HARVEY MUDD COLLEGE

Pig E-bank

E158 CMOS VLSI Final Project

Becky Glick and Madeleine Ong
4/19/2010

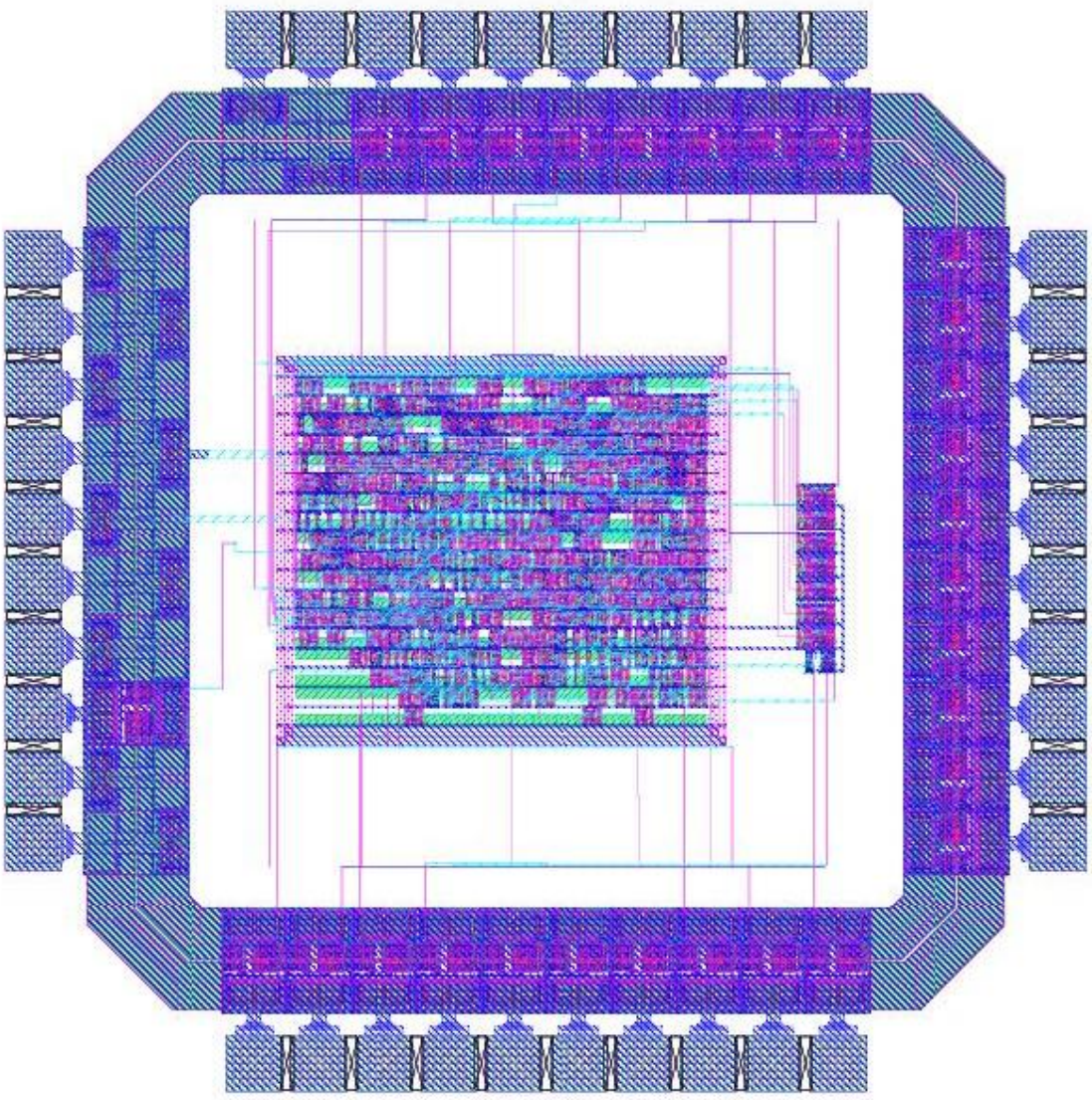


Table of Contents

I. Introduction

The practice of discarding loose change, especially pennies, has become a common practice in the United States. For a convenience motivated, fast passed society, counting coins is often not worth the effort. To address this problem, we designed the chip for an electronic piggy bank, to make counting change effortless and saving money fun.

The Pig-E Bank chip can count 1¢, 5¢, 10¢, 25¢, and $1 coins and display the total value deposited on a 16x2 dot matrix LCD display.

Manufacturing Process

The design process began by dividing the problem into three modules: the 6:1 Mux Module, the Summing Module, and the LCD Module. The FSM diagram can be found in Appendix I.

