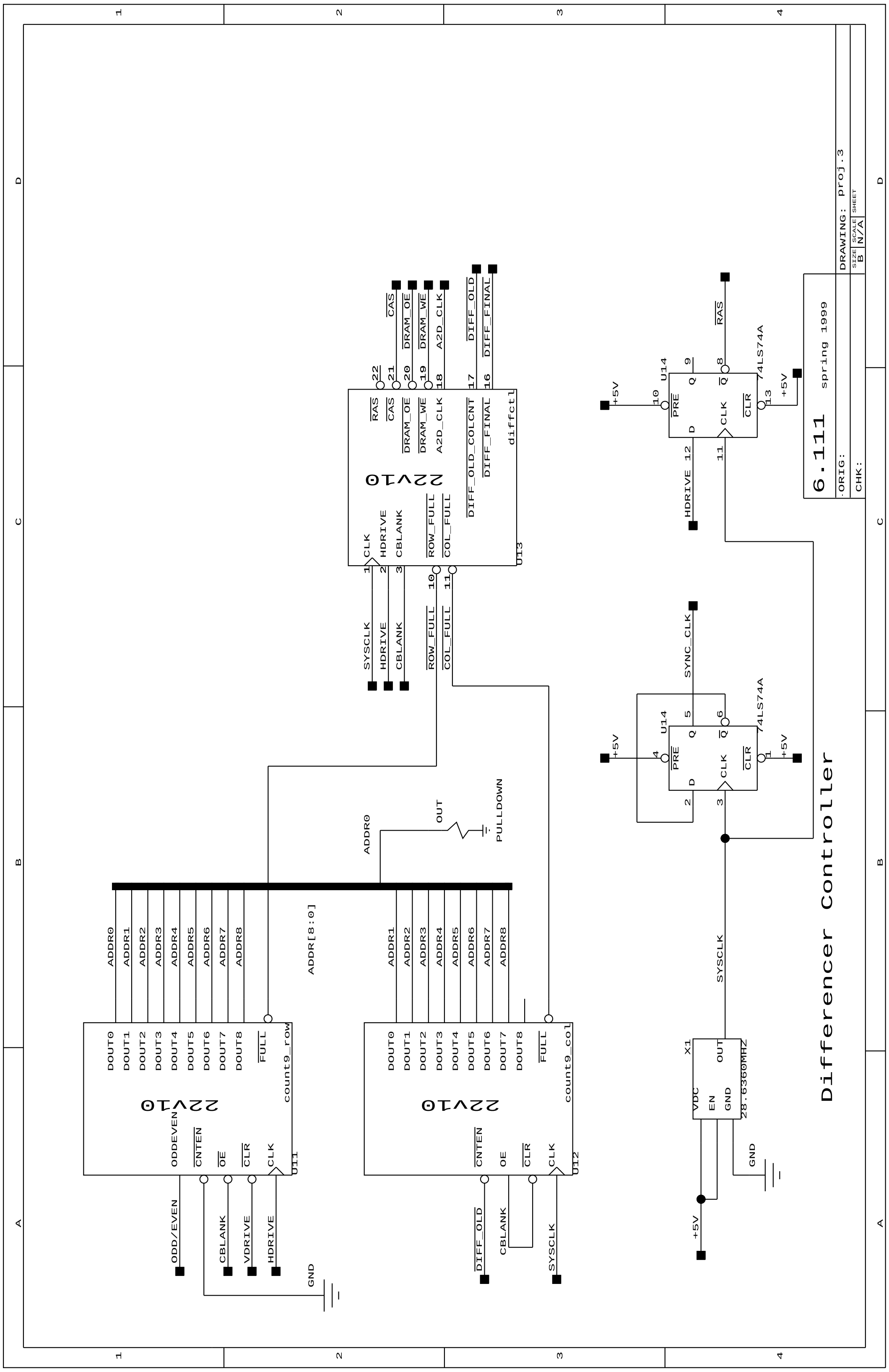


Figure 7: Differencer Controller

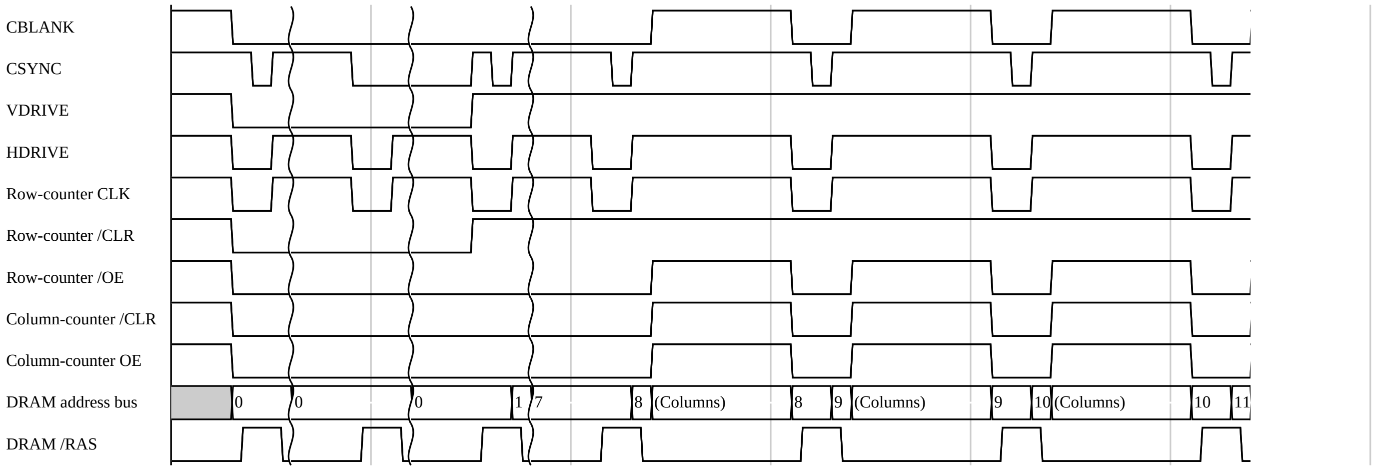


Figure 8: Long-Term Timing
The beginning of a field and the first three rows of that field.

bit from the LM1882 is the low bit of the row address. Lastly, the counters have internal comparators which cause the `/FULL` bit to go low when the counter passes a certain constant; for the column, this constant is 180, and for the row, it is 250. This gives 180 columns and roughly 500 lines on the screen (somewhat less because of sloppiness in the handling of the edge conditions).

2.3 Pong Rules Module

The rules module keeps track of the motion of the ball, and updates it according to information about regions of motion in the field of view of the camera. It implements a very simple version of the game Pong, with rules as follows:

- The ball's x and y positions increment by certain values every frame, which do not change unless the ball hits something.
- If the ball hits the top or the bottom of the screen, it reverses its y direction and continues as before.

