Harvey Mudd Micro Machine

A Field Programmable Logic Array

Designed by
Andrew Macrae and
William Koven
For E158

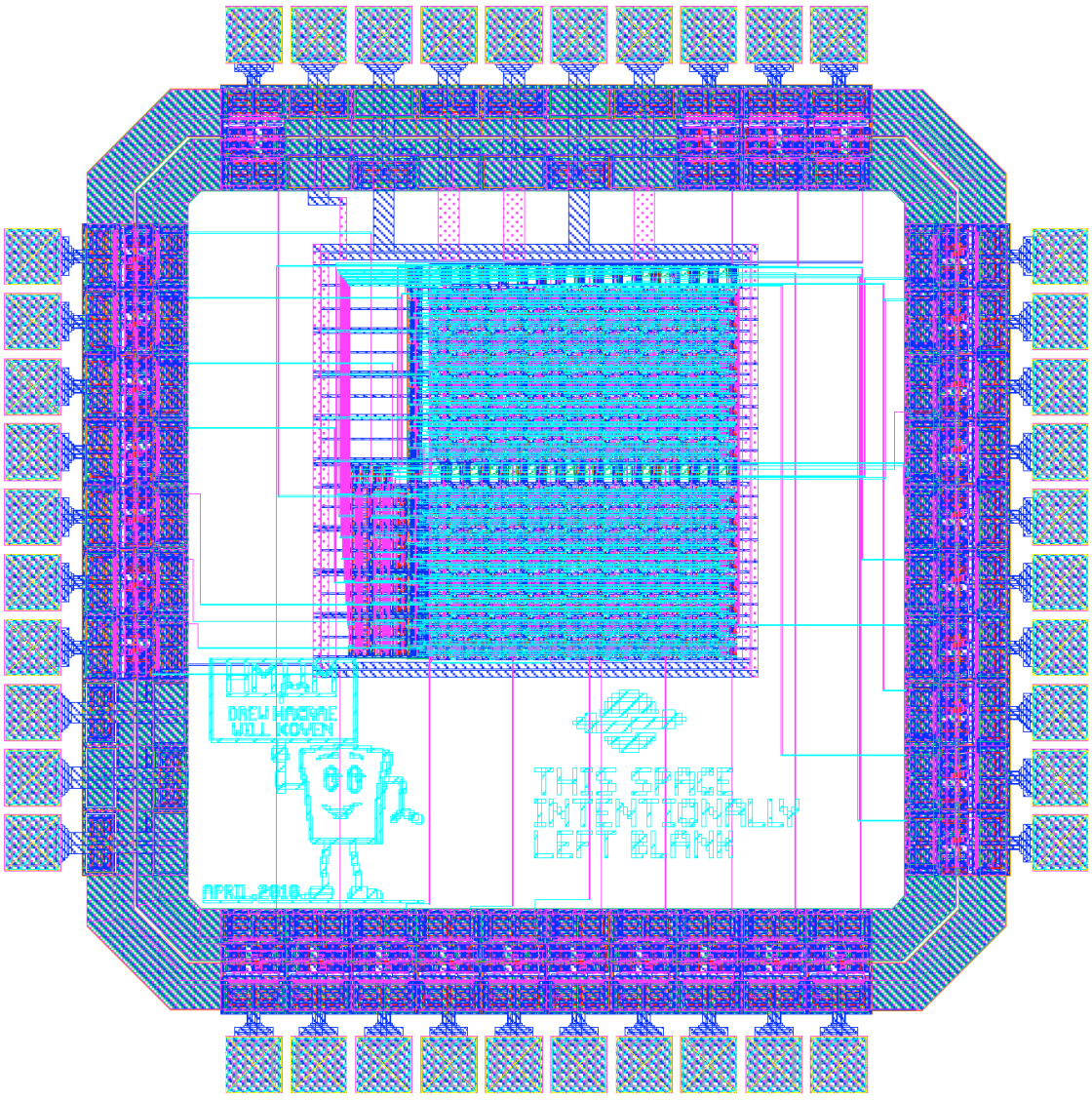


Table of contents

Table of Figures

Introduction

In soul of a new machine, Tracy Kidder describes a the Eagle minicomputer built with programmable logic chips. The then novel PLAs provided designers with reconfigurable sea-of-gates logic chips that allow users to configure a chip to execute sum of products computations efficiently and quickly. PLAs allowed designers to accelerate certain parts of the design cycle and reduce design costs. FPGAs and CPLDs have replaced PLAs in modern electronics markets because of their increased utility and decreasing costs. Simply speaking, a PLA is “a ROM with a reconfigurable decoder”.

We designed a small field programmable PLA that uses a shift register to allows the user to serially configure and reconfigure the PLA. The user has the option to feed some of the outputs back into the PLA in order to build simple state machines. With a 520-bit configuration file, the chip can be configured to act as a 7-

segment decoder, a 4-bit counter, a 7-bit shift register and even the adventure game from Harvey Mudd College's E85 lab 3.

Specifications

Configuration

The chip will store 520 configuration bits after they are scanned in. Of the 520 configuration bits, 256 of these will be used to allow 8 inputs and their complements to generate 16 products. Another 256 configuration bits will be used to let these 16 products generate 16 outputs. The last 8 bits will be used to optionally feed the outputs back in as inputs after being registered by a bank of 8 flops.

Feedback Flops

Unlike a traditional PLA, which is designed to implement only combinational logic, our PLA will allow any of the 8 inputs to be bypassed by a flop controlling a mux so that state can be stored and fed back to the system. This allows our PLA to act as a small reconfigurable FSM as well as implement combinational logic.

Pin	signal	Dir
1	configPh1	Input
2	configPh2	Input
3	configD	Input
5,8	VDD	Supply
4,6,7,9	GND	Supply
10-17	DIN[7:0]	Input
21-36	DOUT[15:0]	Output
37	logicPh1	Input
38	logicPh2	Input
39	configQ	Output
40	reset	Input

Inputs and Outputs

Two two-phase clocks will drive all the sequential logic on chip that will be implemented synchronously. The configuration clock, configPh1 and configPh2, scans in configuration bits on configD, while the logic clock controls the optional sequential logic on the PLA. DIN provides data on the 8 inputs to the AND array while DOUT provides the outputs from the OR array. Reset synchronously resets only the feedback flops allowing the PLA to be reset without loosing configuration data. Because all the configuration bits must be scanned in anyway, they will be a large shift register and will not be resettable.

Theory of Operation

The combinational logic on this PLA accepts 8 inputs, including feedback from previous state of the chip's outputs. Setting the feedback bits chooses whether inputs are read from the current value at DIN or from the previous value at DOUT. Only bits zero through seven of DOUT can be read as feedback and all must be registered between stages and read into the equal valued portion of DIN. The shift register is made up of configurable pulldown cells that will allow a high value on their input "A" to pull down their outputs "Y" depending on their current registered value. These are configured into rows, and the rows are stacked into blocks to form both the andblock and the orblock. The andblock is configured to select 16

configurable minterms of the inputs and their complements. These minterms act as 8 input AND gates with each input configurable to select an input from DIN, select the complement of an input from DIN, or select a 1. The orblock acts as a set of OR gates that can select up to 16 inputs and output a 1 if any of them is high. The andblock and orblock are functionally very similar which simplifies layout. This requires a crooked scanchain through the orblock as compared to the scanchain we initially designed, but the changes to the schematics, the Verilog and the assembler made the layout easier and saved a reasonable amount of time.

Process

The system was designed for a 0.6µm AMI to process to be fabricated by MOSIS. The process has an FO4 delay of about 400ps, 3 metal layers with a metal 1 and metal 2 pitch of 6λ, a metal 3 pitch of 10λ and one polysilicate layer that allows for minimum transistors that are 2λ by 3λ. The system is designed to fit on a "tiny chip" which is 1.5mm on a side and allows for about 1mm within the pad frame.

Floorplan

Figure 1 Original floorplan

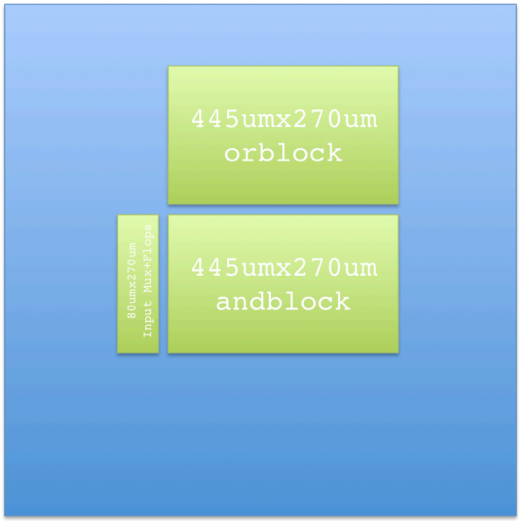


Figure 2 Revised floorplan

Here is a comparison of the originally proposed floorplan with one edited to reflect a more realistic sizing of the blocks. The orblock in the second floorplan controls 16 outputs rather than the specified 8, and the both large blocks are built with a grid of custom gates that have been designed to reduce the size. The feedback cells and the rest of the control circuitry are incorporated into the input mux+flops.

Pinout

If fabbed, the chip will be packaged in a 40 pin DIP with the following pinout

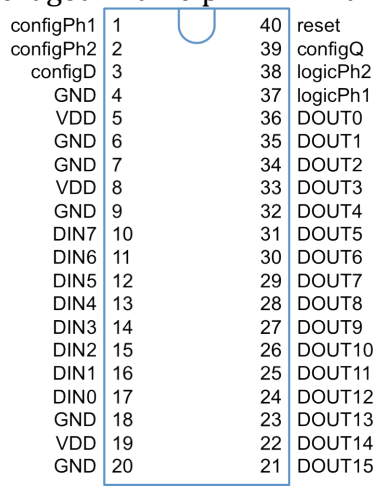


Figure 3 Pinout

Verification

- The Verilog passes testbench for all sample files
- The schematics pass testbench for all sample files.
- The layout passes DRC and LVS
- The CIF loads correctly and the loaded layouts pass DRC and LVS

HSPICE Verification

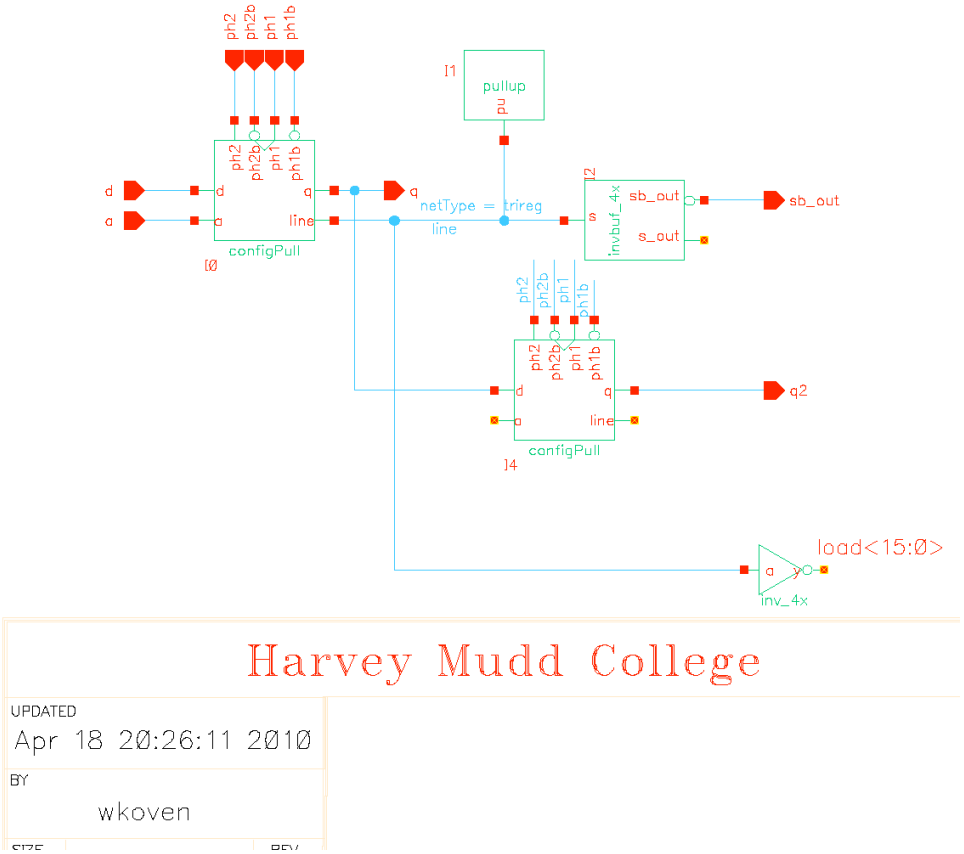


Figure 4 A schematic of the circuit simulated with HSPICE

The configPull cell is the basic building block of both the orblock and the andblock of the PLA. Therefore the configPull must function properly or the PLA will not work. A configPull consists of a register, a separate logic input, and two series n-mos transistors that pull down a bitline when both the logic input and the register output are high. Multiple configPulls are connected together to create a scan chain with the output of the register of one configPull going to the input of the register in the next configPull. The andblock and orblock are made out of a dense array of configPulls where the scanchain runs horizontally across the rows and the products run vertically through the columns. Each column of a block is a pseudo n-mos nor. At the top of each column of configPulls in a block is a small pull-up resistor that keeps pulls the bitline high when none of the configPulls in the column are on. Therefore the critical functions tested were that a configPull will properly register its values, the configPull must be able to pull a bitline low, the pull-up p-mos must be able to pull a bitline high, and the configPulls must shift values properly when connected together in a scanchain.