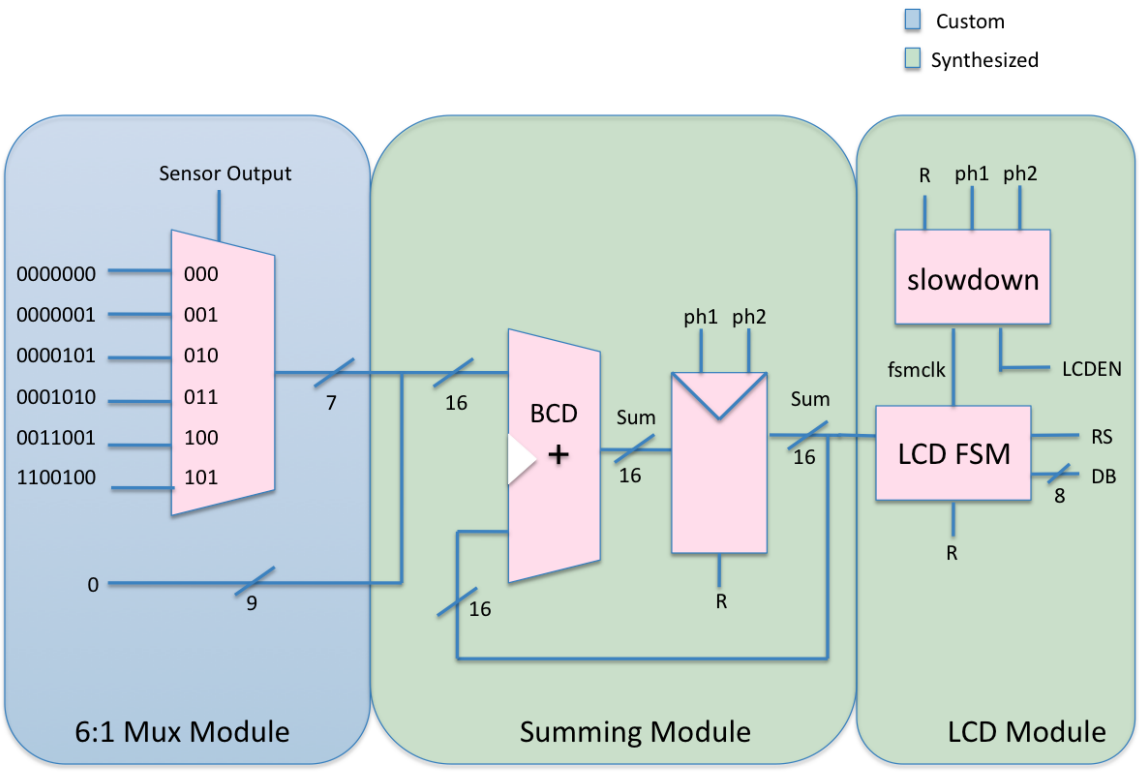


Figure 1: Block diagram showing interactions between the three top-level modules

We first implemented the entire proposed functionality in Verilog and simulated with a self-checking testbench. Then, we created a schematic and layout for our custom block, interfacing with the appropriate portion of synthesized Verilog. Finally, the synthesized block and custom datapath were wired to a padframe to create the final chip.

II. Specifications

The chip required 22 pins of the 40 available pins in the padframe. A padframe schematic made in Cadence can be found in Appendix II.

Pinout

The chip padframe pinout is displayed below in Figure 2 followed by Table 1 listing the I/Os. The unused pins were left blank.

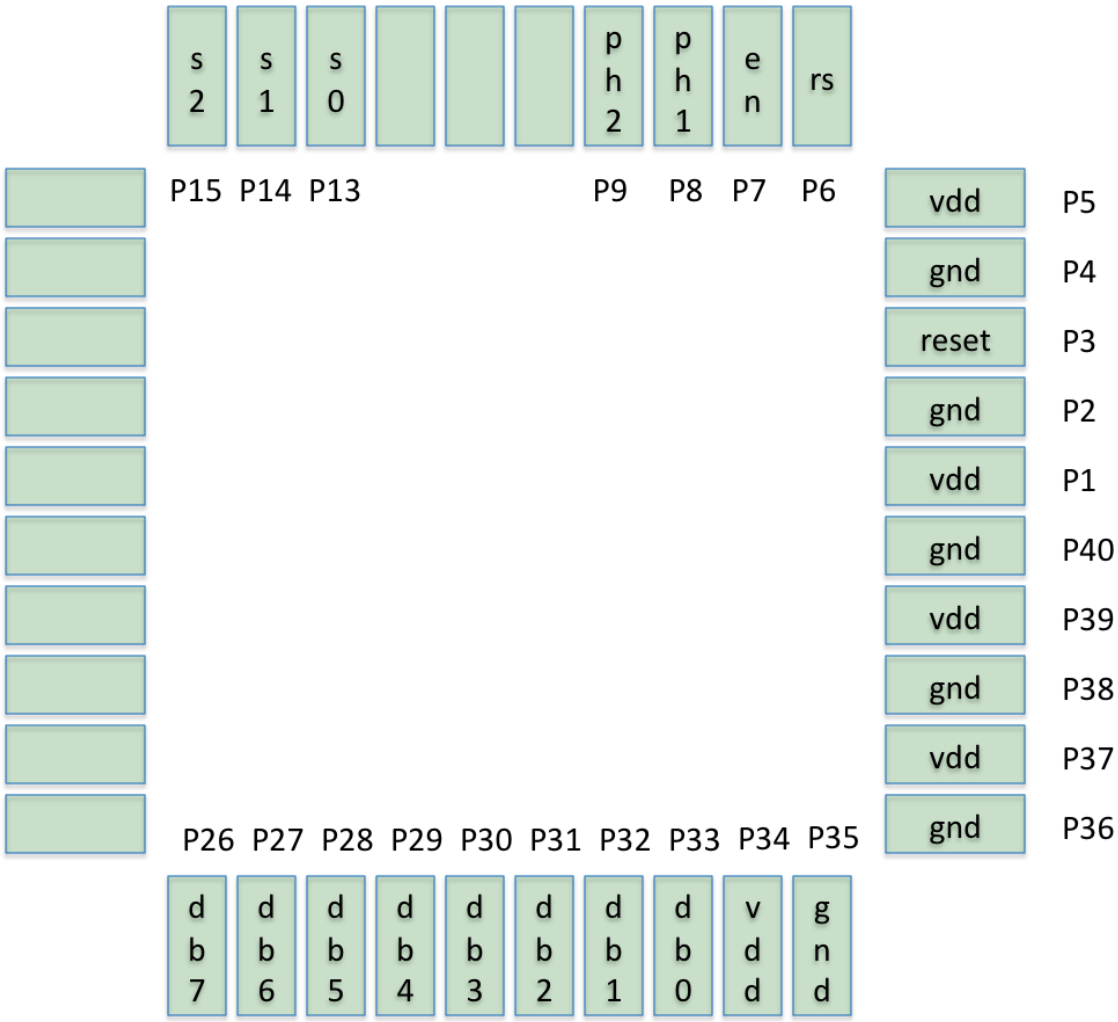


Figure 2: Pinout diagram for padframe of finished chip

Pin Categories	I/O	Number of Required Pins
VDD/GND	Bidirectional	6
LCD Control signals: lcden:1, rs:1, db: 8	Output	10
Coin Counting Sensor	Input	3
Clock ph1, ph2	Input	2
Reset	Input	1
Total:		22

Table 1: Table of inputs, outputs indicating name, direction, and number of required pins

IO Relations

The sensors detect what type of coin has been deposited. Their default output is 3'b0, indicating no coin has been added. After the sensors detect a penny, nickel, dime, quarter or dollar coin, a multiplexer uses the sensor outputs to determine which BCD value the coin represents. Then, this new BCD encoded value is added to a running total of the amount of deposited change, held in a resettable register. Next, this sum passes to a FSM that drives the control signals for the LCD. In the "slowdown" module, ph1 and ph2 create a slower clock signal for the LCD, lcden, allowing enough time to process each control signal. The LCD FSM also runs on this clock to time the LCD controls, rs and db, as it cycles through its states. The LCD FSM is first initialized and then communicates to the LCD what characters to display on the various rows.