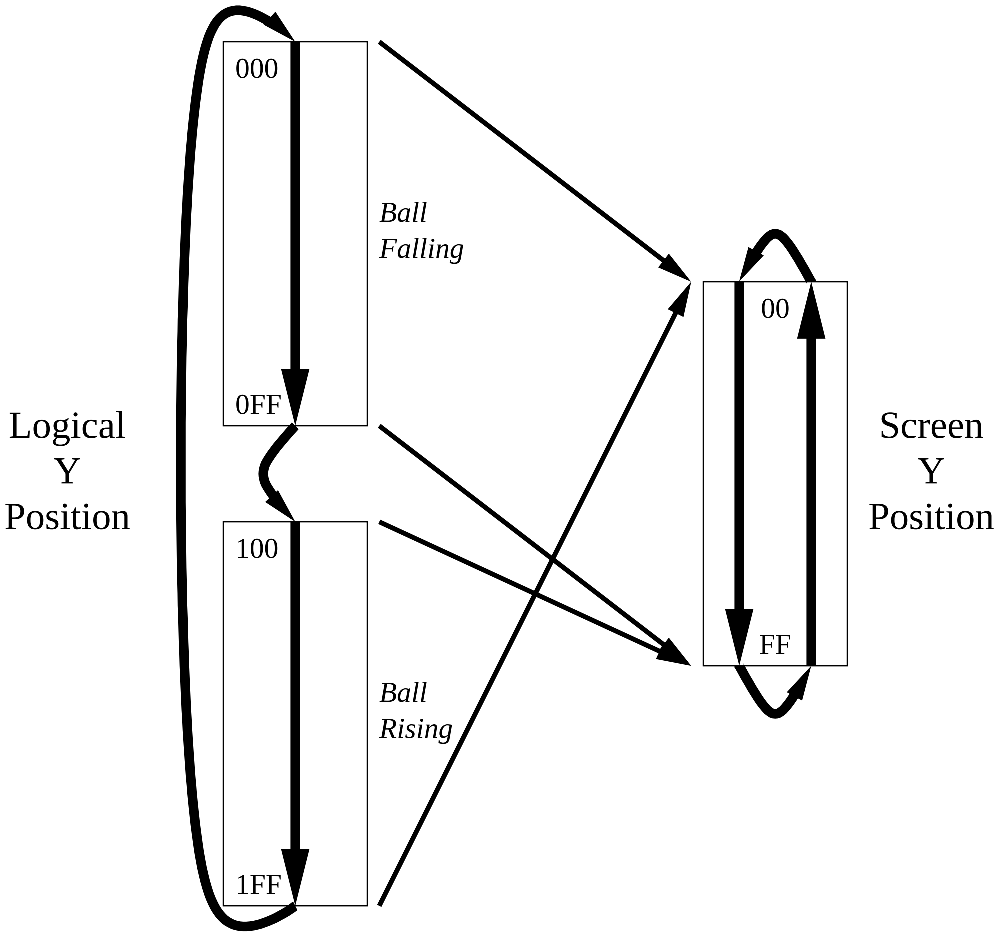


Figure 9: Internal encoding of the ball's Y position

- If the ball hits a region of motion while in a "bounce region", it reverses its x direction and randomizes its x and y velocities and y direction.
- If the ball goes off the left or right edge of the screen, the appropriate player loses a point, and the ball is returned to the center of the screen, where it remains motionless until hit by a region of motion.

A clever algorithm is used to automate the bouncing of the ball. The internal ball x and y positions are nine-bit counters which contain information not only about the position of the ball, but also about its direction of motion. The encoding for the y position is the simpler

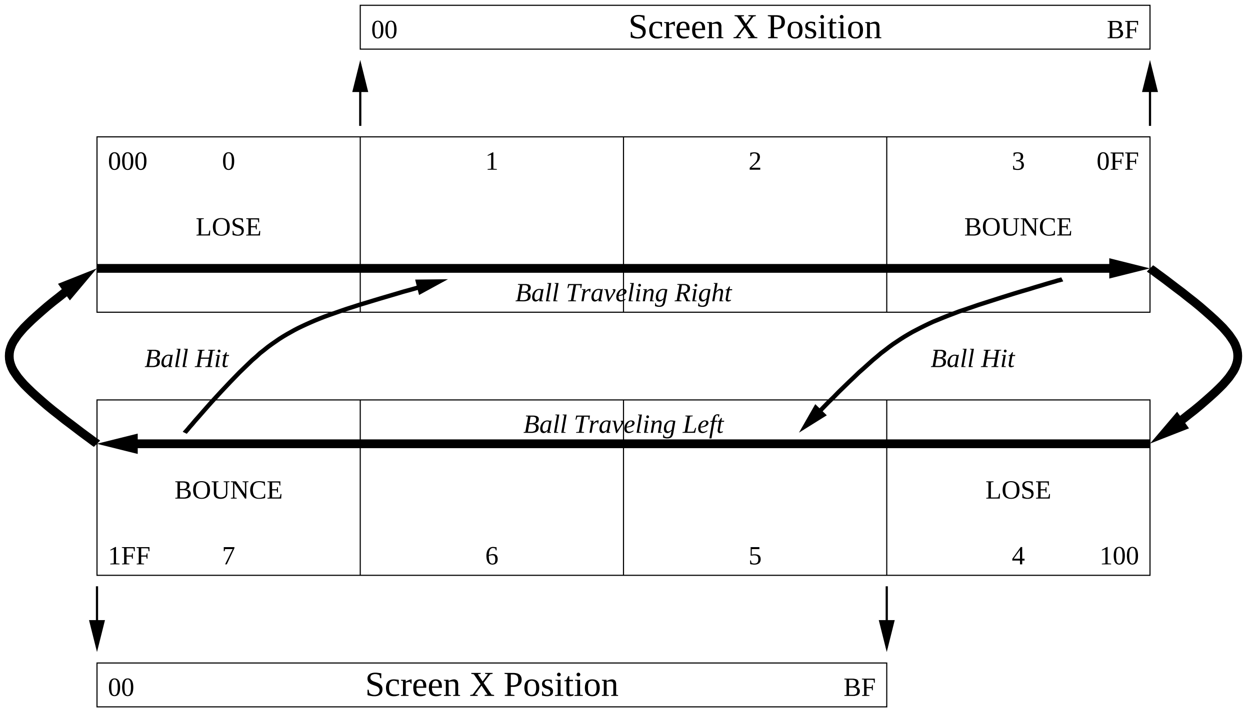


Figure 10: Internal encoding of the ball's X position

of the two (see Figure 9): if the high bit is set, then the ball is going up, and if clear, then the ball is going down. The visual position of the ball is obtained by XORing the high bit of the logical position with every other bit; the y position is updated every cycle by adding a two-bit velocity to it. The result of this is that the ball moves in the appropriate direction at any time, and when the ball hits the top or bottom of the screen, its position simply wraps around and the high bit toggles, with the result that the visual representation of the ball "bounces" from the wall (although the logical position of the ball is still incrementing in the same "direction".)

The x position is somewhat more complicated, since it determines the behavior of the ball with respect to the users. The top three bits of the nine-bit accumulator (which, like y, increments by a two-bit velocity every frame) determine the behavior of the ball; thus, the position space is logically divided into eight segments, shown in Figure 10. Segments 1 through 3 map to the screen by subtracting 64, while segments 5 through 7 map to the screen by a bitwise NOT. The ball begins in either segment 2 or segment 6 (thus, the initial direction is random), and as it increments, it moves in the appropriate direction on the screen, and goes through the segments in order. Segments 3 and 7 are "bounce" segments, i.e. if the ball touches a region of movement while in either of those segments, it will follow the "bounce" arrow shown by the diagram by bitwise inverting and adding 64. The visual result of this is a reversal of the direction of the ball; because the segment a bounce action jumps to is not a "bounce" segment, the ball will not bounce again until it has reached the "bounce" segment on the other side of the screen. If the ball passes through either bounce segment without touching a region of motion, it will pass though into either segment 4 or segment 0, which are the "lose" segments. If the ball is found to be in a "lose" segment, the appropriate player (determined by the segment number) loses a point, and the ball returns to the center of the screen.