The schematic in Figure 4 shows two configPulls connected together like they would be in a scanchain with one of the configPulls also connected to a bitline. There are 16 inv_4x also connected to the bitline to act as a load bigger than any

actual load present on a single bitline in the PLA. The layout also has a bitline that is the same vertical height as a bitline in the andblock.

Figure 5 shows the waveforms of an HSPICE simulation of the layout of the test cell, while the testbench used is available in appendix C. Note that everything works properly with a clock period of 150ns. The series n-mos in the configPull can pull down the bitline even with the large load and the pull-up p-mos. The pull-up p-mos is slow to pull up the bitline, but the chip has very few logic levels that portion of the chip will limit operation to 6Mhz. Finally, the values scanned into the first configPull shift properly through the second configPull. (Note that the high value on input d at about 400ns was intentionally placed so as to not be captured by the first configPull register.)

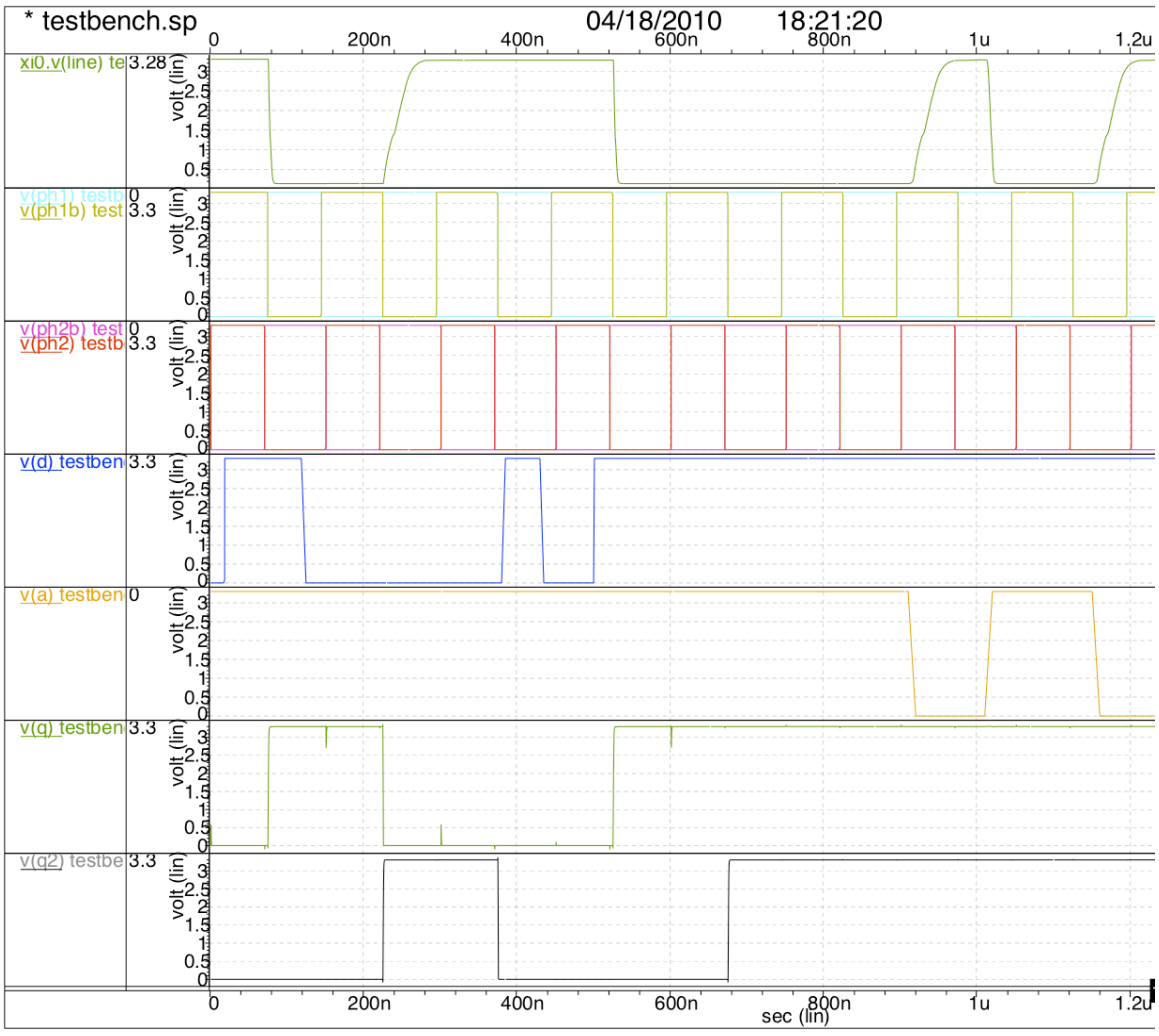


Figure 5 The input and output waveforms as simulated by HSPICE

Post Fabrication Testplan

If we are allowed to fabricate the chip we will test the chip by implementing several functions including but not limited to

- a 3-bit decoder
- an eight input AND and eight input NOR gate
- a 7-segment display decoder
- a resettable 4-bit counter
- a 7-bit shift register

We will then produce a demo system that uses a PIC to load several configurations into the PLA on demand and allow users to interact with the inputs using buttons and to read the outputs with LEDs and two seven segment displays.

Demo System

The PIC will provide a clock to the PLA and will scan in three interesting configuration files, and will connect the appropriate buttons and LEDs for the demo. The PIC will ground the LEDs needed to demonstrate the current mode allowing them to light. The PIC will also connect the appropriate switches allowing them to control the PLA. A reset switch will allow the logic of the PLA to be reset without resetting the hardware. The following files will be included so that a user can interact with the chip running in the following modes. The clock will also be displayed and will run at 1 Hz or so to allow the user to ensure synchronous single presses of the intended buttons. The chip is not synchronized and inputs are not designed only for buttons so there is no debouncing of the switches.

Hexadecimal counter

The files will include a hexadecimal counter with a 7-segment display and a binary output using a 4 bit binary and a 4 bit grey coded output. The 7-segment display will count upwards from 0x0-0xF, and will then overflow to 0x0 again. The binary and grey coded bits will simultaneously represent the state of the counter.

Adventure Game

The adventure game described in E85 lab 3 will be implemented with LEDs to indicate the position of the player, whether or not they have the sword and whether or not they have won or lost. Four buttons will allow them to navigate North, East, South and West.

Seven Segment Decoder

A row of switches will allow the user to enter four-bit numbers asynchronously and the PLA operating combinationally will decode them into a seven-segment display signal.

Design Time

Activity	Time Required
HDL production	2 hours
HDL debug	30 minutes
HDL revision	2 hours
Assembler Design and Test	1 hour
Assembler revision	2 hours
PLA file generation	2 hours
PLA edits	30 minutes
Schematic production	4 hours
Schematic debug	1 hour
Schematic revision	2 hours
Layout Production	10 hours

Note that revision of HDL and schematic was to aid layout generation and verification of the PLA. Assembler revision was to reorder bits fed into the scanchain.

File Locations

The files will be placed in the following location before this report is handed in so that they can later be recovered in order to aid future work and testing. All files will be available for read to all users on Chips in the following directories.

Verilog Code

```
/users/amacrae/courses/e158/proj2/verilog
```

Configuration Files, Batch Tester and Assembler

```
/users/amacrae/courses/e158/proj2/config
```

Cadence Libraries

```
/users/amacrae/courses/e158/proj2/IC_CAD/cadence
```

HSPICE simulations

```
/users/amacrae/courses/e158/proj2/HSPICE
```

CIF

```
/users/amacrae/courses/e158/proj2/CIF
```

Chip Plot PDF

```
/users/amacrae/courses/e158/proj2/plot
```

Report PDF

```
/users/amacrae/courses/e158/proj2/report
```

References

None of this would have happened without the following works:

Harris and Weste, *CMOS VLSI Design 3rd ed*. Addison Wesley, 2005.

Kidder, Tracy. *The Soul of a New Machine*. Back Bay Books, 2000. (Originally published by Little, Brown, and Company in 1981).

"Programmable Logic Arrays". Wikipedia. http://en.wikipedia.org/wiki/Programmable_logic_array, accessed April 19, 2010.

"Programmable Array Logic". Wikipedia. http://en.wikipedia.org/wiki/Programmable_Array_Logic, accessed April 19, 2010.

Harris and Harris. "E85 Lab 3: Adventure Game." Elsevier, 2007. http://www3.hmc.edu/~harris/class/e85/Lab03.pdf, accessed April 19, 2010.

Appendices

Appendix A

Verilog Testbench and Modules

testbench.sv

```
`include "pla.sv"
`timescale 1ns / 100ps
module testbench();
  reg                    ph1, ph2, reset;
  reg                    configD;
  wire                   configQ;
  reg  [7:0]             a;
  reg  [15:0]             yexpected;
  wire [15:0]             y;
  reg                    logicph1, logicph2, configph1, configph2;
  reg  [519:0]           config [0:1];
  reg  [31:0]            configmum;
  reg  [24:0]            testvectors [0:100];
  reg  [31:0]            vectornum, errors;
  pla dut(a,
          y, configph1, configph2,
             logicph1,  logicph2,
             configD,   configQ,   reset);
  //generate clock
  always
  begin
    ph1 = 0;ph2 = 0; #5; ph1=1; #5;ph1=0; #5;ph2=1;#5;
  end      //assign clock to drive configuration or testing

  always @ (*)
  begin
    if (configmum<=520)
    begin
      configph1 <= ph1;
      configph2 <= ph2;
      logicph1  <= 0;
      logicph2  <= 0;
    end

    else
    begin
      configph1 <= 0;
      configph2 <= 0;
      logicph1  <= ph1;
      logicph2  <= ph2;
    end
  end

  // at start of test, load vectors
  // and pulse reset
  initial
  begin
    $display ("testing 4count");
    $readmemb("4count.tv", testvectors);
    $readmemb("4countc.tv", config);
    vectornum = 0; errors = 0; configmum=0;
    reset <= 1;

    #10430; //wait for configuration binary to be loaded
    reset <= 0;
  end     // update configuration on rising edge of configph2

  always @ (posedge configph1)
  begin
    #1;
    configD=config[0][configmum];
    configmum = configmum+1;
  end     // apply test vectors on rising edge of clk
```

```
  always @ (posedge logicph1)
  begin
    #1; {a, yexpected} = testvectors[vectornum];
  end     // check results on rising edge of logicph1

  always @ (posedge logicph2)
  begin
    if(~reset)
    begin
    vectornum = vectornum + 1;
      if (y !== yexpected)
      begin
        $display ("Error: a = %b", a);
        $display (" (%b computed)\n (%b expected)", y, yexpected);
        errors = errors + 1;
      end

      else
      begin
        $display ("Successfully Computed: %b = %b",a,y);
      end

      if (testvectors[vectornum][0] === 1'bx)
      begin
        $display("%d tests completed with %d errors", vectornum, errors);
        $finish;
      end
    end
  end
endmodule
```