III. Floorplan

There was a total available area of 3400λ x 3400 λ for the chip, with a border for the padframe.

Original Floorplan

Before we started building our project, we had to estimate our floorplan size for each component in our chip. This is represented in Figure 3 below.

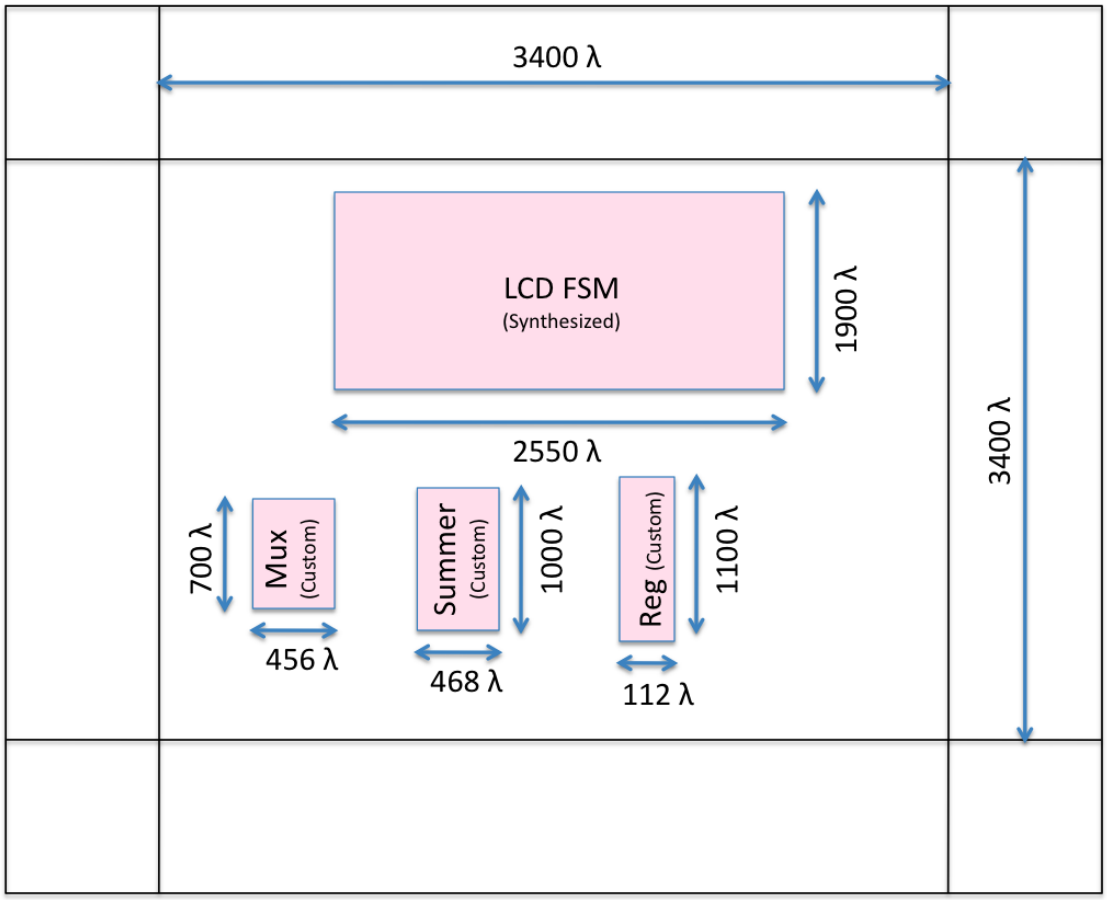


Figure 3: Estimated floorplan

There was a total expected occupied area of approximately 5755400 λ^2.

Final Floorplan

After the chip was constructed and verified through tool checks and simulation, the final floor plan was more condensed than our original floorplan. Figure 4 is a diagram of the final floorplan.