When the ball is bounced from a region of motion, the x and y velocities and the y direction are randomized. The random data to accomplish this is fed in from the low bit of the A2D output and accumulated (via a bit-shifting process) into internal registers in the CPLD (which performs most of the tasks of the rules module). Special care is taken to ensure that the x velocity is never zero, for if it ever became zero, then the ball would bounce up and down in the same segment forever.

All that remains to be described is the method of detection of a region of motion. Two 20v8 PALs are programmed as eight-bit-to-eight-bit equality comparators: one compares the

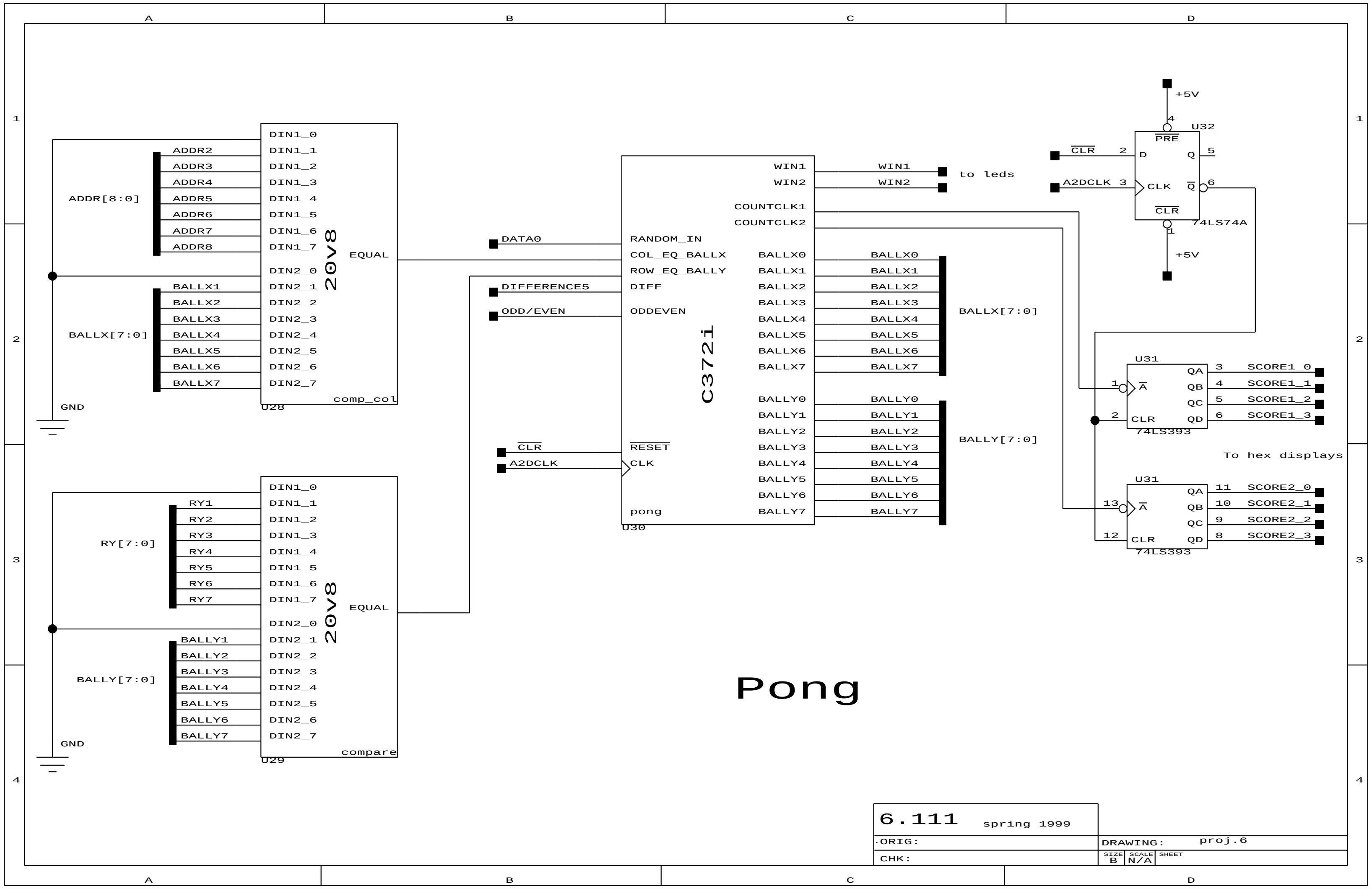


Figure 11: Pong Rules Module

ball's screen x position with the current pixel x coordinate, and the other compares the ball's screen y position with the current pixel y coordinate. (A tweak to increase sensitivity was to ground the low bits of each compared value, effectively increasing the motion detection area to four pixels instead of one.) Each outputs a single bit which is high if the values are equal. The CPLD takes these two bits, and a single bit of the difference stream, logically ANDs all three, and decides that the ball has hit a region of motion if the result is ever true over the course of a frame. The effect is that the ball has "hit" something if the difference value is greater than a certain threshold at the screen position of the ball; if so, and if the ball is in a "bounce" region, the CPLD will perform the "bounce" operation.

2.4 Bitmap Overlay Module

The Bitmap Overlay module takes the difference stream, the current pixel coordinates, and the ball position as inputs, and outputs a digital video stream which consists of the difference stream with an image overlaid at the position of the ball. The circuit is entirely combinatorial; see Figure 12.

By bitwise inverting one input to each of two adders, the circuit finds the differences between the current pixel location's and the current ball location's x and y coordinates. These two eight-bit offsets form a sixteen-bit address, the high thirteen bits of which become the low thirteen bits of the PROM address, and the low three bits of which are used to select one of the eight bits which the PROM outputs (using a '151 8-to-1 multiplexor).

The two high address lines on the PROM come from a slowly cycling '393; in this way, the counter selects among four images in sequence. The animation cycle lasts roughly half