pla.sv

```
/* HMuM (Harvey Mudd Micro Machine, yes the u is a mu)
 * William Koven and Drew Macrae
 * 21 March 2010
 *
 * A relatively simple PLA that is configured by a giant scan chain
 * and has 8 inputs, 8 outputs, and any output can be fed back to
 * its corresponding input (i.e. input a[1] either comes from off chip
 * or from output[1]).
 */
`timescale 1ns / 100ps

module pla(a, y, scan_ph1, scan_ph2, fb_ph1, fb_ph2, d, q, reset);

        input  logic scan_ph1, scan_ph2, fb_ph1, fb_ph2, d;
        input  logic [7:0] a;
        output logic q;
        output logic [7:0] y;
         input  logic reset;

        // either a or feedback selected by a mux
        logic [7:0] ins;
        logic [7:0] ins_b;

        logic [7:0] feedback;

        logic d_and, d_or;

        assign ins_b = ~ins;

        // the products specified by the programming
        tri1 [15:0] products;
        tri1 [7:0] or_out;

        logic [7:0] state;

        and_block andblock(ins, ins_b, products, scan_ph1, scan_ph2, d, d_and);
        or_block orblock(products, or_out, scan_ph1, scan_ph2, d_and, d_or);
```

```
        assign y = ~or_out;

        flopr #(8) feedback_flops(fb_ph1, fb_ph2, reset, y, state);

        // muxes for possible feedback
        mux_slice feedback_mux(a,state,feedback,ins);

        //feedback select flops
        shiftreg8 fb_flop(scan_ph1,scan_ph2,d_or,feedback);

        assign q = feedback[7];

endmodule

module shiftreg8(ph1,ph2,d,q);
        input        ph1;
        input        ph2;
        input        d;
        output [7:0] q;

        flop #(1) flop0(ph1, ph2, d, q[0]);
        flop #(1) flop1(ph1, ph2, q[0], q[1]);
        flop #(1) flop2(ph1, ph2, q[1], q[2]);
        flop #(1) flop3(ph1, ph2, q[2], q[3]);

        flop #(1) flop4(ph1, ph2, q[3], q[4]);
        flop #(1) flop5(ph1, ph2, q[4], q[5]);
        flop #(1) flop6(ph1, ph2, q[5], q[6]);
        flop #(1) flop7(ph1, ph2, q[6], q[7]);

endmodule

module mux_slice(d0,d1,s,y);
        input  [7:0] d0;
        input  [7:0] d1;
        input  [7:0] s;
         output [7:0] y;

        mux2 #(1) mux0(d0[0], d1[0], s[0], y[0]);
        mux2 #(1) mux1(d0[1], d1[1], s[1], y[1]);
        mux2 #(1) mux2(d0[2], d1[2], s[2], y[2]);
        mux2 #(1) mux3(d0[3], d1[3], s[3], y[3]);

        mux2 #(1) mux4(d0[4], d1[4], s[4], y[4]);
        mux2 #(1) mux5(d0[5], d1[5], s[5], y[5]);
        mux2 #(1) mux6(d0[6], d1[6], s[6], y[6]);
        mux2 #(1) mux7(d0[7], d1[7], s[7], y[7]);

endmodule

module or_block(bits, outs, ph1, ph2, d, q);

        input logic ph1, ph2, d;
        input logic [15:0] bits;
        output logic q;
        inout tri1 [7:0] outs;

        logic d1, d2, d3, d4, d5, d6, d7;

        // rows
        or_row row0(bits, outs[0], ph1, ph2, d, d1);
        or_row row1(bits, outs[1], ph1, ph2, d1, d2);
        or_row row2(bits, outs[2], ph1, ph2, d2, d3);
        or_row row3(bits, outs[3], ph1, ph2, d3, d4);

        or_row row4(bits, outs[4], ph1, ph2, d4, d5);
        or_row row5(bits, outs[5], ph1, ph2, d5, d6);
        or_row row6(bits, outs[6], ph1, ph2, d6, d7);
        or_row row7(bits, outs[7], ph1, ph2, d7, q);

endmodule
```

```
module or_row(bits, out, ph1, ph2, d, q);

        input logic ph1, ph2, d;
        input logic [15:0] bits;
        output logic q;
        inout tri1 out;

        logic d1, d2, d3, d4, d5, d6, d7, d8, d9;
        logic d10, d11, d12, d13, d14, d15;

        single_or_block block15(bits[15], out, ph1, ph2, d, d1);
        single_or_block block14(bits[14], out, ph1, ph2, d1, d2);
        single_or_block block13(bits[13], out, ph1, ph2, d2, d3);
        single_or_block block12(bits[12], out, ph1, ph2, d3, d4);

        single_or_block block11(bits[11], out, ph1, ph2, d4, d5);
        single_or_block block10(bits[10], out, ph1, ph2, d5, d6);
        single_or_block block9(bits[9], out, ph1, ph2, d6, d7);
        single_or_block block8(bits[8], out, ph1, ph2, d7, d8);

        single_or_block block7(bits[7], out, ph1, ph2, d8, d9);
        single_or_block block6(bits[6], out, ph1, ph2, d9, d10);
        single_or_block block5(bits[5], out, ph1, ph2, d10, d11);
        single_or_block block4(bits[4], out, ph1, ph2, d11, d12);

        single_or_block block3(bits[3], out, ph1, ph2, d12, d13);
        single_or_block block2(bits[2], out, ph1, ph2, d13, d14);
        single_or_block block1(bits[1], out, ph1, ph2, d14, d15);
        single_or_block block0(bits[0], out, ph1, ph2, d15, q);

endmodule

module single_or_block(a, out, ph1, ph2, d, q);

        input logic a, ph1, ph2, d;
        output logic q;
        inout tri1 out;

        flop #(1) aflop(ph1, ph2, d, q);

        assign out = (a & q) ? 1'b0 : 1'bz;

endmodule

module and_block(inputs, inputs_b, bits, ph1, ph2, d, q);

        input logic [7:0] inputs, inputs_b;
        input logic ph1, ph2, d;
        output logic q;
        inout tri1 [15:0] bits;

        and_row row0(inputs[0], inputs_b[0], bits, ph1, ph2, d, d1);
        and_row row1(inputs[1], inputs_b[1], bits, ph1, ph2, d1, d2);
        and_row row2(inputs[2], inputs_b[2], bits, ph1, ph2, d2, d3);
        and_row row3(inputs[3], inputs_b[3], bits, ph1, ph2, d3, d4);

        and_row row4(inputs[4], inputs_b[4], bits, ph1, ph2, d4, d5);
        and_row row5(inputs[5], inputs_b[5], bits, ph1, ph2, d5, d6);
        and_row row6(inputs[6], inputs_b[6], bits, ph1, ph2, d6, d7);
        and_row row7(inputs[7], inputs_b[7], bits, ph1, ph2, d7, q);


endmodule


module and_row (a, a_b, bits, ph1, ph2, d, q);
        input logic a, a_b, ph1, ph2, d;
        output logic q;
        inout tri1 [15:0] bits;

        logic d1, d2, d3, d4, d5, d6, d7, d8;
        logic d9, d10, d11, d12, d13, d14, d15;
```

```
        single_and_block block15(a, a_b, bits[15], ph1, ph2, d, d1);
        single_and_block block14(a, a_b, bits[14], ph1, ph2, d1, d2);
        single_and_block block13(a, a_b, bits[13], ph1, ph2, d2, d3);
        single_and_block block12(a, a_b, bits[12], ph1, ph2, d3, d4);

        single_and_block block11(a, a_b, bits[11], ph1, ph2, d4, d5);
        single_and_block block10(a, a_b, bits[10], ph1, ph2, d5, d6);
        single_and_block block9(a, a_b, bits[9], ph1, ph2, d6, d7);
        single_and_block block8(a, a_b, bits[8], ph1, ph2, d7, d8);

        single_and_block block7(a, a_b, bits[7], ph1, ph2, d8, d9);
        single_and_block block6(a, a_b, bits[6], ph1, ph2, d9, d10);
        single_and_block block5(a, a_b, bits[5], ph1, ph2, d10, d11);
        single_and_block block4(a, a_b, bits[4], ph1, ph2, d11, d12);

        single_and_block block3(a, a_b, bits[3], ph1, ph2, d12, d13);
        single_and_block block2(a, a_b, bits[2], ph1, ph2, d13, d14);
        single_and_block block1(a, a_b, bits[1], ph1, ph2, d14, d15);
        single_and_block block0(a, a_b, bits[0], ph1, ph2, d15, q);


endmodule

module single_and_block(a, a_b, product, ph1, ph2, d, q);
        input logic a, a_b, ph1, ph2, d;
        output logic q;
        inout tri1 product;

        logic d_q;

        flop #(1) aflop(ph1, ph2, d, d_q);
        flop #(1) a_bflop(ph1, ph2, d_q, q);

        assign product = (a & d_q) | (a_b & q) ? 1'b0 : 1'bz;

endmodule

// Taken from E158 lab2 mips.sv
module flop #(parameter WIDTH = 8)
             (input  logic             ph1, ph2,
              input  logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q);

  logic [WIDTH-1:0] mid;

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

module flopen #(parameter WIDTH = 8)
               (input  logic             ph1, ph2, en,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);

  logic [WIDTH-1:0] d2;

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

module flopr #(parameter WIDTH = 8)
                (input  logic             ph1, ph2, reset,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

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

  assign resetval = 0;

  mux2 #(WIDTH) enrmux(d, resetval, reset, d2);
  flop #(WIDTH) f(ph1, ph2, d2, q);
endmodule
```

```
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module latch #(parameter WIDTH = 8)
              (input  logic             ph,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_latch
    if (ph) q <= d;
endmodule
```

Appendix B

Testbench.sp

```
* testbench.sp
* William Koven 3 April 2010
* To test the configPull cell for the HMuM (Koven and Macrae)
* which is essentially a flop with a series pull down nmos for
* data. The cell is a configurable pull down down to be used in
* a AND and OR blocks.

*************************************************************************
* Parameters and models
*************************************************************************
.param SUP=3.3
.lib '~/simulation/layout_models_ami06/opConditions.lib' TT
*.option scale=.3u
.option post

*************************************************************************
* Simulation netlist
*************************************************************************
.global vdd!
.include 'netlist'

*************************************************************************
* Supplies
*************************************************************************
Vdd vdd! gnd 'SUPPLY'
Vph1 ph1 gnd PULSE 0 'SUPPLY' 75ns 100ps 100ps 70ns 150ns
Vph1b ph1b gnd PULSE 'SUPPLY' 0 75ns 100ps 100ps 70ns 150ns
Vph2 ph2 gnd PULSE 0 'SUPPLY' 1ns 100ps 100ps 70ns 150ns
Vph2b ph2b gnd PULSE 'SUPPLY' 0 1ns 100ps 100ps 70ns 150ns

Va a gnd PWL 0ns +
  'SUPPLY' 910ns +
  'SUPPLY' 920ns 0 1010ns 0 1020ns +
  'SUPPLY' 1150ns +
  'SUPPLY' 1160ns 0
Vd d gnd PWL 0 0 18.9ns 0 19ns +
  'SUPPLY' 118.9ns +
  'SUPPLY' 125ns 0 380ns 0 385ns +
  'SUPPLY' 430ns +
  'SUPPLY' 435ns 0 500ns 0 501ns +
  'SUPPLY'

*************************************************************************
* Stimulus
*************************************************************************
.tran 5ps 1500ns

.end
```