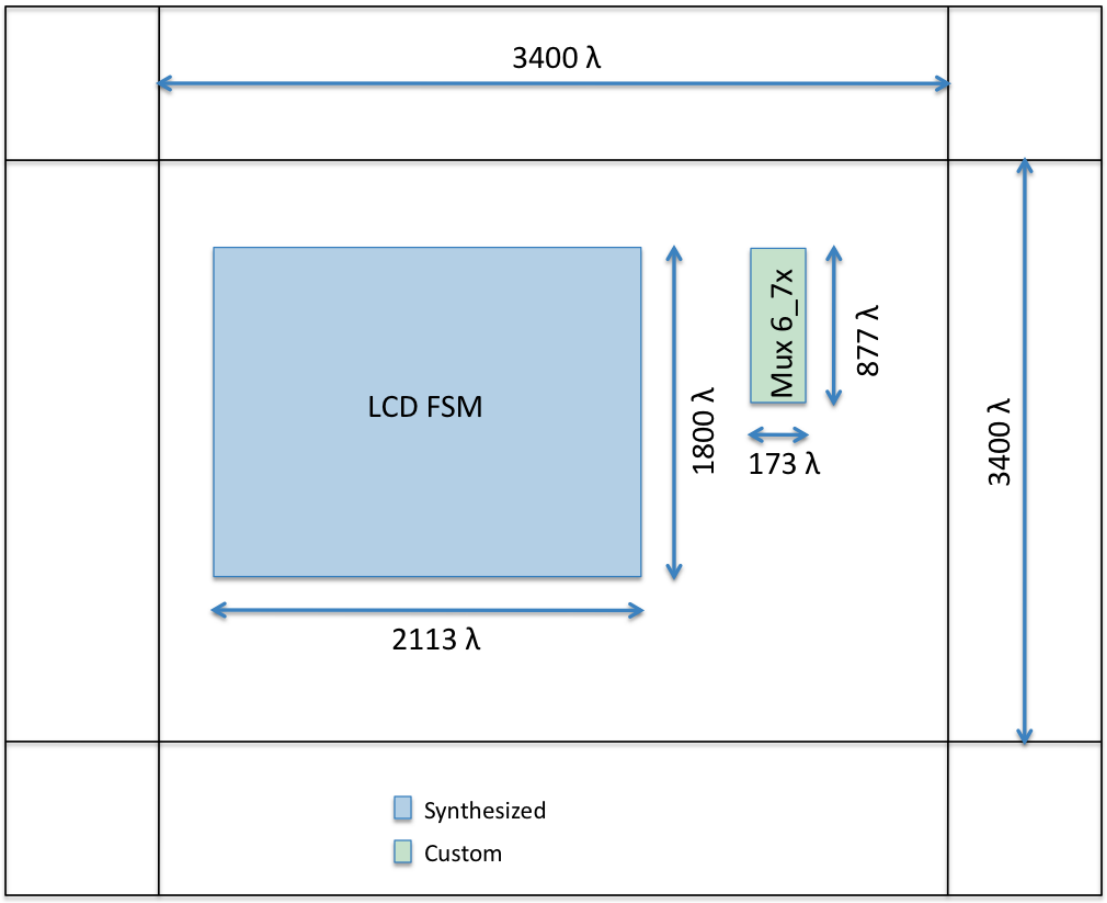


Figure 4: Final floorplan design.

There is a total occupied area of approximately 3955121 λ^2. This shows a difference of 1800279 λ^2 occupied space between the original and final floorplans.

Floorplan Discrepancies

The main change in our floorplan was due to our decision to synthesize more of the logic. Originally, we planned to lay out the smaller out by hand and to group them relatively close together. However, due to the amount of time it would take to layout a mux6_7x, we decided the BCD adders and various registers would be synthesized along with the LCD FSM. This led to us having a larger synthesized block, and less custom logic.

The FSM, displayed in full in Appendix I, was much smaller than predicted. The original estimations were based on the amount of logic/state in a MIPS processor. The MIPS processor was much more complicated than the LCD FSM which only needed to output a few signals and had very little intrastate logic. Therefore, despite having the same number of states in the FSM, the occupied space was significantly overestimated.

Similarly, the original multiplexer size estimate was based off the assumption we would use multiple mux2's to create the mux6. Instead, we chose the more efficient design of building it on the transistor level, which made the layout more challenging.

Custom Datapath

The final custom datapath only contained the mux6_7x, consisting of an inverter-buffer and seven mux6_1x's. The inverter buffer provides the complements of the sensors for the multiplexers, with the inputs either fixed at VDD or GND, representing the BCD value of the different coins. The final schematics and layout figures for the datapath components are located in Appendix III.

IV. Verification

The Verilog code in Appendix IV represents the entire project's proof of concept. It simulated successfully using the self-checking testbench in Appendix V. When testing the synthesized logic, the only modification made was the replacement of 'sensors' with what the 16-bit output of the datapath would be., thereby bypassing the written mux6 module.

All of the submodules passed their appropriate self-checking testbench. Other verification requirements are displayed below in Table 2.

Facet	Schematic/Layout	DRC	LVS
Mux6_1x	Schematic Layout	Pass Pass	Pass
Mux6_7x	Schematic Layout	Pass Pass	Pass
Synthesized Verilog	Schematic Layout	Pass Pass	Pass
Core	Schematic Layout	Pass Pass	Pass
Padframe	Schematic Layout	Pass Pass	Pass
Chip	Schematic Layout	Pass Pass	Pass
CIF - loaded from chip CIFout	Layout	Pass	Pass

Table 2: Chip Validation

The self-checking test bench does not check the functionality of the BCD adders. The results were checked visually by loading the testbench into NC Sim and observing the output waveform at the end of the testbench. All four BCD adders demonstrated correct addition and overflow functions.

V. Post fabrication Test Plan

If the chip was chosen for fabrication, the first step would be to first verify the FSM functionality in hardware, not just in simulation. The code was written specifically for a Xiamen Ocular GDM1602K LCD. After the FSM was proven to work when programmed on an FPGA, we would be to find compatible sensors and add extra combinational logic to put the sensor outputs in the binary form needed for the multiplexer.

Once the chip was fabricated, the same test that was done in hardware through Xilinx, would be repeated, but with the chip in place of the FPGA. The sensors would be hooked up to test whether or not the coins were accurately detected and counted. If the counting is accurate and the LCD displayed the correct message, this would indicate the chip was fully functioning.

VI. Design Time

Facet	Version	Time(h)
Structural Verilog (Proof of Concept)		5
Self-checking testbench		3
Mux6_1x	Schematic Layout	4 14
Mux6_7x	Schematic Layout	1 1
Synthesized Logic	Schematic Layout	3 2
Padframe	Schematic Layout	0.5 0.5
Chip	Schematic Layout	1 1
CIF	Layout	0.5
Subtotal		36.6

Other Items	Time(h)
Proposal	5
Estimated Floorplan	3
Final Report	4
Subtotal	12

Total	**48.6**

VI. File Locations

All of these files are located on the chips server in the Harvey Mudd College Engineering Department.