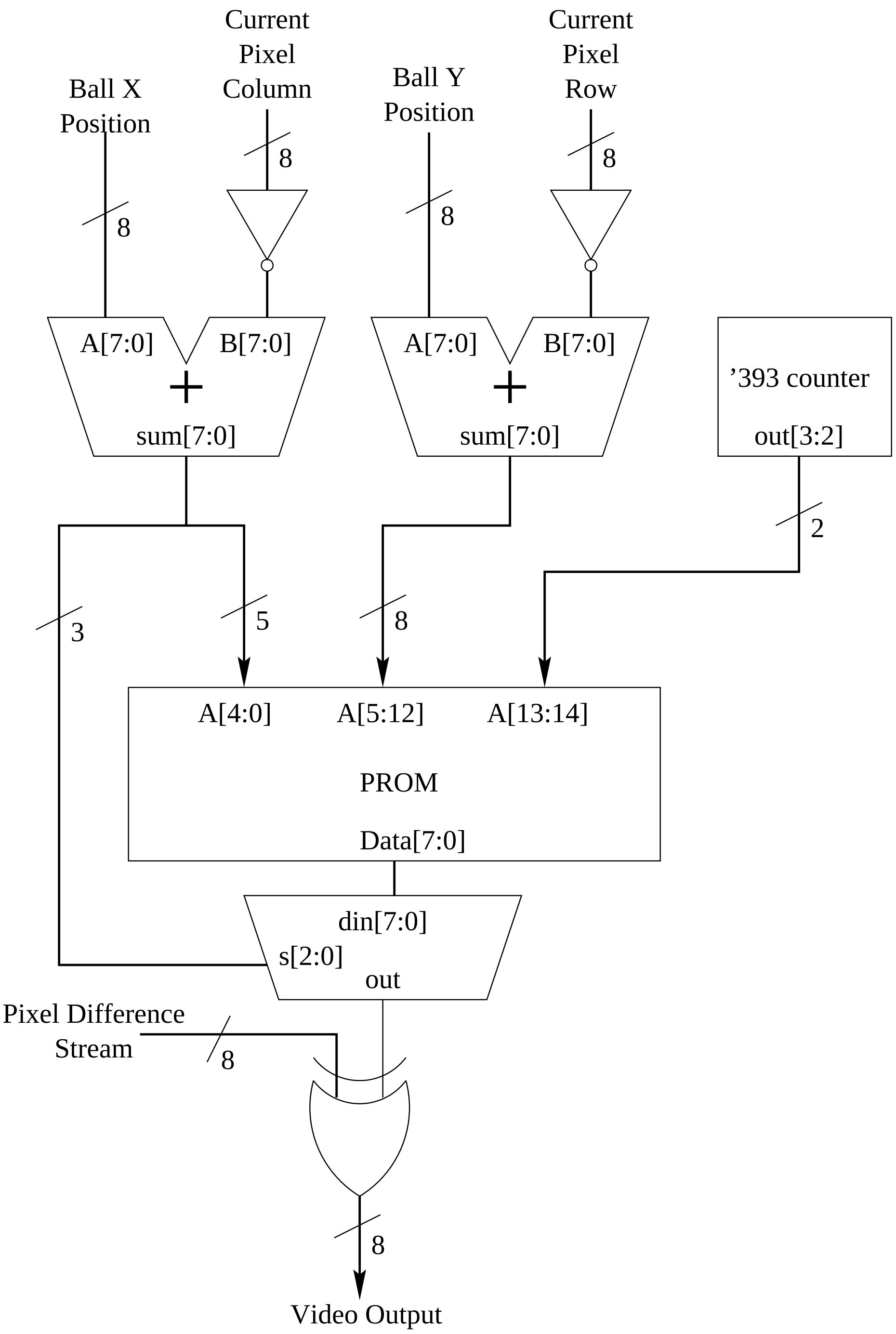


Figure 12: Block Diagram for Bitmap Overlay

a second.

The single-bit output of this lookup table is XORed with every bit of the difference bus, and the result is fed to the DAC; thus, where the lookup table is zero, the output is the difference, and where it is one, the output is the inverse of the difference, which is usually white or very close to white.

Because this is the end of a long chain of combinational devices, the output of the '151 is glitchy, which tends to produce lines on the display monitor. Adding a small (1000pF) capacitor to the output of the multiplexor diminished these glitches enough to render them invisible, although they are actually still present at a small magnitude.

