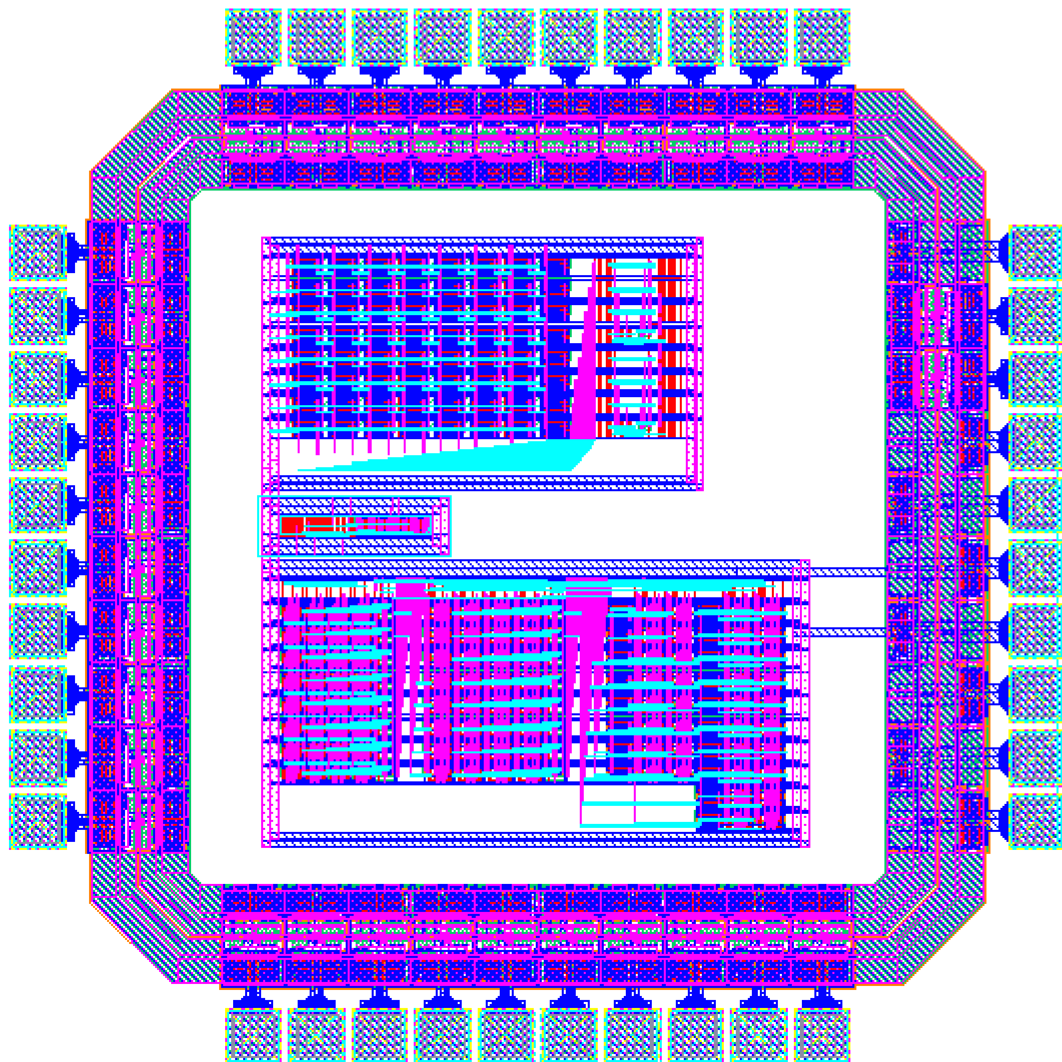
HARVEY MUDD COLLEGE
Department of Engineering

FIR Filter

E158 - CMOS VLSI Final Report



by

Jerry Hsiung, Max Waugaman

April 2015

Contents

1 Main Report **1**
- 1.1 Introduction 1
- 1.2 Specification 2
  - 1.2.1 Data input and output 2
  - 1.2.2 Clocks 3
  - 1.2.3 Shift in 3
- 1.3 Floorplan 4
- 1.4 Verification 7
- 1.5 Postfabrication Test Plan 7
- 1.6 Design Time 7
- 1.7 File Locations 7
- 1.8 References 8