• Verilog code
/home/rglick/IC_CAD/finalsynth/synthpig.sv

• Test vectors
/home/rglick/IC_CAD/finalsynth/pig.tv

• Synthesis results
/home/rglick/IC_CAD/finalsynth /chip_run1

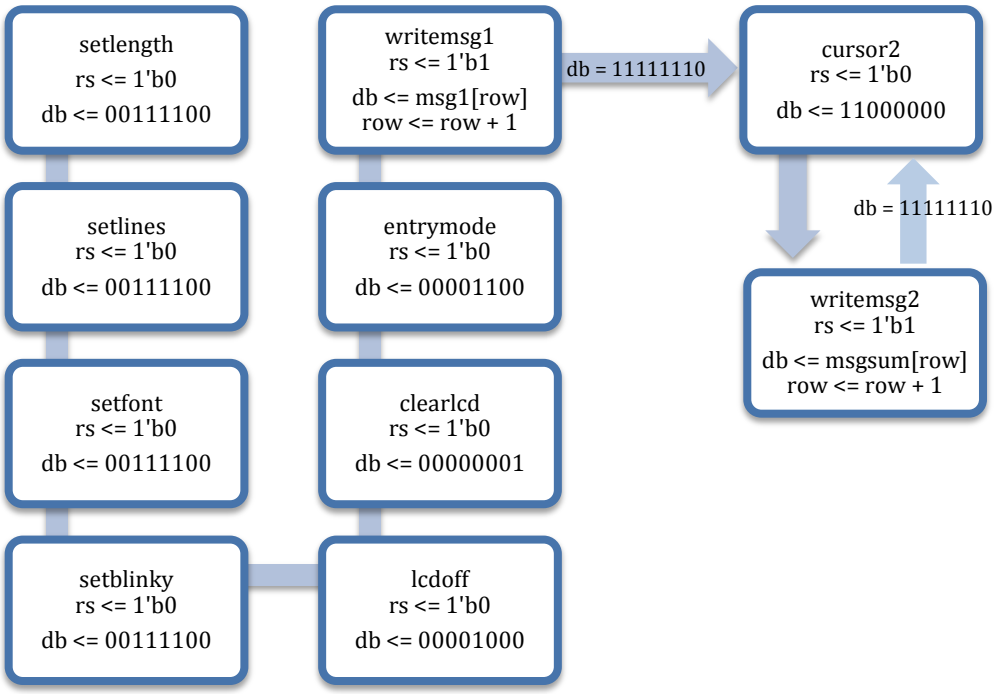
• All Cadence libraries
/home/rglick/IC_CAD/cadence/finalproj_morg
• CIF
/home/rglick/IC_CAD/cadence/supperchippiggy_cifin

• PDF chip plot
/home/rglick/chip.pdf

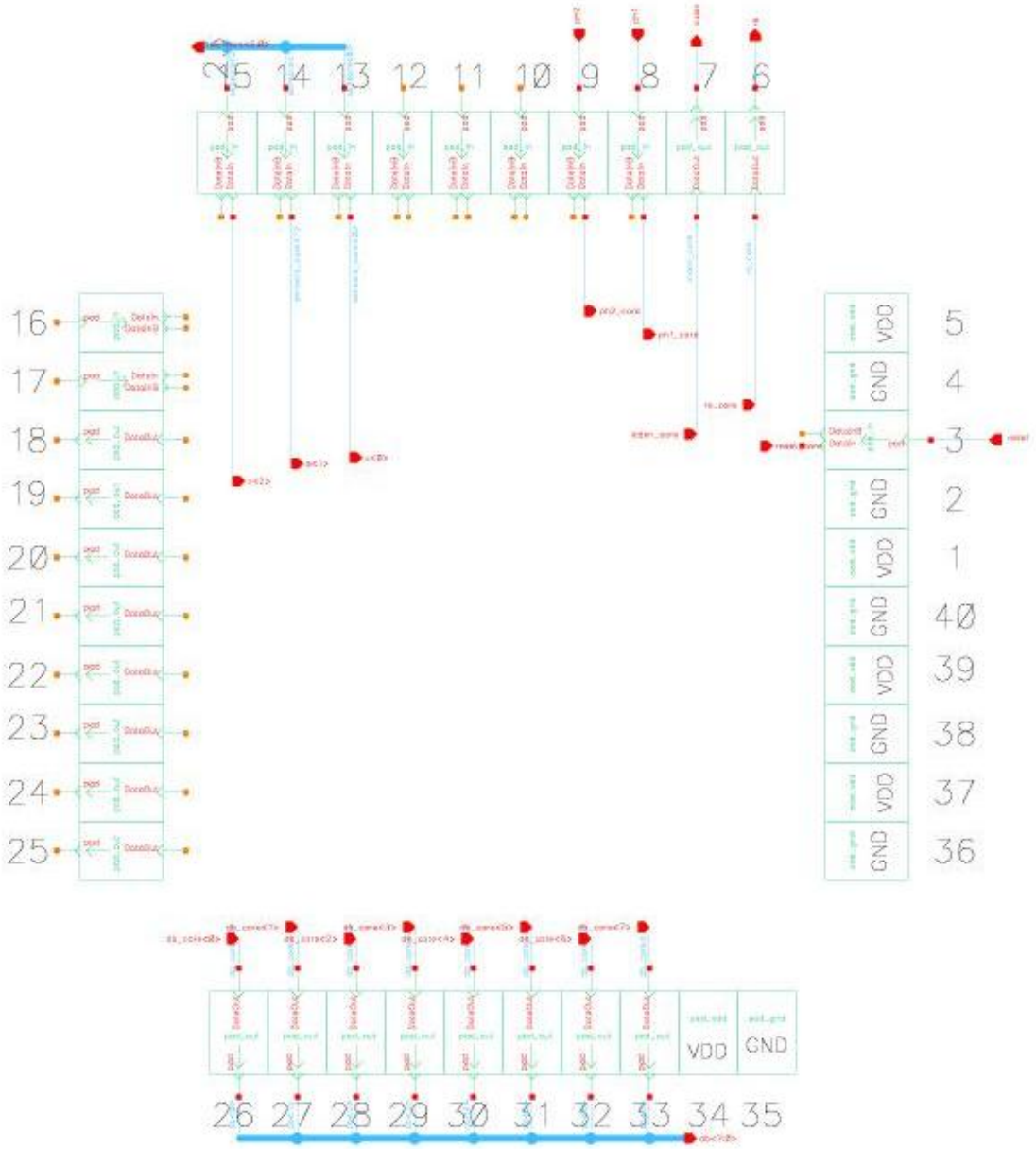
• PDF of final report
/home/rglick/ProjTwoMorg.pdf