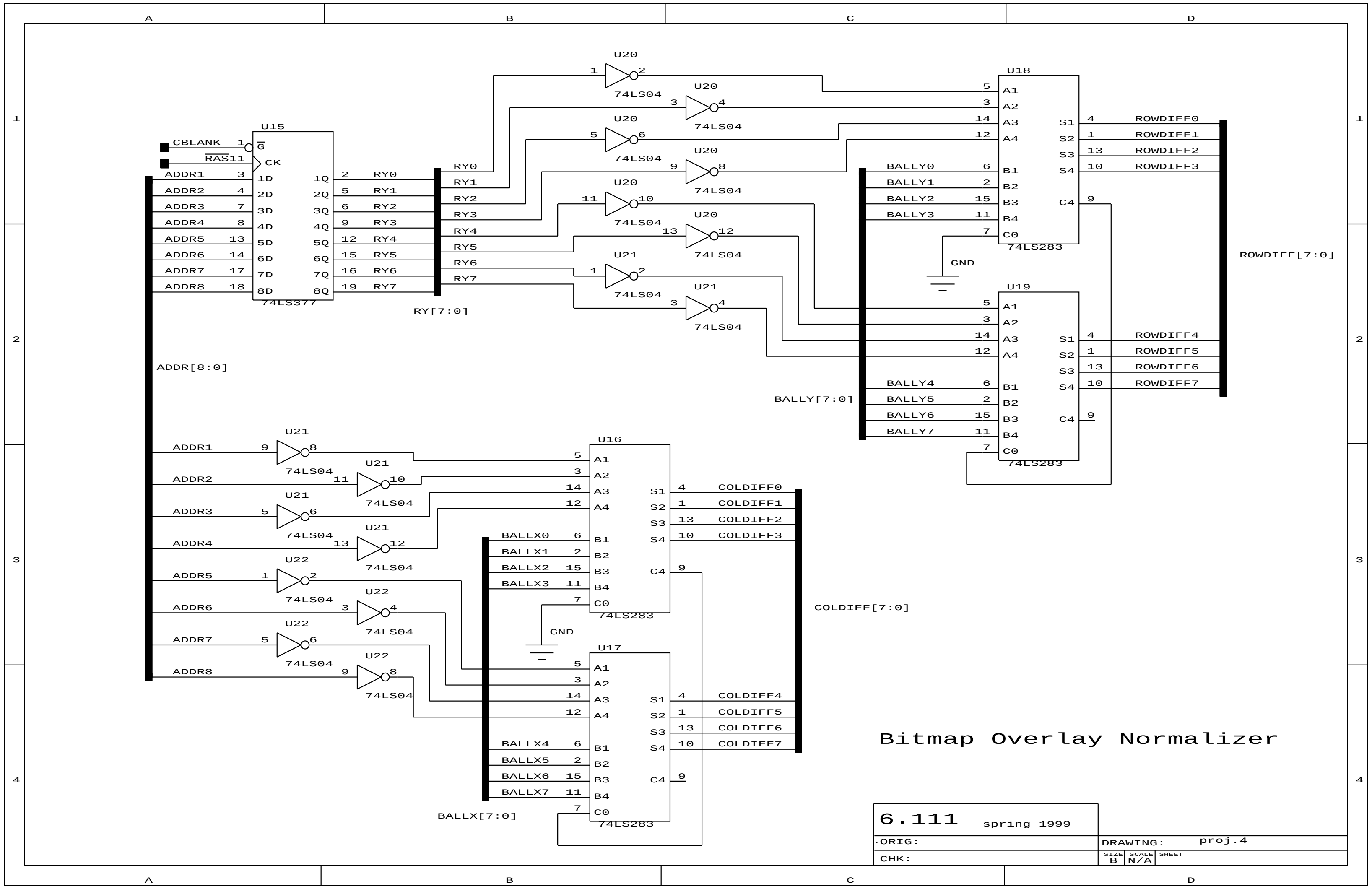


Figure 13: Bitmap Overlay Offset

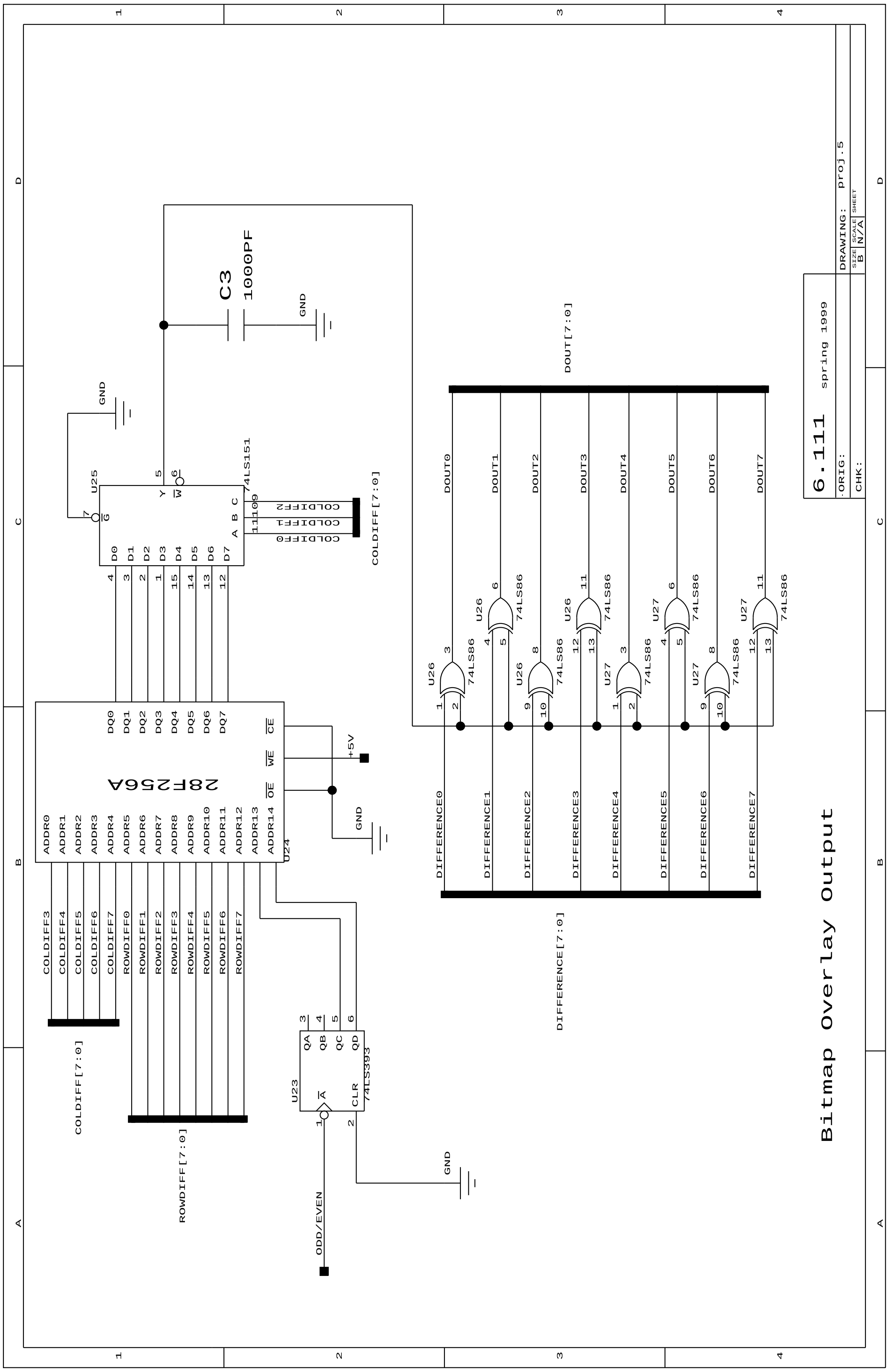


Figure 14: Bitmap Overlay Output

3 Conclusion

After an extensive period of continuous refinement, redesign, and debugging, sometimes encouraging, sometimes discouraging, the final result of this project well exceeded expectations. The device is impressive both in technical scope and entertainment value.

The greatest difficulties in designing and building this project were in the analog components and the DRAM timing. Since this was expected, we begin with the analog I/O module, passing the digital output of the ADC directly to the input of the DAC to form a pass-through analog video device. Making this work correctly took quite a long time, most of which was spent on the DAC. Much of this difficulty arose from the op amp connected to the output: this chip could not be modeled as an ideal op amp. If it could, then the resistor across the inputs would be irrelevant, but in reality it keeps the output from ringing uncontrollably. A cheap, fast, voltage based DAC would have been better, but was not available to us.

The differencer module's control FSM went through several design revisions before the final version. We spent a good deal of time experimenting with timing delays and control signals before the module actually performed time differencing: most configurations we tested differenced horizontally or vertically adjacent pixels, giving an edge detector rather than a motion detector. Originally, we had planned to process one pixel for every four clock cycles; however, the timing constraints for the DRAM and the various PALs were only barely met by this design, and the amount of digital noise in the signal was unacceptable. Thus, we moved on to a six-clock-per-pixel revision, which was an improvement, but not sufficient. The final eight-clock-per-pixel design satisfies all chips' timing requirements, and the resulting difference signal has only minimal (on the order of the least significant bit) noise.

Originally, we had intended build a targeting system. The change to a pong game was actually faily minor, and did not affect the I/O, Differencer, or Bitmap Overlay modules. Only the section that controlled the position of the "ball" (originally a target) needed modification. The targeting system would have tracked moving objects by monitoring the stream of differences and placing the target on the region of greatest motion. However, pong is a far more amusing and captivating project, and it more effectively demonstrates the capabilities of the differencer.

The implementation of bouncing in the CPLD was a particularly clever part of the project: the naive way to do this would be to implement a signed velocity and an eight-bit accumulator, and implement both addition and subtraction in the accumulator. However, this naive approach would have taken far too many resources, and would have resulted in the splitting of the pong module over multiple CPLDs, which would have been very inelegant. The separation of logical positions and screen positions was an elegant solution to this problem, allowing the logic to treat the ball as if it moved in one direction only.

The other prideworthy aspect of the design was the tightness and efficiency of the differencer. In order to operate at the speeds we desired, we ordered a 28.8 MHz clock and a high-speed DAC from an electronics supplier. This speed pushed most of our components to their limits, and all the timing constraints were met only after a we did a great deal of work on the timing diagrams. The high speed of our data transfer required us to keep all of the circuitry on one lab kit to limit noise and delays, which was quite a handicap to overcome: by the end, we had used every row of every protoboard, and the ICs were packed bumper to bumper. Finally, our decision to perform all operations synchronously — instead of delaying operations until the blanking periods of the video signal — allowed us to have such an efficient and flexible design.

Unlike many of the other video games built for 6.111 projects, this project could not be written simply as software for an entertainment system. Disregarding the fact that most such systems have no camera, most entertainment system CPUs are not fast enough to implement a full frame rate differencer in software. A hardware solution such as ours will always be more efficient than software.

In conclusion, a purely visual interface, involving no buttons, switches, keyboards, or keypads, is far more intuitive and engaging than the traditional interfaces to digital devices. A similar design to ours could well be very successful in a “real-world” design situation geared towards non-technical users.