Appendix C

Schematics and layouts of newly generated cells

Undocumented cellviews are taken from Muddlib10

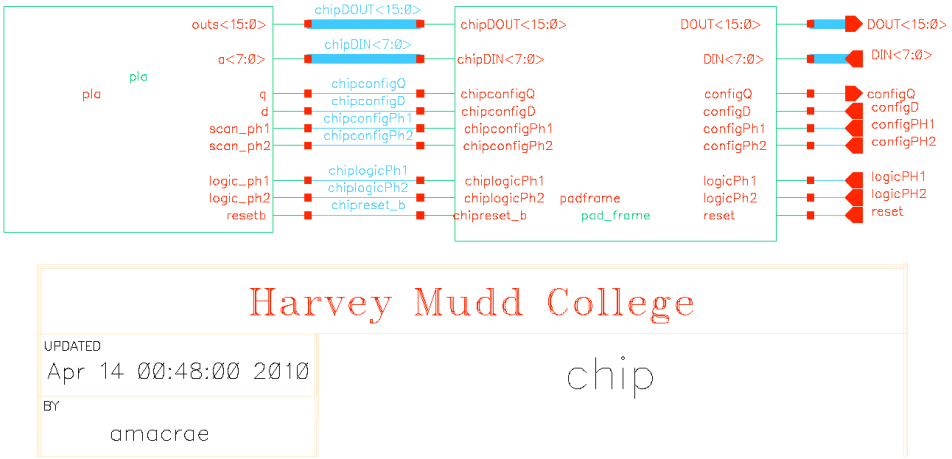


Figure 6 chip schematic

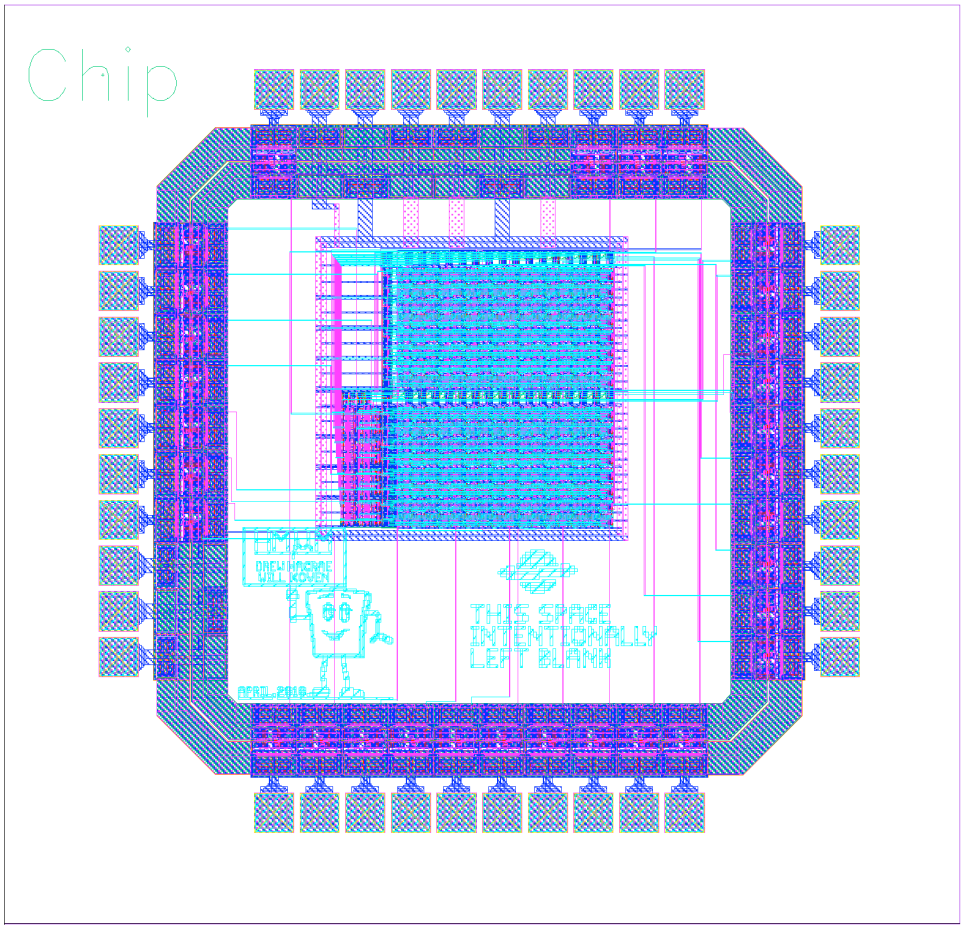


Figure 7 chip layout

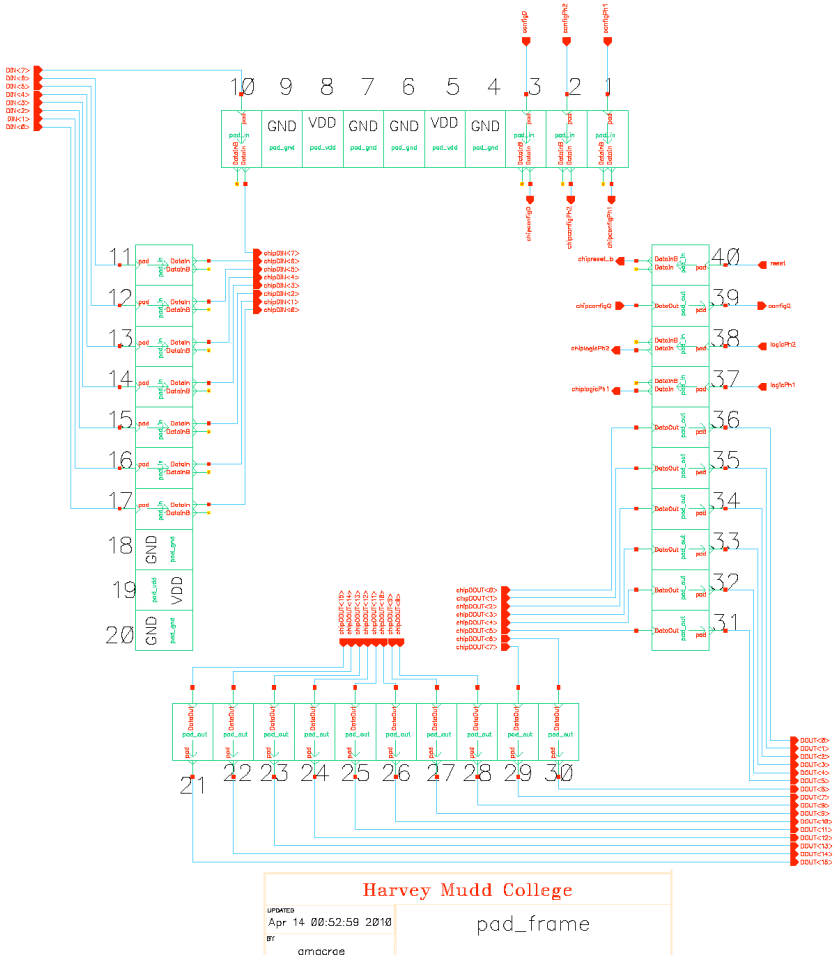


Figure 8 pad_frame schematic

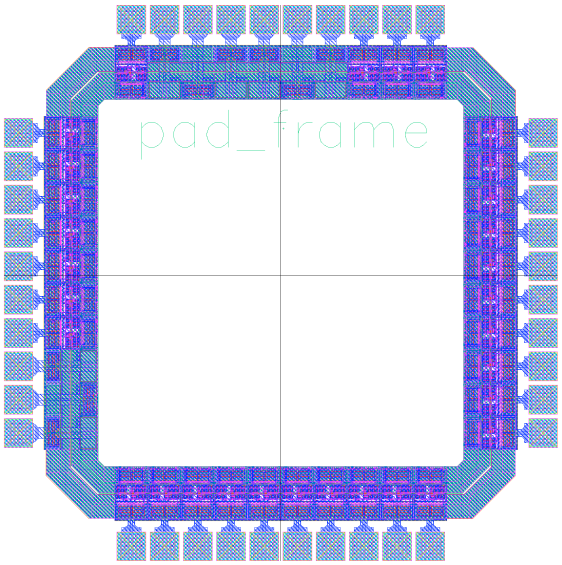


Figure 9 pad_frame layout

This cell was copied from the MIPS library and then edited to provide the needed pins.