VII. Appendices

Appendix I: FSM controlling the LCD



Appendix II: Cadence Padframe Schematic



Appendix III: Schematics and Layouts for the Custom Datapath

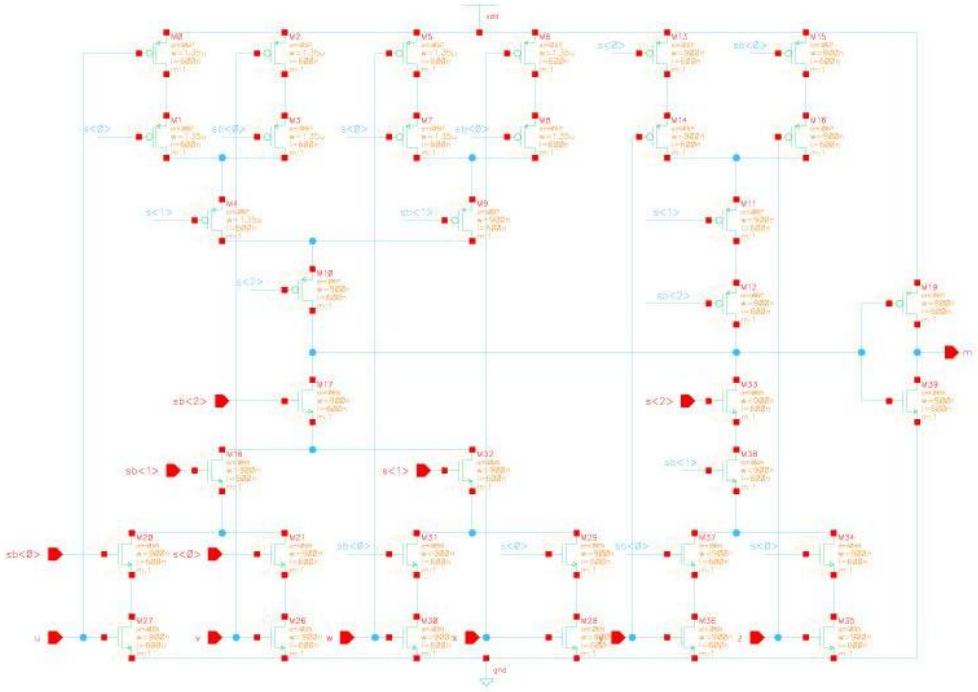


Figure 1: mux6_dp_1x cmos_sch – CMOS Schematic of 6:1 multiplexer

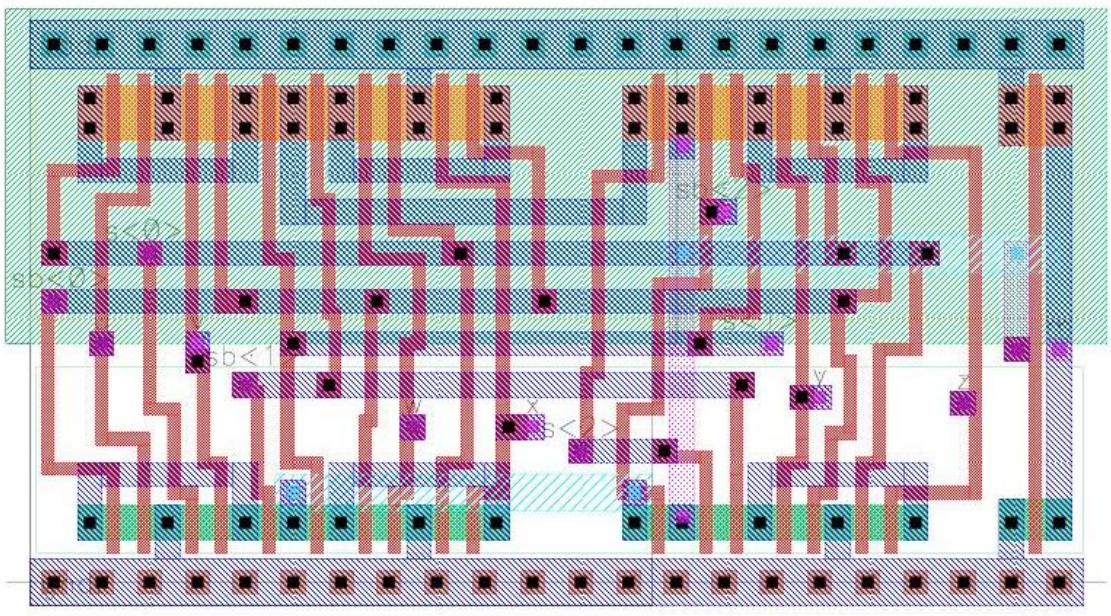


Figure 2: mux6_dp_1x layout - Layout of 6:1 multiplexer

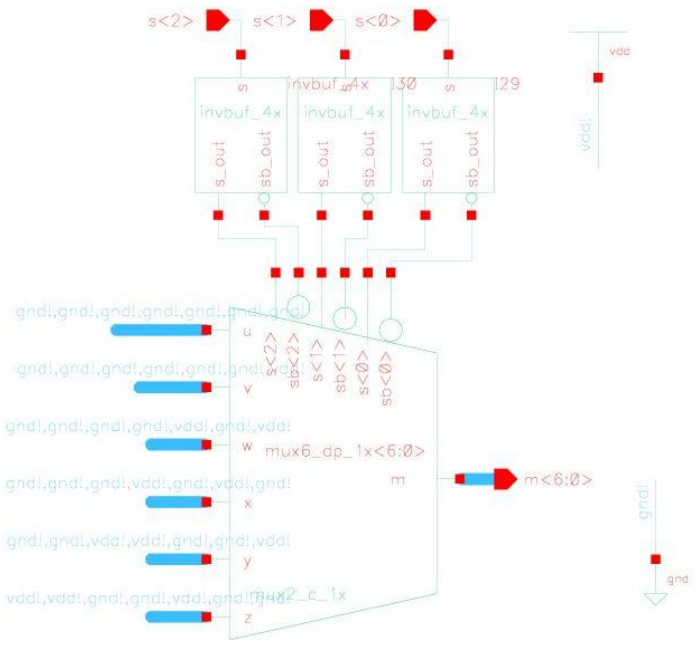


Figure 3: mux6_dp_7x schematic - Schematic of 6:1 multiplexer

Figure 4: mux6_dp_7x layout – Layout of datapath

Appendix IV: Verilog Code for Proof of Concept

```
// Location on Chips server at Harvey Mudd College
// /home/rglick/IC_CAD/finalsynth/wholepig.sv
// Authors: Becky Glick & Madeleine Ong
// Function: Money Counter
// VLSI Final Project 2010

module pig(
    input ph1,
    input ph2,
    input [2:0] sensors, // assumption: logic for sensors can be done off chip
    input reset,
    output lcden,
    output rs,
    output [7:0] db
    );

       wire [3:0] one;
       wire [3:0] ten;
       wire [3:0] hun;
       wire [3:0] tho;
       wire overflow;
       wire cout0;
       wire cout1;
       wire cout2;

       wire [15:0]coin;
       wire [7:0] thousands;
       wire [7:0] hundreds;
       wire [7:0] tens;
       wire [7:0] ones;
       wire fsmclk;


      slowdown slow(ph1, ph2, reset, lcden, fsmclk);

      mux_6to1 sense(ph1, ph2, reset, sensors, coin);

      bcd_4bit oneadd(ph1, ph2, reset, coin[3:0], 1'b0, cout0, one);
      bcd_4bit tenadd(ph1, ph2, reset, coin[7:4], cout0, cout1, ten);
      bcd_4bit hunadd(ph1, ph2, reset, coin[11:8], cout1, cout2, hun);
      bcd_4bit thoadd(ph1, ph2, reset, coin[15:12], cout2, overflow, tho);

      lcddecoder tend(tho, thousands); // ten dollar
      lcddecoder oned(hun, hundreds); // one dollar
      lcddecoder tenc(ten, tens); // ten cent
      lcddecoder onec(one, ones); // one cent

      lcd_fsm lcd(ph1,ph2,fsmclk,reset,ones,tens,hundreds,thousands,overflow, rs, db);


endmodule


// Function: Makes the slow clock for the lcden and fsm
module slowdown(
      input ph1,
      input ph2,
      input reset,
      output slowclk,
      output blipclk);

      wire [18:0] counter;
      wire [18:0] newcount;

      flopr #19  count(ph1, ph2, reset, newcount, counter);

      assign newcount = counter + 1'b1;
      assign slowclk = newcount[18];
```

```
      assign blipclk = (newcount == 19'b1000000000000000000);

endmodule

// Function: Placeholder for the 6to1 mux
module mux_6to1(
       input ph1,
       input ph2,
       input reset,
       input [2:0] sensors,
       output [15:0] coin_bcd);

       wire [15:0] coin;

       parameter penny  = 16'b0000_0000_0001;
       parameter nickel       = 16'b0000_0000_0101;
       parameter dime   = 16'b0000_0001_0000;
       parameter quarter      = 16'b0000_0010_0101;
       parameter dollar = 16'b0001_0000_0000;

    assign coin = sensors[2] ?
                        sensors[0] ? dollar : quarter    :
                        sensors[1] ?
                              sensors[0] ? dime    : nickel :
                              sensors[0] ? penny   : 16'b0  ;

       flopr #16  cointype(ph1, ph2, reset, coin, coin_bcd);

endmodule

// Function: binary coded decimal adder
module bcd_4bit(
            input ph1,
            input ph2,
            input reset,
            input [3:0] a,
            input cin,
            output cout,
            output [3:0] y
            );

      wire [3:0] newsum;
      wire newcout;
      reg  [4:0] binsum;
      reg  [4:0] correctsum;



      always @(*)
            begin
            binsum <= y + a + cin;
            if (binsum > 9)
               correctsum <= binsum + 4'b0110; // correction factor
            else
                  correctsum <= binsum;
            end

      assign newsum = correctsum[3:0];
      assign newcout = correctsum[4];


      // flops for overflow and sum
      flopr #4  bcd1(ph1, ph2, reset, newsum, y);
      flopr #1  bcd2(ph1, ph2, reset, newcout, cout);


endmodule

// Function: Decodes the binary value into character for LCD
module lcddecoder( input [3:0] thing,
              output reg [7:0] char);
```

```
      always @(*)
            case (thing)
            4'b0000:    char = 8'b0011_0000;
            4'b0001:    char = 8'b0011_0001;
            4'b0010:    char = 8'b0011_0010;
            4'b0011:    char = 8'b0011_0011;
            4'b0100:    char = 8'b0011_0100;
            4'b0101:    char = 8'b0011_0101;
            4'b0110:    char = 8'b0011_0110;
            4'b0111:    char = 8'b0011_0111;
            4'b1000:    char = 8'b0011_1000;
            4'b1001:    char = 8'b0011_1001;
            default:    char = 8'b1111_1110;
            endcase

endmodule

// Function: Initializes maintains the LCD screen display
module lcd_fsm(
       input ph1,
       input ph2,
       input en,
       input reset,
       input [7:0] ones,
       input [7:0] tens,
       input [7:0] hundreds,
       input [7:0] thousands,
       input overflow,
       output reg rs,
       output reg [7:0] db  );

      wire [3:0] state;
      reg [3:0] nextstate;
      wire [3:0] row;
      reg [3:0] newrow;

      wire[15:0] msg1 [7:0];


      wire[7:0] msgsum [7:0];


      // Message Definition
      assign msg1[0] = 8'b0101_0100; // T
      assign msg1[1] = 8'b0110_1111; // o
      assign msg1[2] = 8'b0111_0100; // t
      assign msg1[3] = 8'b0110_0001; // a
      assign msg1[4] = 8'b0110_1100; // l
      assign msg1[5] = 8'b1111_1110; // space
      assign msg1[6] = 8'b1111_1110; // space

      assign msgsum[0] = 8'b0010_0100; // dollar