4 Appendix

A Differencer Components Source Code

A.1 Differencer latch and invert: `absdiff_invertff.vhd`

This is the VHDL source for the 20v8 that is part of the differencer that latches old intensity value and inverts it. This is U7 in Figure 5.

```
-- An input register to the differencer: not only registers, but
-- inverts as well!  Oooh!
library ieee;
use ieee.std_logic_1164.all;

entity absdiff_invertff is
  port (clk, n_lden : in std_logic;
        input  : in  std_logic_vector(7 downto 0);
        output : out std_logic_vector(7 downto 0));
  attribute pin_numbers of absdiff_invertff:entity is
    "clk:1 " &
    "input(0):3 input(1):4 input(2):5 input(3):6 " &
    "input(4):7 input(5):8 input(6):9 input(7):10 " &
    "output(0):22 output(1):21 output(2):20 output(3):19 " &
    "output(4):18 output(5):17 output(6):16 output(7):15 ";
end absdiff_invertff;

architecture absdiff_invertffarch of absdiff_invertff is
begin
  process (clk)
  begin
    if rising_edge(clk) then
      if n_lden = '0' then
        output <= not input;
      end if;
    end if;
  end process;
end absdiff_invertffarch;
```

A.2 Differencer Mux and Latch: `absdiff_final.vhd`

This is the VHDL source for the 20v8 that is part of the differencer that muxes and outputs the final difference value. This is U10 in Figure 5.

```
-- Final stage of the differencer PALs.
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;

entity absdiff_final is
  port (clk, n_lden : in std_logic;
        cblank, sel : in std_logic;
        in_val  : in  unsigned(7 downto 0);
        out_val : out unsigned(7 downto 0));
  attribute pin_numbers of absdiff_final:entity is
    "clk:1 cblank:2 sel:11 n_lden:13 " &
    "in_val(0):3 in_val(1):4 in_val(2):5 in_val(3):6 " &
    "in_val(4):7 in_val(5):8 in_val(6):9 in_val(7):10 " &
    "out_val(0):22 out_val(1):21 out_val(2):20 out_val(3):19 " &
    "out_val(4):18 out_val(5):17 out_val(6):16 out_val(7):15 ";
end absdiff_final;

--sel is the top bit of the adder
--cblank sets all outputs to zero during blanking periods

architecture absdiff_finalarch of absdiff_final is
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if cblank = '0' then
        out_val <= (others=>'0');
      elsif n_lden = '0' then
        if sel = '1' then
          out_val <= in_val + 1;
        else
          out_val <= not in_val;
        end if;
      end if;
    end if;
  end process;
end absdiff_finalarch;
```

A.3 Differencer Controller: `count8-diffctl.vhd`

This is the VHDL source for the 22v10 that controls the differencer counters, DRAM, and output latches. U13 on Figure 7

```
-- Control FSM for the video differencer.
```

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;

entity diffctl is
  port (clk,                      -- System clock
        hdrive,                   -- HDRIVE from LM1882
        cblank,                   -- CBLANK from LM1882
        row_n_full,               -- Row > 250
        col_n_full                -- Col > 180
        : in std_logic;

        dram_n_ras                --this exists from an earlier
                                  --revision, but this signal is
                                  --no longer used
        : buffer std_logic;
        dram_n_cas,
        dram_n_oe,
        dram_n_we,
        a2d_n_oe,
        diff_old_en_colcnt,
        diff_final_en
        : out std_logic;

        count :
        buffer unsigned(2 downto 0));

  attribute pin_numbers of diffctl:entity is
    "clk:1 hdrive:2 cblank:3 row_n_full:10 col_n_full:11 " &
    "dram_n_ras:22 " &
    "dram_n_cas:21 dram_n_oe:20 dram_n_we:19 " &
    "a2d_n_oe:18 diff_old_en_colcnt:17 diff_final_en:16";
end diffctl;

architecture diffctlarch of diffctl is
begin
  process(clk)
  begin
    if rising_edge(clk) then
      --the counter need to be reset at the beginning of each line
      --and at the end of each pixel, which is when the counter
      --wraps every 8 clocks
      if hdrive = '0' then
        count <= "001";
      elsif cblank = '0' then
        count <= "111";
      else
```

```
    count <= count + 1;
  end if;

  --the delayed signal from the LM1882 is used instead of
  --this one.
  dram_n_ras <= hdrive;


  --only update signals when the line is valid:
  --cblank is high for the duration of each valid row
  if cblank = '1' and row_n_full = '1' and col_n_full = '1' then

    --hold cas low for the duration of the pixel, pulse it to latch
    --the next value into the dram at the end of each pixel
    if count /= "111" then
      dram_n_cas <= '0';
    else
      dram_n_cas <= '1';
    end if;

    --dram output is enabled for 3 of the clocks,
    --sampled at the end of the second of the three
    if count = "001" or count="010" or count="011" then
      dram_n_oe <= '0';
    else
      dram_n_oe <= '1';
    end if;

    --the column counter has to update sometime when cas has already been
    --brought low
    --old_en latches the old value out of the dram
    if count = "010" then
      diff_old_en_colcnt <= '0';
    else
      diff_old_en_colcnt <= '1';
    end if;

    --a2d output enable and clock are the same signal
    --output enabled during the second half of the cycle
    --after the dram oe is brought high again
    if count = "100" or count="101" or count = "110" or count="111"
    then
      a2d_n_oe <= '0';
    else
      a2d_n_oe <= '1';
    end if;
```

```
        --write the new value in after the a2d has had two clocks to settle
        if count = "110" then
          dram_n_we <= '0';
        else
          dram_n_we <= '1';
        end if;

        --latch the difference value after the adders have time to settle
        if count = "111" then
          diff_final_en <='0';
        else
          diff_final_en <='1';
        end if;

      else
        --turn everything off
        dram_n_cas <= '1';
        dram_n_oe <= '1';
        dram_n_we <= '1';
        a2d_n_oe <= '1';
        diff_old_en_colcnt <= '1';
        diff_final_en <= '1';
      end if;
    end if;
  end process;
end diffctlarch;
```

A.4 Column Counter: count9_col.vhd

This is the VHDL source for the 22v10 that counts the columns for the address bus, U12 in Figure 7.

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;

entity count9_col is
  port (clk,              -- System clock
        n_clr,            -- Clear counter
        n_cnten,          -- Count enable
        oe                -- Output enable
        : in std_logic;

        n_full            -- count > 180 (negative true)
        : out std_logic;
```

```
        cnt_out            -- Counter output
        : out unsigned(8 downto 0));

  attribute pin_numbers of count9_col:entity is
    "clk:1 n_clr:2 n_cnten:3 oe:4 " &
    "n_full:14 " &
    "cnt_out(0):23 cnt_out(1):22 cnt_out(2):21 cnt_out(3):20 " &
    "cnt_out(4):19 cnt_out(5):18 cnt_out(6):17 cnt_out(7):16 " &
    "cnt_out(8):15";
end count9_col;

--9 bit counter with count and clear, and output enables
--n_full asserted when at the right end of the screen

architecture count9_colarch of count9_col is
  signal count : unsigned(8 downto 0);
begin
  cnt_out <= count when oe = '1' else (others=>'Z');
  process(clk)
  begin
    if rising_edge(clk) then
      if n_clr = '0' then
        count <= (others=>'0');
      elsif n_cnten = '0' then
        count <= count + 1;
      end if;

      if n_clr = '0' then
        n_full <= '1';
      elsif count = 180 then
        n_full <= '0';
      end if;
    end if;
  end process;
end count9_colarch;
```

A.5 Row Counter: `count9_row.vhd`

This is the VHDL source for the 22v10 that counts the rows for the address bus, U11 in Figure 7.