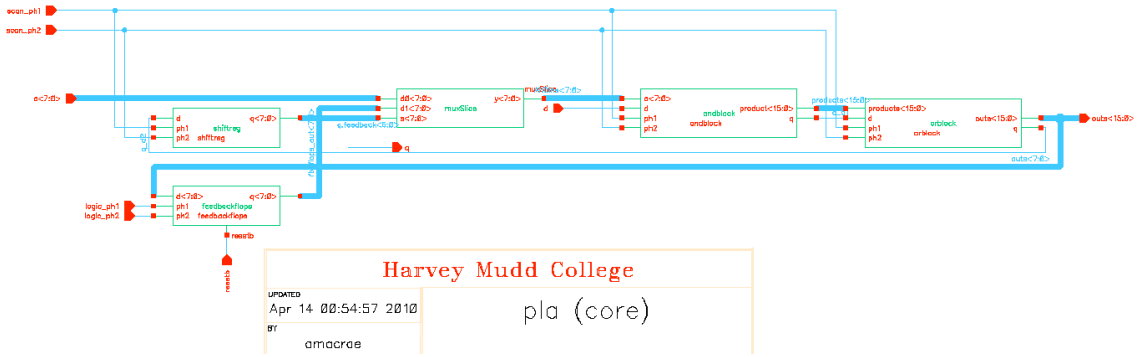


Figure 10 pla schematic

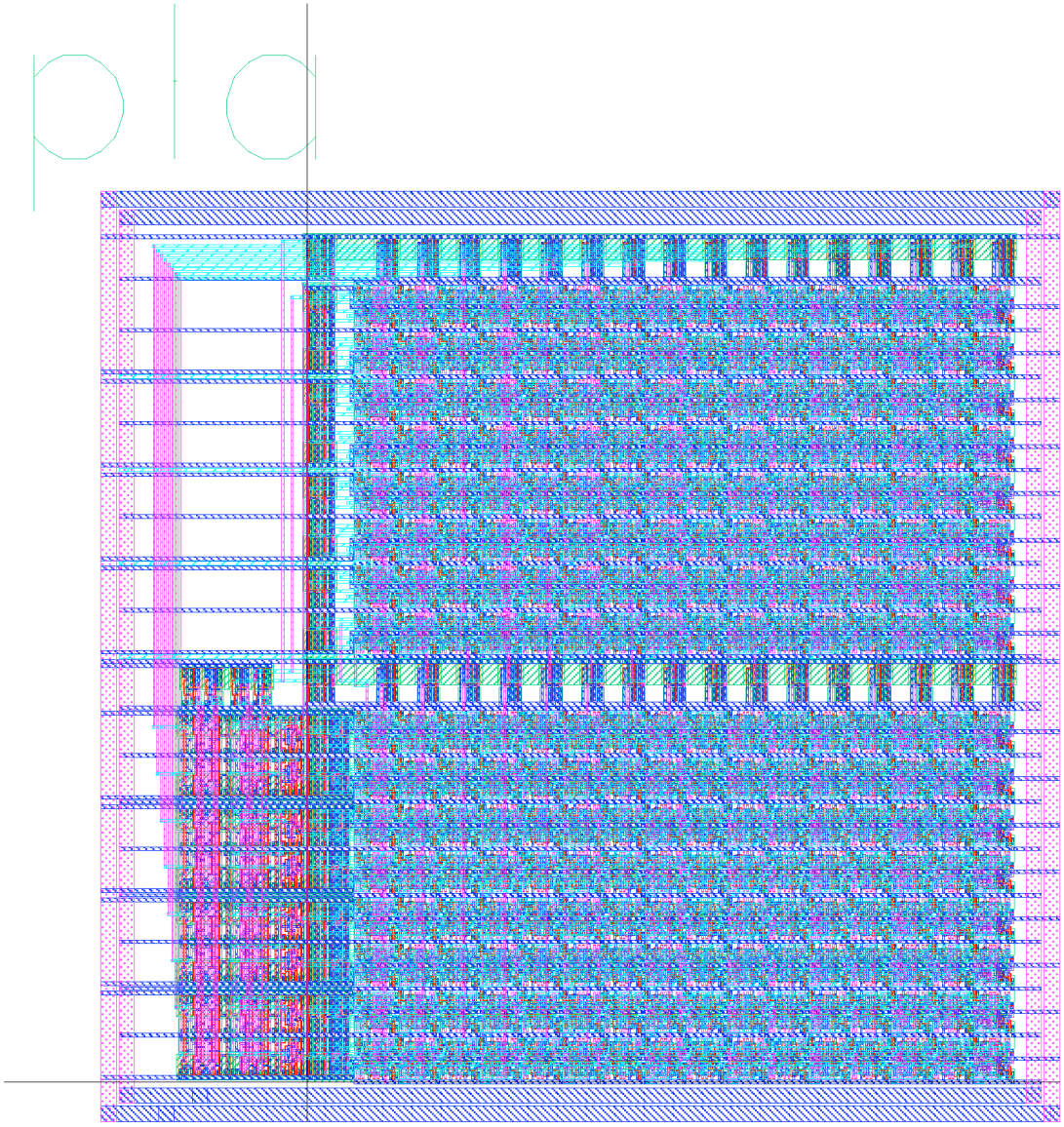


Figure 11 pla layout

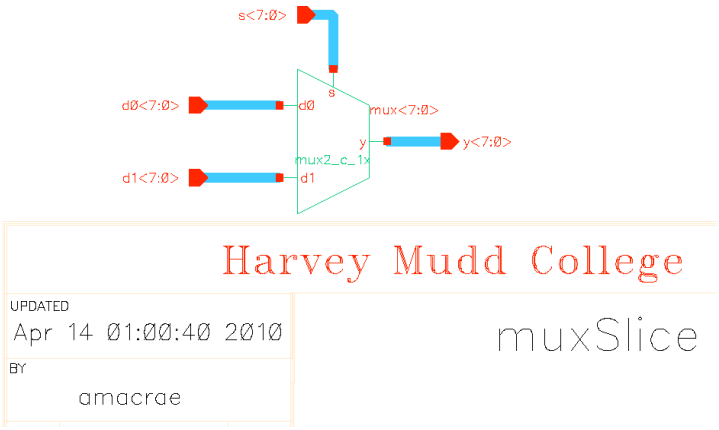


Figure 12 muxSlice schematic

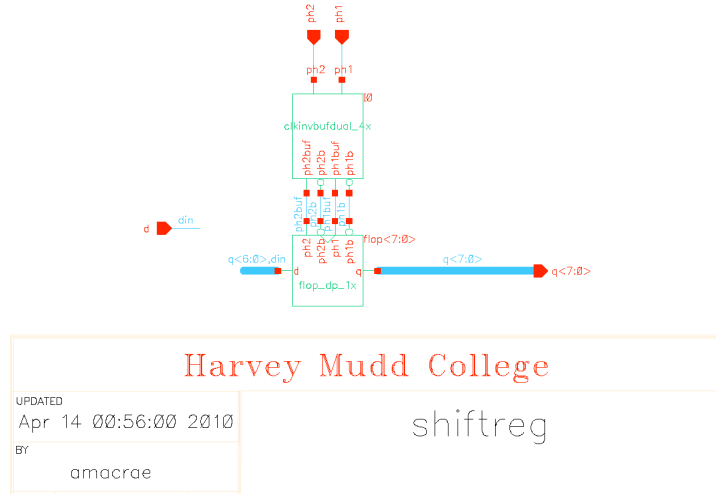


Figure 13 shiftreg schematic

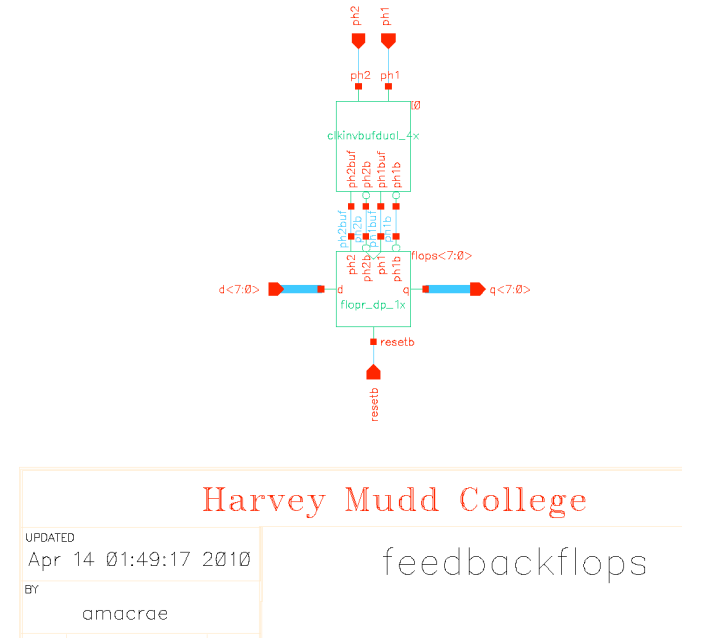
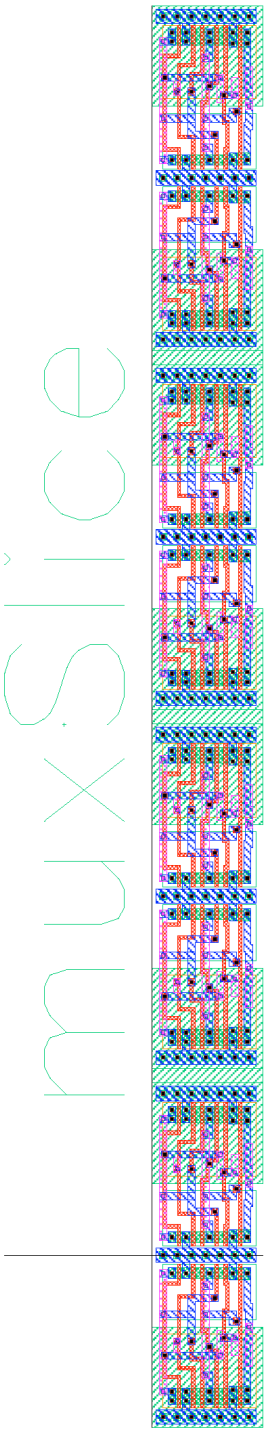
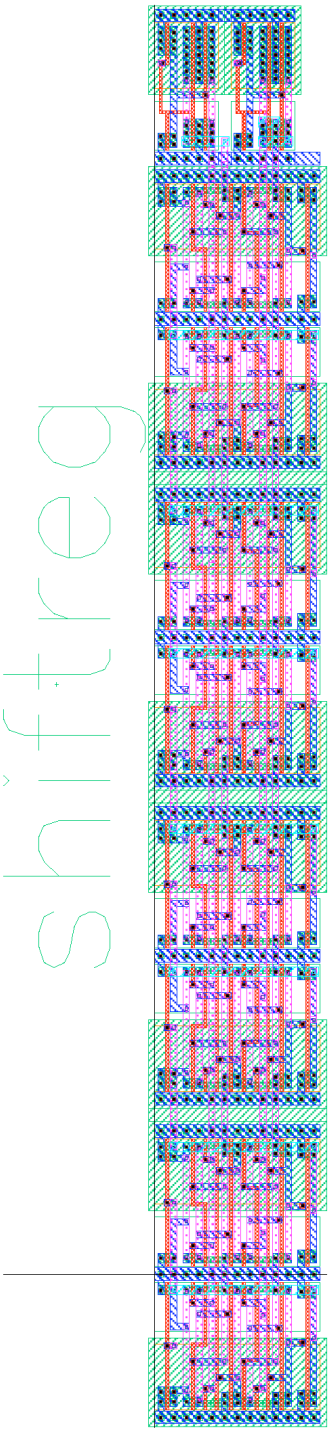


Figure 14 feedbackflops





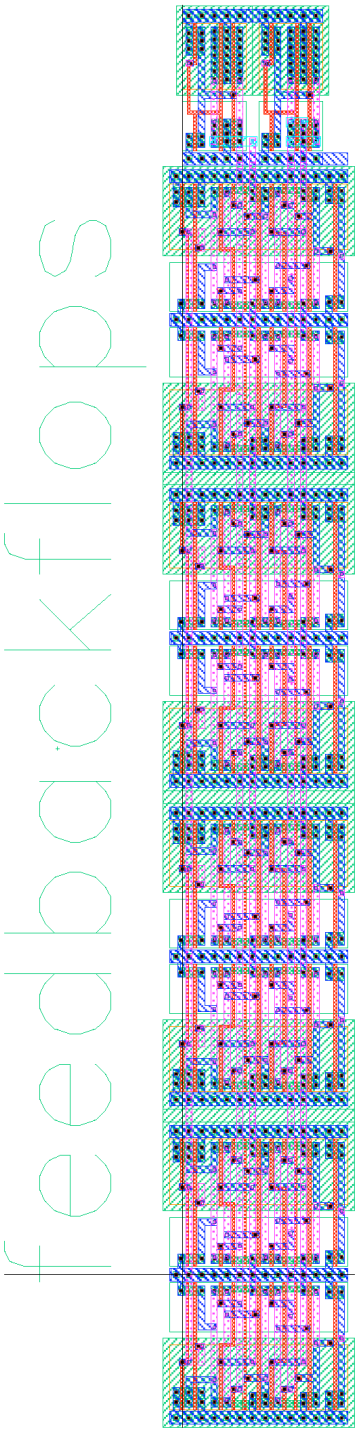


Figure 15 muxSlice layout **Figure 16 shiftreg layout** **Figure 17 feedbackflops layout**

These cells are all pitch matched to the androws, hence the odd vertical arrangement.