      assign msgsum[1] = thousands;
      assign msgsum[2] = hundreds;
      assign msgsum[3] = 8'b0010_1110; // decimal point
      assign msgsum[4] = tens;
      assign msgsum[5] = ones;
      assign msgsum[6] = 8'b1111_1110; // space
      assign msgsum[7] = 8'b1111_1110; // space

      // various states
      parameter setlength    = 4'b0001;
      parameter setlines     = 4'b0010;
      parameter setfont      = 4'b0011;
      parameter setblinky    = 4'b0100;
      parameter lcdoff       = 4'b0101;
      parameter clearlcd     = 4'b0110;
      parameter entrymode    = 4'b0111;
      parameter writemsg1    = 4'b1001;
      parameter cursor2      = 4'b1010;
```

```
parameter writemsg2    = 4'b1011;

// state register
flopenr #4 statereg(ph1, ph2, reset, en, nextstate, state);
flopenr   #4 rowreg(ph1, ph2, reset, en, newrow, row);

// next state logic
always@(*)
      case(state)
            setlength:
                  begin
                  rs <= 1'b0;
                  db <= 8'b00111100; // set to 8 bits
                  nextstate <= setlines;
                  end
            setlines:
                  begin
                  rs <= 1'b0;
                  db <= 8'b00111100; // set 2 lines
                  nextstate <= setfont;
                  end
            setfont:
                  begin
                  rs <= 1'b0;
                  db <= 8'b00111100; // set font size
                  nextstate <= setblinky;
                  end
            setblinky:
                  begin
                  rs <= 1'b0;
                  db <= 8'b00111100; // invisible cursor
                  nextstate <= lcdoff;
                  end
            lcdoff:
                  begin
                  rs <= 1'b0;
                  db <= 8'b00001000; // turn off lcd
                  nextstate <= clearlcd;
                  end
            clearlcd:
                  begin
                  rs <= 1'b0;
                  db <= 8'b00000001; // clear lcd
                  nextstate <= entrymode;
                  end
            entrymode:
                  begin
                  rs <= 1'b0;
                  db <= 8'b00001100; // ready to write!
                  nextstate <= writemsg1;
                  end
            writemsg1:
                  begin
                        rs <= 1'b1;
                        db <= msg1[row];   // pull char from matrix
                        if (db == 8'b1111_1110)
                              begin
                              nextstate <= cursor2;
                              newrow <= 4'b0;
                              end
                        else
                              begin
                              nextstate <= writemsg1;
                              newrow <= row + 1'b1;
                              end
                  end
            cursor2:
                  begin
                  rs <= 1'b0;
                  db <= 8'b11000000; // set to 2nd row first square
                  nextstate <= writemsg2;
```

```
                  end
            writemsg2:
                  begin
                        rs <= 1'b1;
                        db <= msgsum[row];
                        if (db == 8'b1111_1110)
                              begin
                              nextstate <= cursor2;
                              newrow <= 4'b0;
                              end
                        else
                              begin
                              nextstate <= writemsg2;
                              newrow <= row + 1'b1;
                              end
                  end
            default:
                  begin
                  nextstate <= 4'b0001;
                  newrow <= 4'b0;
                  end

      endcase

endmodule

// Provided by: Professor D. Harris
module flop #(parameter WIDTH = 8)
             (input              ph1, ph2,
              input[WIDTH-1:0] d,
              output  [WIDTH-1:0] q);

  wire [WIDTH-1:0] mid;

  latch #(WIDTH) master(ph2, d, mid);
  latch #(WIDTH) slave(ph1, mid, q);
endmodule

// Modified from Professor D. Harris's flopenr
module flopr #(parameter WIDTH = 8)
               (input  ph1, ph2, r,
                input  [WIDTH-1:0] d,
                output [WIDTH-1:0] q);

  wire [WIDTH-1:0] d2, resetval;

  assign resetval = 0;

  mux2 #(WIDTH) enmux(d, resetval, r, d2);
  flop #(WIDTH) f(ph1, ph2, d2, q);
endmodule

// Provided by: Professor D. Harris
module flopenr #(parameter WIDTH = 8)
                (input               ph1, ph2, reset, en,
                 input  [WIDTH-1:0] d,
                 output [WIDTH-1:0] q);

  wire [WIDTH-1:0] d2, resetval;

  assign resetval = 0;

  mux3 #(WIDTH) enrmux(q, d, resetval, {reset, en}, d2);
  flop #(WIDTH) f(ph1, ph2, d2, q);
endmodule

// Provided by: Professor D. Harris
module latch #(parameter WIDTH = 8)
              (input              ph,
               input   [WIDTH-1:0] d,
               output reg [WIDTH-1:0] q);
```

```
  always@(ph or d)
    if (ph) q <= d;

endmodule

// Provided by: Professor D. Harris
module mux2 #(parameter WIDTH = 8)
             (input [WIDTH-1:0] d0, d1,
              input             s,
              output [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

// Provided by: Professor D. Harris
module mux3 #(parameter WIDTH = 8)
             (input [WIDTH-1:0] d0, d1, d2,
              input [1:0]       s,
              output reg [WIDTH-1:0] y);

  always@(*)
    casez (s)
      2'b00: y = d0;
      2'b01: y = d1;
      2'b1?: y = d2;
    endcase
endmodule
```