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
```

```
entity count9_row is
  port (clk,                 -- System clock
        n_clr,               -- Clear counter
        n_cnten,             -- Count enable
        n_oe                 -- Output enable
        : in std_logic;

        oddeven              -- Odd/even field (LSB of address)
        : in std_logic;

        n_full               -- count > 250 (negative true)
        : out std_logic;

        cnt_out              -- Counter output
        : out unsigned(8 downto 0));
  attribute pin_numbers of count9_row:entity is
    "clk:1 n_clr:2 n_cnten:3 n_oe:4 " &
    "oddeven:11 " &
    "n_full:14 " &
    "cnt_out(0):23 cnt_out(1):22 cnt_out(2):21 cnt_out(3):20 " &
    "cnt_out(4):19 cnt_out(5):18 cnt_out(6):17 cnt_out(7):16 " &
    "cnt_out(8):15";
end count9_row;

--almost the same as column counter:
--8 bit counter->9 bit address combined with the odd/even input to
--interlace the rows

architecture count9_rowarch of count9_row is
  signal count : unsigned(7 downto 0);
begin
  cnt_out <= count & oddeven when n_oe = '0' else (others=>'Z');
  process(n_clr, clk)
  begin
    if rising_edge(clk) then
      if n_clr = '0' then
        count <= (others=>'0');
      elsif n_cnten = '0' then
        count <= count + 1;
      end if;

      if n_clr = '0' then
        n_full <= '1';
      elsif count = 250 then
        n_full <= '0';
      end if;
```

```
    end if;
  end process;
end count9_rowarch;
```

B Pong Components Source Code

B.1 8 Bit Comparator: `comp.pal`

This is the PALASM source for the 20v8 that compares the current pixel row with the ball y coordinate. U29 on Figure 11.

```
20v8 c i i i /c /c /c c
an eight bit comparator
dout is true if the two
8 bit inputs are equal
din1_0 din1_1 din1_2 din1_3 din1_4 din1_5 din1_6 din1_7 din2_0 din2_1 din2_2 GND
din2_3 din2_4 pin15 din2_5 din2_6 din2_7 mid lowout upout dout pin23 VCC

    dout = lowout * upout * mid

    /lowout = din1_0 * /din2_0
        + /din1_0 * din2_0
        + din1_1 * /din2_1
        + /din1_1 * din2_1
        + din1_2 * /din2_2
        + /din1_2 * din2_2

    /mid = din1_3 * /din2_3
        + /din1_3 * din2_3
        + din1_4 * /din2_4
        + /din1_4 * din2_4

    /upout =
         din1_5 * /din2_5
        + /din1_5 * din2_5
        + din1_6 * /din2_6
        + /din1_6 * din2_6
        + din1_7 * /din2_7
        + /din1_7 * din2_7
```

B.2 8 Bit Comparator: comp_col.pal

This is the PALASM source for the 20v8 that compares the current pixel column with the ball x coordinate. U28 on Figure 11

```
20v8 c i i i /c /c /c /c
8 bit comparator
dout is true if the two input values are equal,
except for the very left edge of the screen to compensate for edge effects
din1_0 din1_1 din1_2 din1_3 din1_4 din1_5 din1_6 din1_7 din2_0 din2_1 din2_2 GND
din2_3 din2_4 mid din2_5 din2_6 din2_7 notedge lowout upout dout pin23 VCC

    /dout = /lowout
        + din1_3 * /din2_3
        + /din1_3 * din2_3
        + din1_4 * /din2_4
        + /din1_4 * din2_4
        + /upout
        + /notedge

    /notedge = /din1_1 * /din1_2 * /din1_3 * /din1_4
        * /din1_5 * /din1_6 * /din1_7

    /lowout =
          din1_0 * /din2_0
        + /din1_0 * din2_0
        + din1_1 * /din2_1
        + /din1_1 * din2_1
        + din1_2 * /din2_2
        + /din1_2 * din2_2

    /upout =
          din1_5 * /din2_5
        + /din1_5 * din2_5
        + din1_6 * /din2_6
        + /din1_6 * din2_6
        + din1_7 * /din2_7
        + /din1_7 * din2_7
```

B.3 Pong CPLD: pong.vhd

This is the VHDL source for the Cypress 372i CPLD that runs Pong. This chip is shown as U30 in Figure 11.

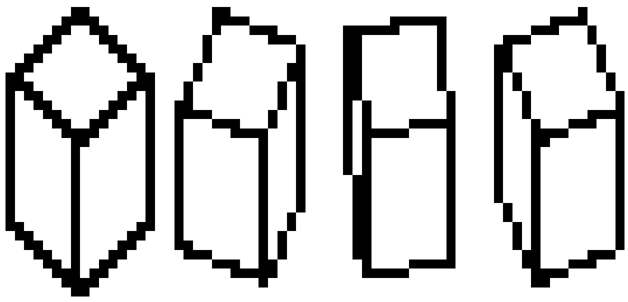
```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;

entity pong is
  port (clk,
        n_reset,
        oddeven,              -- from LM1882
        random_in             -- low bit of data bus should
                              -- be sufficiently random,
                              -- though not very well distributed
        : in std_logic;

        col_eq_ballx,         -- inputs from the comparator pals
        row_eq_bally,
        diff                  -- one upper bit of the difference
                              -- stream.  valid on falling edge
                              -- of the clock.
        : in std_logic;

        reverse, vrandomize   -- on outputs for debugging purposes
        : buffer std_logic;

        -- win1 and win2 to LEDs; countclk[12] clock the '393 score counters
        win1, win2, countclk1, countclk2
        : buffer std_logic;

        ballx, bally          -- output of the position
                              -- in screen coords
        : buffer unsigned(7 downto 0));
  attribute pin_numbers of pong:entity is
    "clk:13 n_reset:10 oddeven:32 random_in:33 diff:35 " &
    "ballx(0):31 ballx(1):30 ballx(2):29 ballx(3):28 " &
    "ballx(4):27 ballx(5):26 ballx(6):25 ballx(7):24 " &
    "bally(0):43 bally(1):42 bally(2):41 bally(3):40 " &
    "bally(4):39 bally(5):38 bally(6):37 bally(7):36 " &
    "col_eq_ballx:2 row_eq_bally:18 " &
    "countclk1:14 countclk2:15 win1:16 win2:17 ";
end pong;

architecture pongarch of pong is
  signal latched_diff : std_logic;          -- diff is invalid on rising edge
                                            -- of clk (A2DCLK), thus is stored
  signal old_oddeven : std_logic;           -- for detecting edges on oddeven
  signal clock_2 : std_logic;               -- clock, frequency divided by 2
  signal x, y : unsigned(8 downto 0);       -- internal logical coordinates
  signal vx, vy : unsigned(1 downto 0);     -- current velocity
```