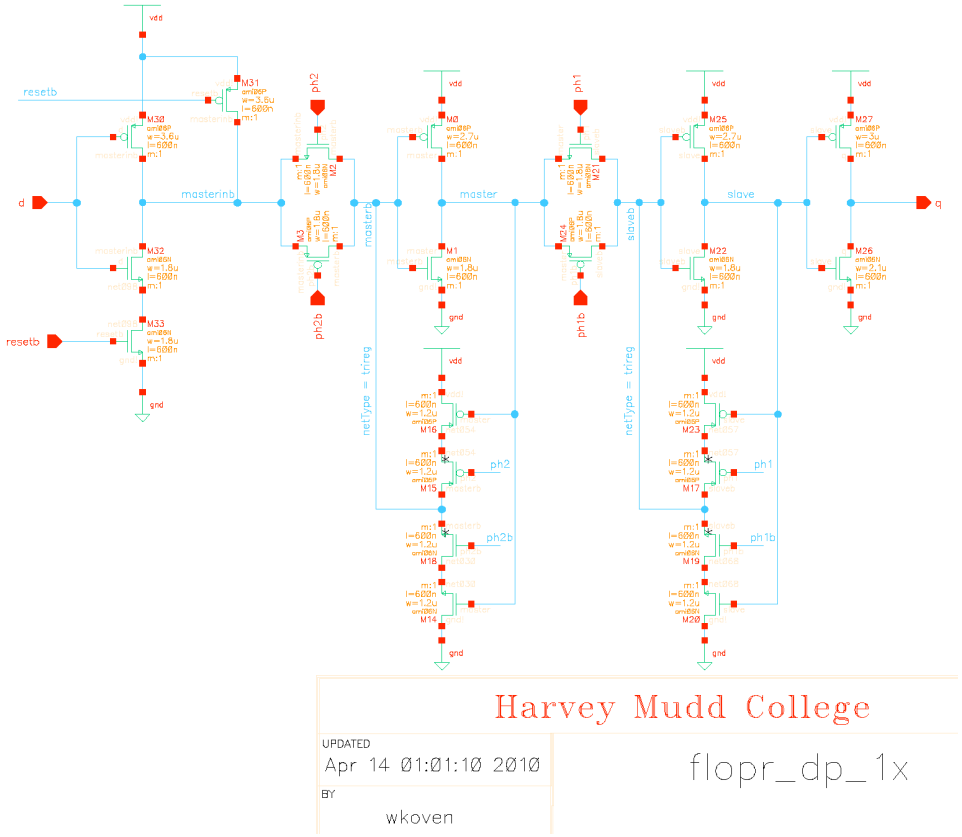


Figure 18 flopr_dp_1x schematic

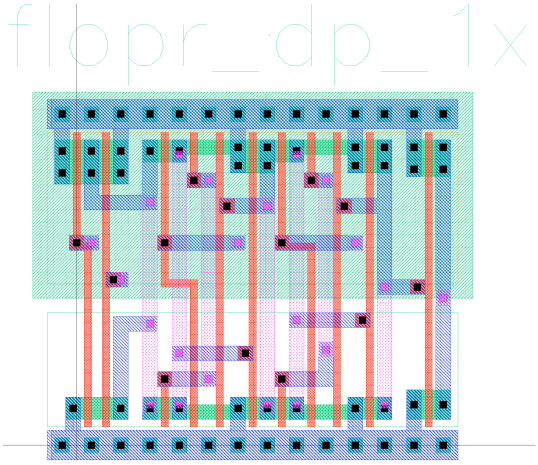


Figure 19 flopr_dp_1x layout

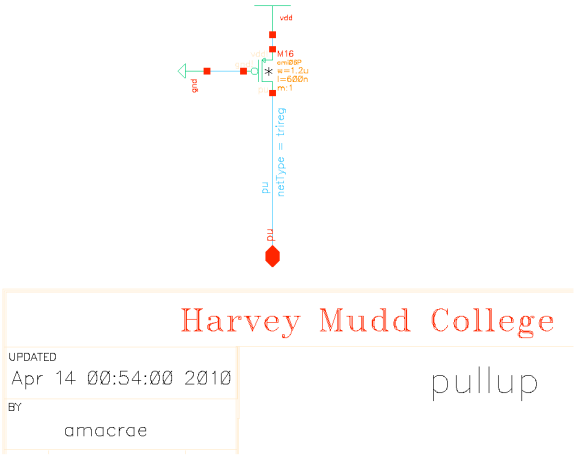


Figure 20 pullup schematic

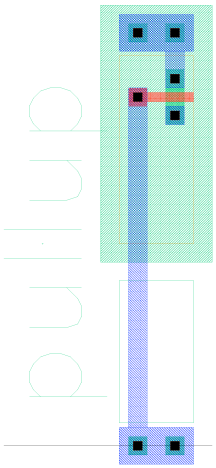


Figure 21 pullup layout

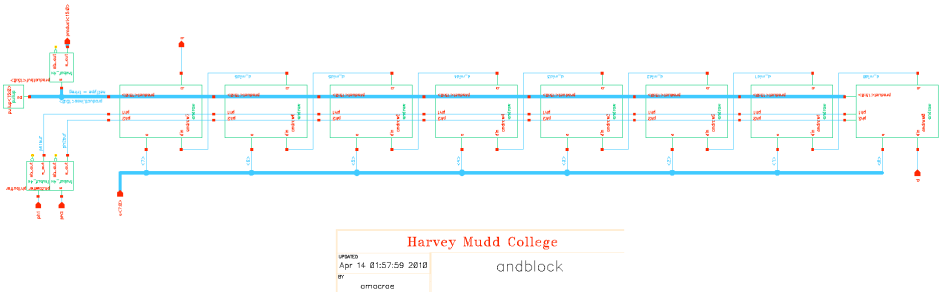


Figure 22 andblock schematic

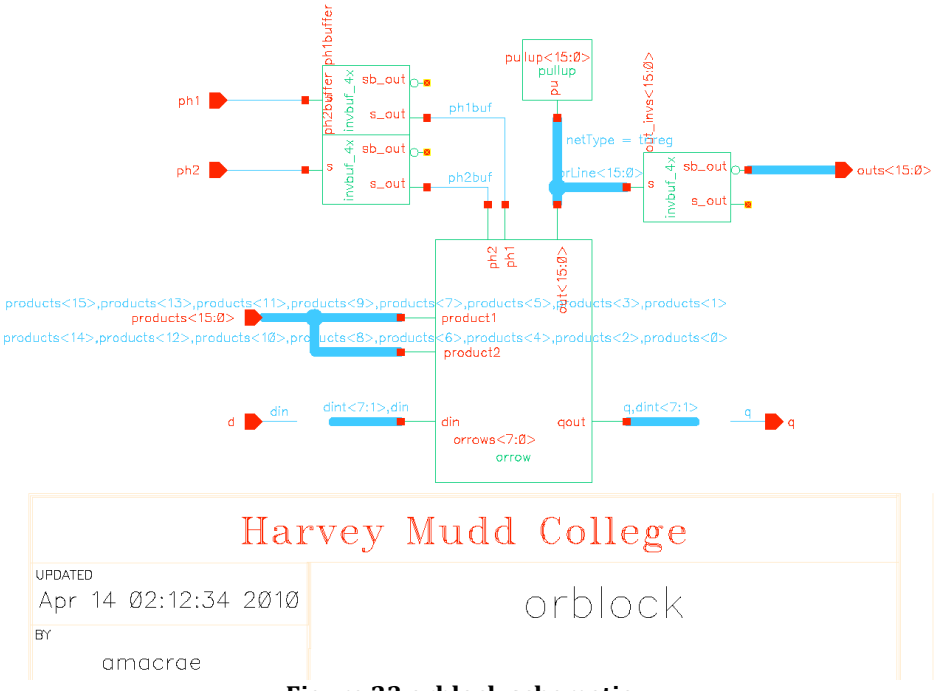


Figure 23 orblock schematic

The andblock and orblock are very similar, difference in schematics highlight andblock scanchain arrangement and orblock arrangement of product lines and clocks.