Appendix V: Self-Checking Testbench

```
// Location on Chips server at Harvey Mudd College
// /home/rglick/IC_CAD/finalsynth/wholepigtest.sv
// Authors: Becky Glick & Madeleine Ong
// Function: Self-Checking Testbench

// Set delay unit to 1 ns and simulation precision to 0.1 ns (100 ps)
`timescale 1ns / 100ps

// testbench for testing
module testbench #(parameter WIDTH = 8, REGBITS = 3)();

  logic ph1, ph2, reset, rs;
  logic [WIDTH-1:0] db;
  logic [2:0] sensors;
  logic lcden;
  logic pixel0;
  logic pixel1;
  logic pixelT;

  // DUT
  chip rawr( db, lcden, rs, ph1, ph2, reset, sensors );

  // initialize test
  Initial begin
            pixel0 <= 0;
      pixel1  <= 0;
      pixelT  <= 0;
            reset   <= 1;
            sensors <= 0;
            # 22;
            reset <= 0;
            sensors[0] <= 1;  // add a penny!
            # 20;
            sensors[0] <= 0;
    end

  // generate clock to sequence tests
  always begin
     ph1 <= 0; ph2 <= 0; #1;
     ph1 <= 1; # 4;
     ph1 <= 0; #1;
     ph2 <= 1; # 4;
    end

  // checking for various things :)
  always @(posedge lcden) begin
      if(rs == 1'b1 & pixel0 == 1'b0) begin // displayed the first chacter!
      assert(db[6:0] == 8'b01010100)      // indicates it has been initialized correctly
      pixel0 = 1'b1;
      $display("Displayed T successfully");
      end
      if(db[7]==1'b1 & rs == 1'b0 & pixel1 == 1'b0 & pixel0 ==1'b1) begin // at 2nd row
        assert(db[6:0] == 7'b1000000) // indicates it finished printing all msg1
        pixel1 = 1'b1;
        $display("Cursor at start of 2nd row successfully");
      end
      if(rs==1'b1 & pixel1 == 1'b1 & pixelT == 1'b0 & pixel0 == 1'b1) begin // printed $
      assert(db[7:0] == 8'b00100100)      // started displaying the sum amount
        $display("Displayed $ successfully");
      pixelT = 1'b1;
      end
      if(pixel0 == 1'b1 & pixel1 == 1'b1 & pixelT == 1'b1) begin // smooth sailing!
       $display("Test was SUPERBLY Successful :)");
       $finish;
      end
  end

endmodule
```