```
  signal random_vx : unsigned(1 downto 0); -- - random bit accumulators
  signal random_vy : unsigned(2 downto 0); -- /
--signal reverse, vrandomize : std_logic;
  signal ymsb : unsigned(7 downto 0);
  signal padded_vx, padded_vy : unsigned(8 downto 0);
begin
  -- Logical coordinates to screen coordinates map.
  ballx <= x(7 downto 0) - 64 when x(8) = '1' else not x(7 downto 0);
  ymsb <= (others=>y(8));
  bally <= ymsb xor y(7 downto 0);

  padded_vx <= "0000000" & vx;
  padded_vy <= "0000000" & vy;

  countclk1 <= not win1;
  countclk2 <= not win2;

  process(clk)
  begin
    if falling_edge(clk) then
      latched_diff <= diff;
    end if;
    if rising_edge(clk) then
      clock_2 <= not clock_2;
      if clock_2 = '0' then
        -- Ensures that random_vx is never zero.
        if random_vx(0) /= '0' or random_in /= '0' then
          random_vx <= random_vx(0) & random_in;
        end if;
      else
        random_vy <= random_vy(1 downto 0) & random_in;
      end if;

      -- n_reset = '0' means that the external reset has been triggered.
      -- x(7 downto 6) = "00" means that we are in a "lose" segment.
      if n_reset = '0' or x(7 downto 6) = "00" then
        -- set the "win indicators"
        win1<= x(8);
        win2<= not x(8);
        -- reset the ball to the center of the screen and randomize direction.
        x <= "010011010";
        reverse <= random_vy(1);
        vrandomize <= '0';
        y <= random_vy(0) & "10000000";
        old_oddeven <= not oddeven;
        -- stop the ball from moving.  this is the only situation
        -- where vx should be zero.
```

```
        vx <= (others=>'0');
        vy <= (others=>'0');
      else
        old_oddeven <= oddeven;

        -- test for an edge on oddeven.
        if old_oddeven /= oddeven then
          -- update the x and y logical coordinates.
          if reverse = '1' then
            x <= (not x) + 64;
          else
            x <= x + padded_vx;
          end if;
          if vrandomize = '1' then
            if random_vy(2) = '1' then
              y <= not y;
            end if;
            vx <= random_vx;
            vy <= random_vy(1 downto 0);
          else
            y <= y + padded_vy;
          end if;
          reverse <= '0';
          vrandomize <= '0';
        elsif col_eq_ballx = '1' and row_eq_bally = '1'
          and latched_diff = '1'
        then
          -- we have detected motion at the position of the ball.
          -- now we must test whether the ball is in a bounce segment
          -- (or if vx = "00", which indicates that the ball is
          -- motionless at the center of the screen and needs to
          -- be started moving).
          if vx = "00" or x(7 downto 6) = "11" then
            win1<='0';
            win2<='0';
            -- setting reverse true indicates that the ball should
            -- reverse x direction on the next oddeven edge.
            reverse <= '1';
            -- setting vrandomize true indicates that the velocity
            -- of the ball should be randomized on the next oddeven edge.
            vrandomize <= '1';
          end if;
        end if;
      end if;
    end if;
  end process;
end pongarch;
```

C Bitmap Overlay PROM

C.1 Original Images: `cube.[0-4].xbm`



C.2 Bitmap Processor: `xbm2src.pl`

Perl script to convert a .xbm image to a .src file.

```
sub bin {
  my $z=hex shift;
  $y="";
  for ($i=0;$i<8;$i++) {
$y .= ($z&(1<<$i)) ? "1" : "0";
  }
  return $y;
}

undef $/;
$_ = <>;

/_width\s+(\d+)/;  $w = $1;
/_height\s+(\d+)/; $h = $1;
@a = (/(0x[\da-fA-F][\da-fA-F])/g);
while (@a) {
  for ($x = 0; $x < $w/8; $x++) {
print bin(shift @a), " ";
  }
  print "\n";
}
```

C.3 Character Bitmaps: `cube.[0-4].src`

The four character bitmaps for each frame of the ball image.
`cube.0.src`:

```
00000001 10000000
00000011 11000000
00000110 01100000
00001100 00110000
00011000 00011000
00110000 00001100
01100000 00000110
11000000 00000011
11100000 00000111
10110000 00001101
10011000 00011001
10001100 00110001
10000110 01100001
10000011 11000001
10000001 10000001
10000001 00000001
10000001 00000001
10000001 00000001
10000001 00000001
10000001 00000001
10000001 00000001
10000001 00000001
10000001 00000001
11000001 00000011
01100001 00000110
00110001 00001100
00011001 00011000
00001101 00110000
00000111 01100000
00000011 11000000
00000001 10000000
00000000 00000000
```

`cube.1.src`:

```
00000110 00000000
00000111 10000000
00000100 01110000
00001000 00011100
00001000 00000010
```

```
00001000 00000010
00010000 00000110
00010000 00000110
00100000 00001010
00100000 00001010
01100000 00001010
01111000 00010010
01000111 00010010
01000001 11100010
01000000 00100010
01000000 00100010
01000000 00100010
01000000 00100010
01000000 00100010
01000000 00100010
01000000 00100010
01000000 00100010
01000000 00100100
01000000 00100100
01000000 00101000
01100000 00101000
00111000 00101000
00000111 00110000
00000001 11110000
00000000 00100000
00000000 00000000
00000000 00000000
```

`cube.3.src`:

```
00000000 00000000
00000001 11111000
00111111 00001000
00110000 00001000
00110000 00001000
00110000 00001000
00110000 00001000
00110000 00001000
00110000 00001000
00110000 00000100
00101000 00000100
00101000 00000100
00101000 01111100
00101111 10000100
00101000 00000100
00101000 00000100
```

```
00101000 00000100
00101000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00001000 01111100
00001111 10000000
00000000 00000000
00000000 00000000
00000000 00000000
```

`cube.4.src`:

```
00000000 00000000
00000001 11111000
00111111 00001000
00110000 00001000
00110000 00001000
00110000 00001000
00110000 00001000
00110000 00001000
00110000 00001000
00110000 00000100
00101000 00000100
00101000 00000100
00101000 01111100
00101111 10000100
00101000 00000100
00101000 00000100
00101000 00000100
00101000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
00011000 00000100
```

```
00001000 01111100
00001111 10000000
00000000 00000000
00000000 00000000
00000000 00000000
```

C.4 PROM Datafile Generator: `src2dat.pl`

Combines four character bitmaps into a single PROM .dat file.

```
$header = 0;
if ($ARGV[0] =~ /-h/) { $header = 1; shift; }

@array = ();
while (<>) {
  @bytes = split;
  next if !@bytes;
  @line = ();
  foreach (@bytes) {
@bits = reverse split //;
$value = 0;
for ($i = 0; $i < @bits; $i++) {
  if ($bits[$i] == 1) {
$value |= 2**$i;
  } elsif ($bits[$i] != 0) {
die;
  }
}
push @line, $value;
  }
  push @array, [@line];
}


print "# set_address = 0;\n# BASE = hex;\n" if $header;

$w = @{@array[0]}/2;
$h = @array/2;

for ($y = 255; $y >= 0; $y--) {
  if ($y < $h) {
$yp = $y + $h;
  } elsif ($y >= 256-$h) {
$yp = ($y + $h) % 256;
  } else {
```

```
$yp = -1;
  }
  for ($x = 31; $x >= 0; $x--) {
if ($yp == -1) { print "00 "; next; }
if ($x < $w) {
  $xp = $x + $w;
  printf "%02X ", $array[$yp][$xp];
} elsif ($x >= 32-$w) {
  $xp = ($x + $w) % 32;
  printf "%02X ", $array[$yp][$xp];
} else {
  print "00 ";
}
  }
  print "\n";
}
```