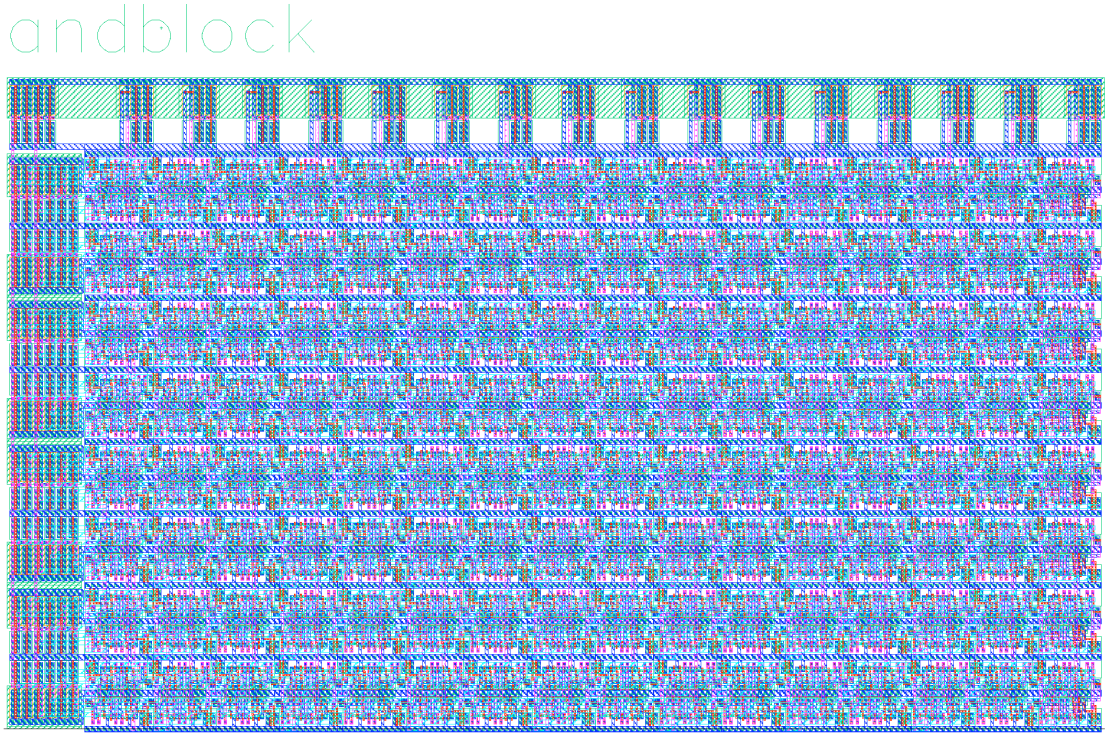


Figure 24 andblock layout

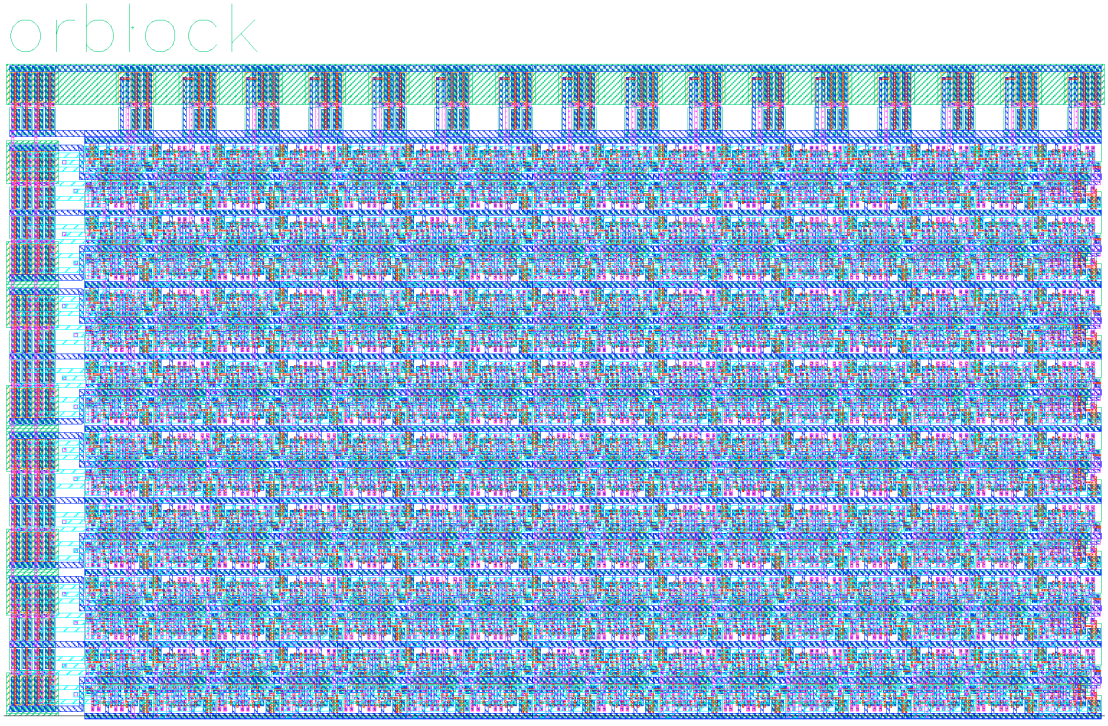


Figure 25 orblock layout

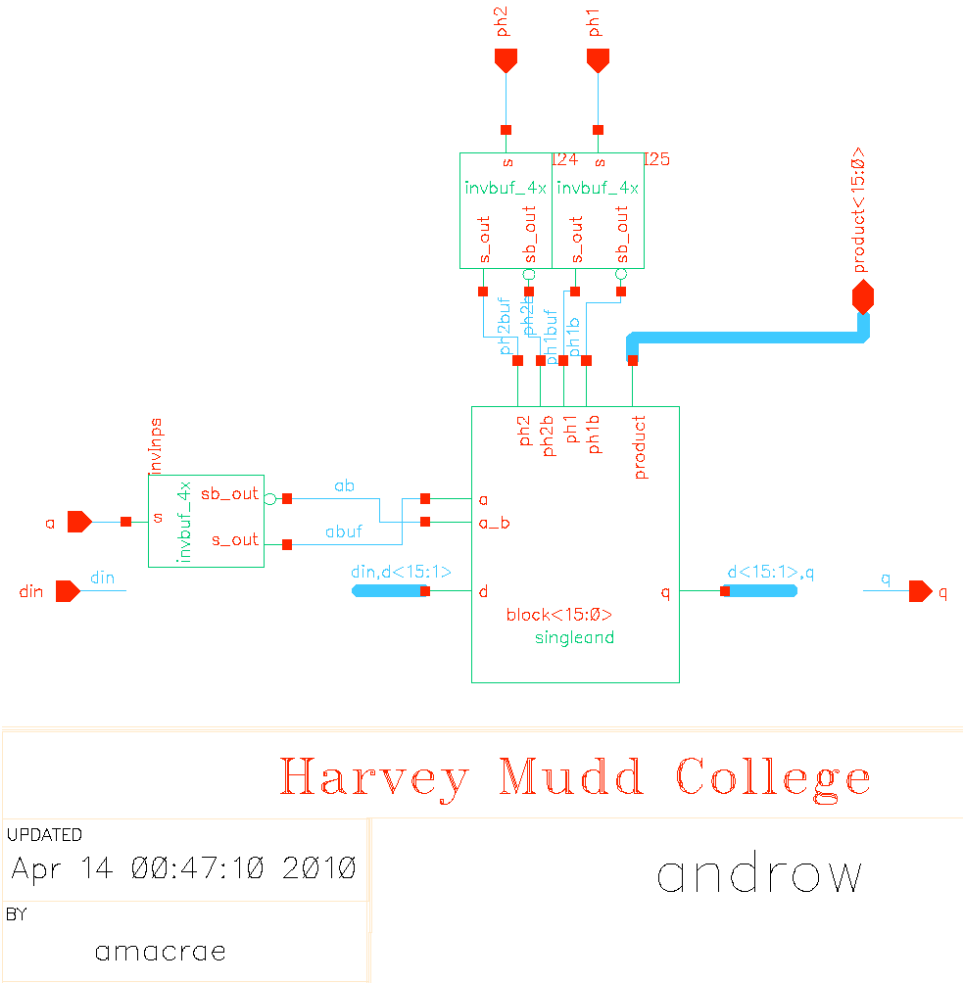


Figure 26 androw schematic

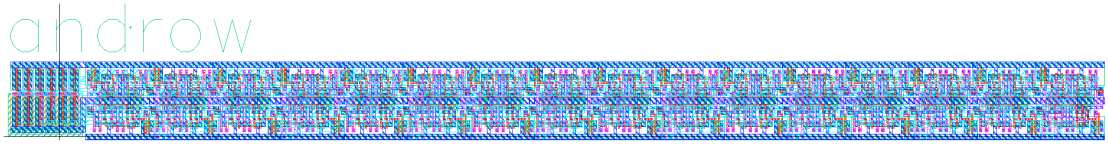


Figure 27 androw layout

One of two androw layouts. The other simply has the inverters flipped to facilitate stacking the androws. The other layout is named androw upsie and is functionally the same. The scanchain starts at the top left, runs to the right before returning to the left along the bottom so that no long wires are required in the layout. The clock is passed along the row.

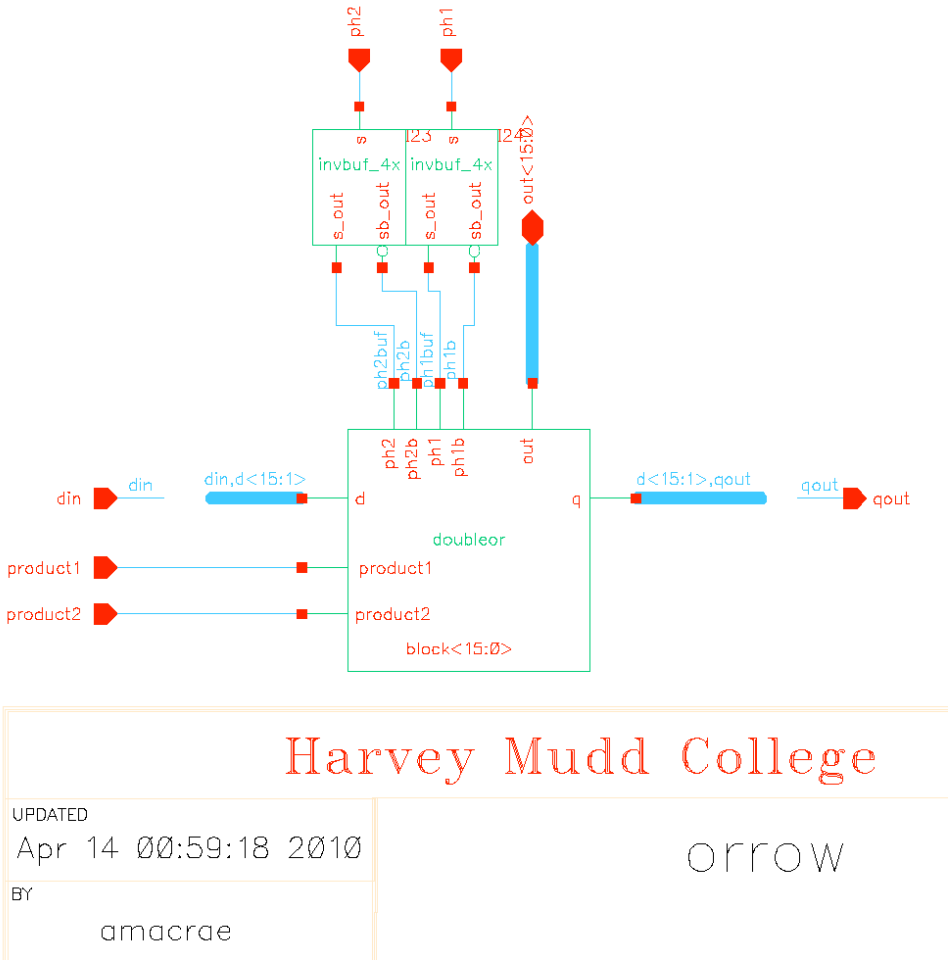


Figure 28 orrow schematic

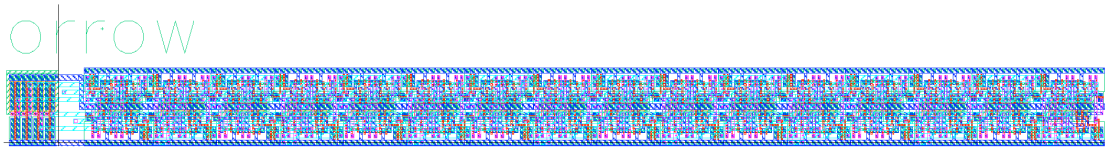


Figure 29 orrow layout

Like the androw, the orrow has two layout views that are functionally the same. The orrow is very similar to the androw.