A Verilog Code **9**

B Schematics **20**

C Layout **32**

Chapter 1

Main Report

1.1 Introduction

This paper describes the final project for the class: E158 - CMOS VLSI. The team implemented a finite impulse response (FIR) filter in a 1.5 x 1.5 mm 40-pin MOSIS "TinyChip" fabricated in a 0.6 μm process. The project size is 5000 x 5000 λ including I/O pads. The core of this project fits in a 3400 x 3400 λ box. Of the 40 pins, four are dedicated to VDD, and four to GND.

The FIR filter implemented is a fourth-order, causal discrete-time filter that is typically seen in signal processing. The chip consists of two sets of registers to hold two kinds of inputs: the 8-bit input signal $a[n]$ and its time delayed values, and four 8-bit coefficients, c_0, c_1, c_2, c_3. The user of the chip can change the behavior of the filter by inputting their own coefficients using a shift chain. Each value of the output sequence is a weighted sum of the most recent input values, which can be described as the equation below:

$$y[n] = c_0 a[n] + c_1 a[n-1] + c_2 a[n-2] + c_3 a[n-3]$$

The multiplier in the FIR filter is time multiplexed, so the chip only outputs a new result once every four clock periods after the accumulations finish. Besides the input and output registers, the FIR filter chip also consists of an 8-bit multiplier that calculates the product of each shifted input with its corresponding coefficients, and an 18-bit adder that accumulates the results. See Figure 1.1 for more detail.

1.2 Specification

The following section describes the operation of the filter. In addition, Figure 1.1 describes FIR filter design. The chip is used as follows

1. Load coefficients using shiftIn and shiftClk1,2.
2. Apply input data to pins a[7:0].
3. Run the main clock (ph1, ph2) for four cycles.
4. Read the filtered output on pins y[17:0].
5. Repeat steps 2-4.

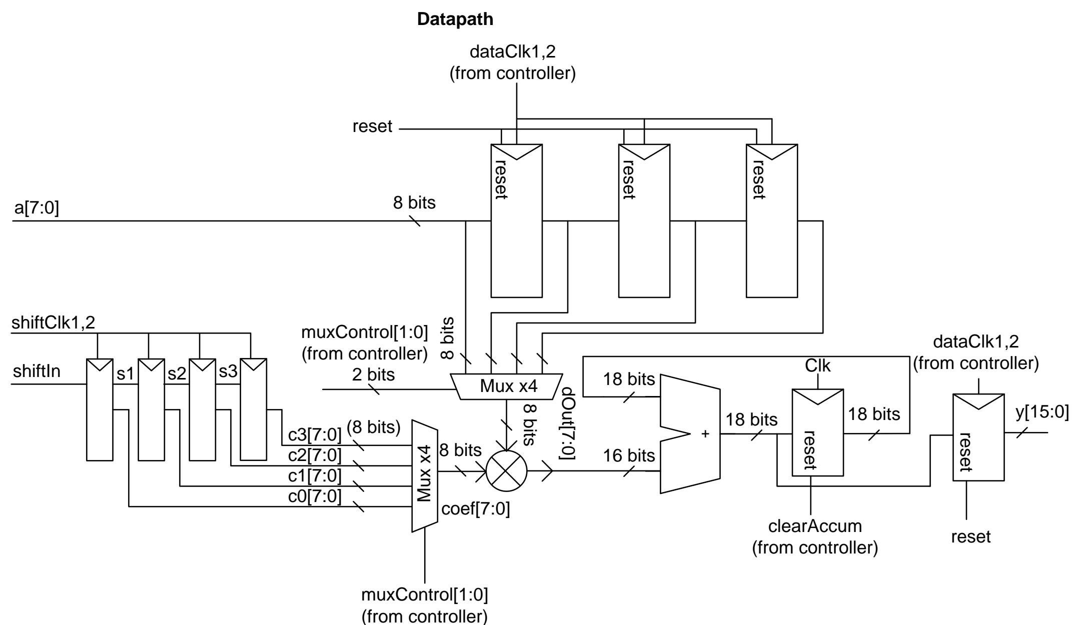


FIGURE 1.1: Schematic of the FIR filter

1.2.1 Data input and output

The eight input pins a[7:0] connect to the filter input data. The filtered output is located on eighteen pins y[17:0]. The FIR filter time multiplexes the multiplier four times, so the data throughput is effectively one fourth the main clock input (ph1, ph2). The values on pins y[17:0] update every fourth cycle of ph1 and ph2. The input pins a[7:0] should be held constant for four cycles of the main clock (ph1, ph2) and update synchronously with it.