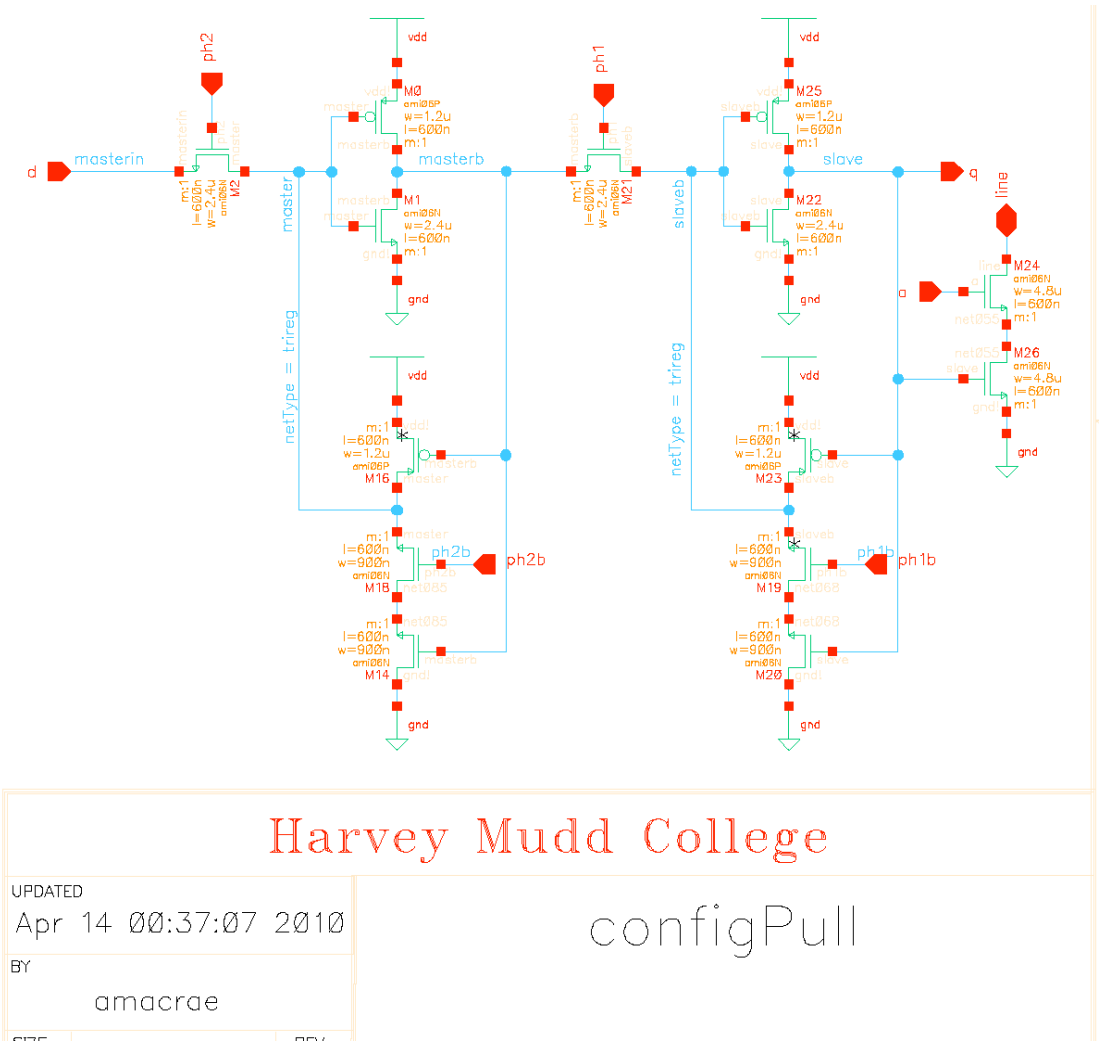


Figure 30 configPull schematic

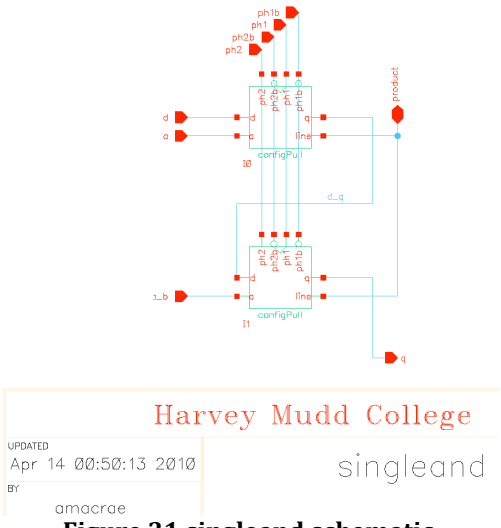


Figure 31 singleand schematic

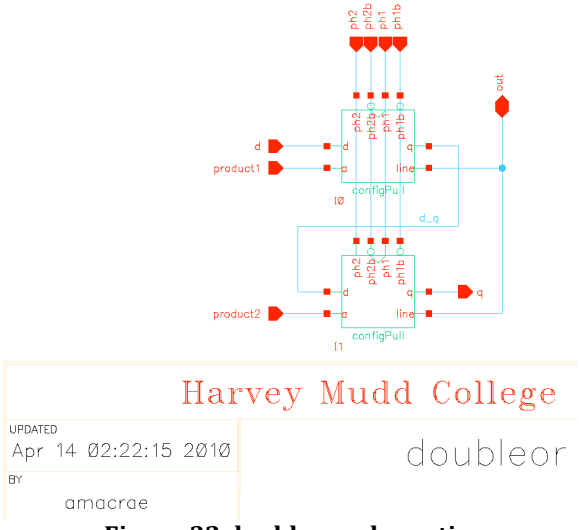


Figure 32 doubleor schematic

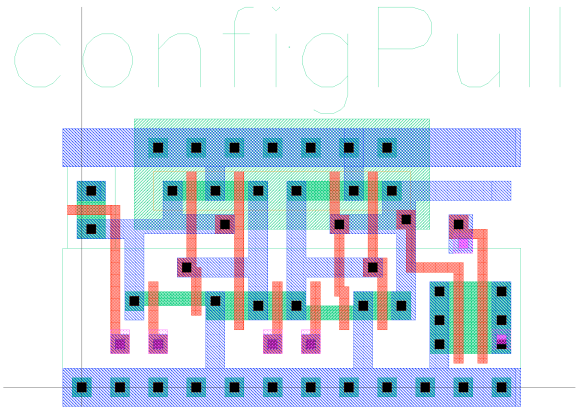


Figure 33 configPull layout

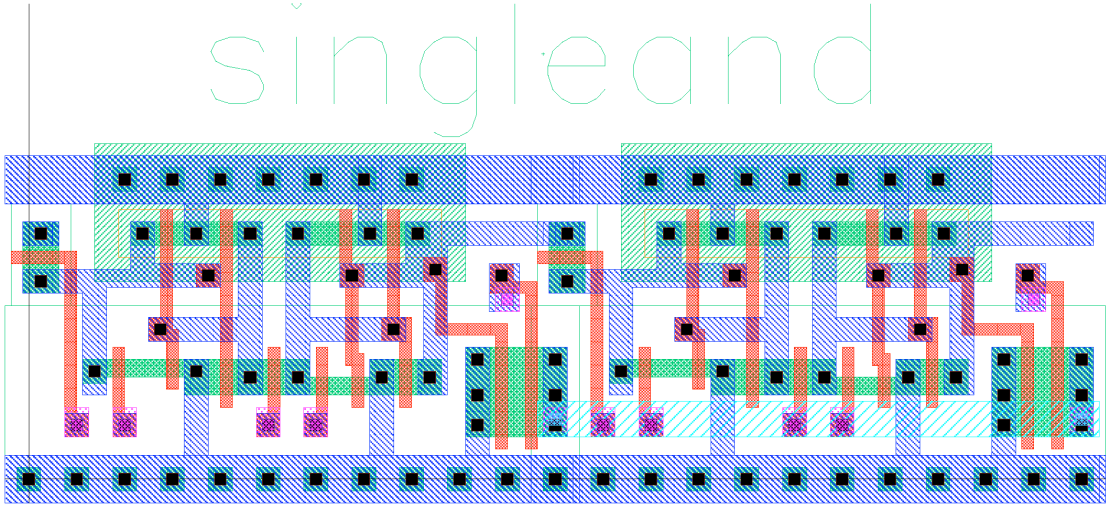


Figure 34 singleand layout

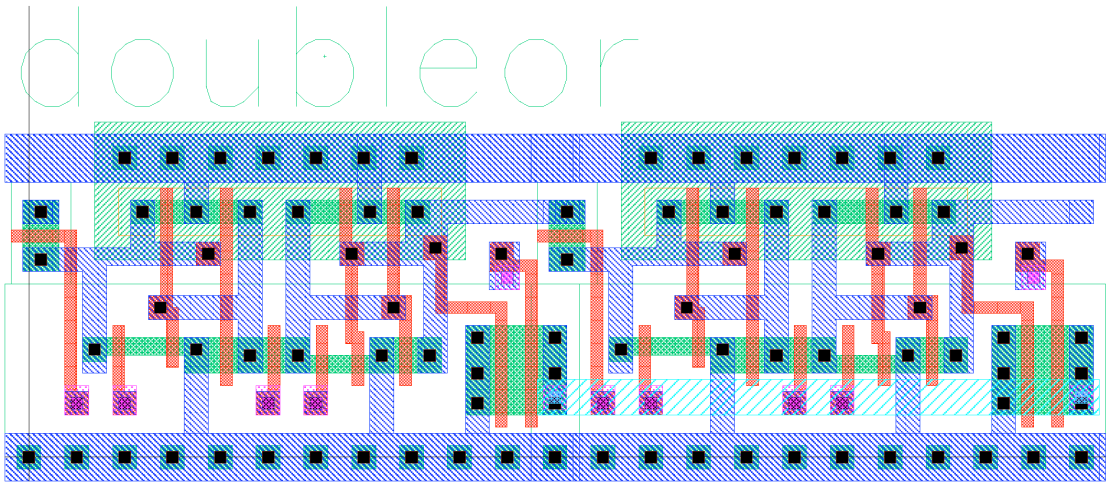


Figure 35 doubleor layout

Here you can see the miniaturized shift register and how it can be assembled into a configurable pulldown that drives one output based on an input and its complement or drives two outputs based on two minterms.

Appendix D

Sample PLA configuration files, testing script and a dot diagram assembler

blank.pla

```
//blank.pla
//this is a commented example of a file to configure the PLA
Products:  //input[7:0] is arranged on the columns. characters show
a xxxxxxxx //what's required to make a lettered product line go high
b xxxxxxxx //can be 1, 0 or x for high, low or don't care
c xxxxxxxx //there are 15 product lines  ordered top to bottom,
d xxxxxxxx //inputs are ordered as they'd appear in a test
e xxxxxxxx
f xxxxxxxx
g xxxxxxxx
h xxxxxxxx
i xxxxxxxx
j xxxxxxxx
k xxxxxxxx
l xxxxxxxx
m xxxxxxxx
n xxxxxxxx
o xxxxxxxx
p xxxxxxxx

Sums:                //a-p are arranged on the columns in the sum array
15 00000000_00000000 //which product lines are summed to make the
14 00000000_00000000 //outputs can be 1 or 0 for inclusion or exclusion
13 00000000_00000000 //from the sum the topmost product is at the left
12 00000000_00000000
11 00000000_00000000 //the topmost sum would appear as the leftmost in
10 00000000_00000000 //the test
09 00000000_00000000
08 00000000_00000000
07 00000000_00000000
06 00000000_00000000
05 00000000_00000000
04 00000000_00000000
03 00000000_00000000
02 00000000_00000000
01 00000000_00000000
00 00000000_00000000

Feedback:
7 0          //which inputs are bypassed to include a registered output
6 0          //a 1 indicates that the registered output is linked
5 0          //registers join inputs of similar names
4 0
3 0
2 0
1 0
0 0
Tests:       //a[7:0] y[15:0] writing tests is recommended.
00000000_0000000000000000
00000000_0000000000000000
```

7seg.pla

```
//7segment display decoder identifies hex signals as 1hot products and
//produces 7 segment codes with the sums. This is provided as an
//example for combinational logical operation
Products:
a 0000xxxx
b 0001xxxx
c 0010xxxx
d 0011xxxx
e 0100xxxx
f 0101xxxx
g 0110xxxx
h 0111xxxx
i 1000xxxx
j 1001xxxx
k 1010xxxx
l 1011xxxx
m 1100xxxx
n 1101xxxx
o 1110xxxx
p 1111xxxx
Sums:
a 01001000_00011100
b 00000110_00011011
c 00100000_00001011
d 01001001_01100001
e 01011101_01000000
f 01110001_00001100
g 11000001_00000000
h 00000000_00000000
Feedback:
a 0
b 0
c 0
d 0
e 0
f 0
g 0
h 0
Tests:
00000000_00000010
00010000_10011110
00100000_00100100
00110000_00001100
01000000_10011000
01010000_01001000
01100000_01000000
01110000_00011110
10000000_00000000
10010000_00011000
10100000_00010000
10110000_11000000
11000000_11100100
11010000_10000100
11100000_01100000
11110000_01110000
```

7shift.pla

```
//7shift produces a 7 bit shift register to demonstrate FSM behavior of
//the PLA. Products pass first input and the 7 enabled feedback signals
//to the sums which shift them by one before they reach the feedback
Products:
a 1xxxxxxx
b x1xxxxxx
c xx1xxxxx
d xxx1xxxx
e xxxx1xxx
f xxxxx1xx
g xxxxxx1x
h xxxxxxx1
i xxxxxxxx
j xxxxxxxx
k xxxxxxxx
l xxxxxxxx
m xxxxxxxx
n xxxxxxxx
o xxxxxxxx
p xxxxxxxx

Sums:
a 00000000_00000000
b 10000000_00000000
c 01000000_00000000
d 00100000_00000000
e 00010000_00000000
f 00001000_00000000
g 00000100_00000000
h 00000010_00000000

Feedback:
a 0
b 1
c 1
d 1
e 1
f 1
g 1
h 1

Tests:
10000000_01000000
00000000_00100000
10000000_01010000
00000000_00101000
10000000_01010100
00000000_00101010
10000000_01010101
00000000_00101010
10000000_01010101
00000001_00101010
```

chiptest.py

```
#chiptest.py
#executes a series of tests to evaluate proper operation in 9 modes
import os
import time

fileList=["3dec","8andnor","8ornand","22andornandnor","23xorxnor",\
          "42xorxnor","4count","7shift","7seg"]

os.system("python tvassembler.py")
  #compile all the files to run simulations

time.sleep(1)#just in case

for files in fileList:
  fin = open("testbenchGen.sv","r")
  fout = open("testbench.sv","w")

  print "Testing: "+files

  Lines = fin.readlines()

  for eachline in Lines:
    fout.write(eachline.replace("MODE",files))
      #findreplace to customize this run

    fin.close()
    os.fsync(fout)
    fout.close()

    os.system(\
      "sim-nc testbench.sv -f verilog.inpfiles|tee "+files+".log"\
    )

    time.sleep(10)
      #python doesn't wait for OS calls to finish before calling the
      #next ones.
```

Tvassembler.py

```
#tvassembler.py
#parses and assembles well formed, human readable "dot diagrams" into
#Verilog readable configuration files and test vectors

fileList=["3dec","8andnor","8ornand","22andornandnor","23xorxnor",\
          "42xorxnor","4count","7shift","7seg"]
Debug = False

for filenames in fileList:
  f = open(filenames+".pla")
  print filenames

  Products = []#instantiate lists of strings to encode configuration
  Sums     = []
  Feedback = []
  Tests    = []
  readingProducts = False#instantiate Booleans to store program state
  readingSums     = False
  readingFeedback = False
  readingTests    = False

  for lines in f:
    if lines[0:4]   == "Prod":  #if a tag is IDed,
      readingProducts = True    #advance to its state
    elif lines[0:4] == "Sums":
      readingProducts = False
      readingSums     = True
    elif lines[0:4] == "Feed":
      readingSums     = False
      readingFeedback = True
    elif lines[0:4] == "Test":
      readingFeedback = False
      readingTests    = True

    elif readingProducts == True:
      Products += [lines.replace("\n","")]#append line to products list
      if Products[-1]=="":                #remove empty lines
        Products = Products[:-1]
      else:                               #grab the configuration from
        Products[-1]=Products[-1].split()[1]#non-empty lines

    elif readingSums     == True:
      Sums += [lines.replace("\n","")]
      if Sums[-1]=="":
        Sums = Sums[:-1]
      else:
        Sums[-1]=Sums[-1].split()[1]

    elif readingFeedback == True:
      Feedback += [lines.replace("\n","")]
      if Feedback[-1]=="":
        Feedback = Feedback[:-1]
      else:
        Feedback[-1]=Feedback[-1].split()[1]
```

```
  elif readingTests    == True:
    Tests += [lines.replace("\n","")]
    if Tests[-1]=="":
      Tests = Tests[:-1]

f.close()
if Debug:
  print Products
  print len(Products)
  print Sums
  print len(Sums)
  print Feedback
  print len(Feedback)
  print Tests

config = "" #initialize the string to hold new configuration
bits = 0    #initialize the count of bits in the string as zero

for entries in Feedback:  #place everything appropriately for a
  config+=entries         #reversed scanchain, stripping _'s
for j in range(len(Sums)):
  for i in range(len(Sums[0])):
    config+=Sums[j][len(Sums[0])-i-1].replace("_","")
for i in range(len(Products[0])): #replace 1,0 and x with appropriate
  for j in range(len(Products)):  #logic while configuring products
    config+=Products[len(Products)-j-1][i].replace("0","ab").\
                          replace("1","ba").replace("x","aa")\
                           .replace("a","0").replace("b","1")
configfile = open(filenames+"c.tv","w")
configfile.write(config[::-1]) #reverse scanorder
configfile.write("\n")
configfile.close()

testfile = open(filenames+".tv","w")#testvectors should work as is
for lines in Tests:
  testfile.writelines(lines+"\n")
testfile.close()
```