Inputs	Description
a[7:0]	Input data
ph1	Main clock phase 1
ph2	Main clock phase 2
shiftClk1	Shift chain clock phase 1
shiftClk2	Shift chain clock phase 2
shiftIn	Shift in data to the coefficients
reset	Reset pin
Outputs	**Description**
y[17:0]	Filtered Output
Bidirectional	**Description**
vdd	Power
gnd	Ground

FIGURE 1.2: Table of chip inputs and outputs

1.2.2 Clocks

ph1 and ph2 are non-overlapping clocks that operate the filter. shiftClk1 and shiftClk2 are used to load the coefficients into the shift registers. The main clock (ph1, ph2) and the shift clock (shiftClk1, shiftClk2) can never operate at the same time. For example, if ph1 and ph2 are toggling, shiftClk1 and shiftClk2 should be held low. This constraint prevents asynchronous operation. Reset clears the accumulator, input data registers, and output register.

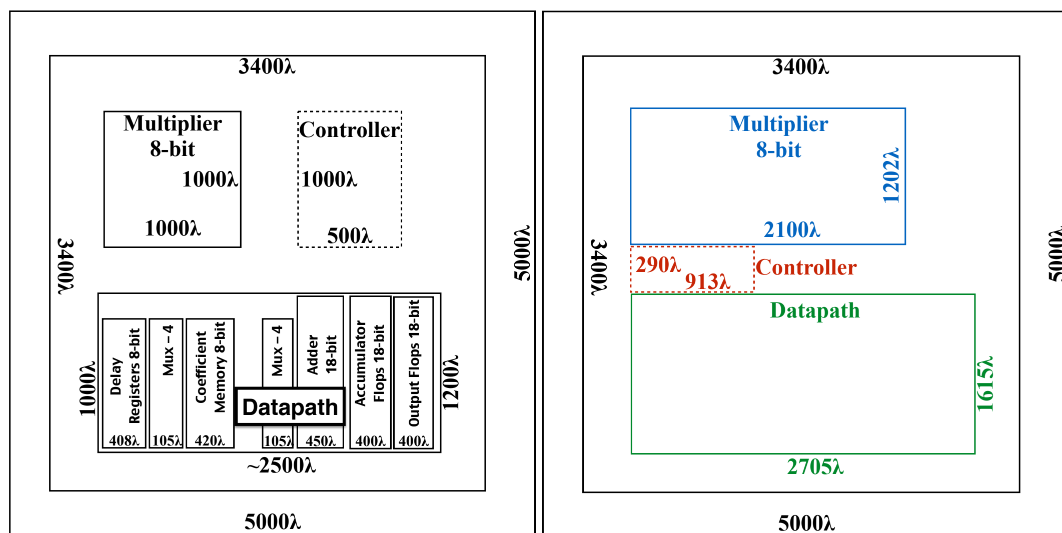
1.2.3 Shift in

The shiftIn input is used to load each bit of the coefficient data. This input should be updated synchronously with each cycle of the shift clock (shiftClk1, shiftClk2). The following input stream loads the coefficients c0 = 7, c1 = 8, c3 = 128, and c4 = 200 where $y = a[n] \cdot c0 + a[n-1] \cdot c1 + a[n-2] \cdot c2 + a[n-3] \cdot c3$. The most recent input bit is located to the left of the bit stream listed below

$$
\begin{aligned}
shiftIn &= c0, c1, c2, c3 \\
&= 8'b7, 8'b8, 8'b128, 8'b200 \\
&= 00000111_00001000_10000000_11001000 \\
&\leftarrow \text{most recent bit — oldest bit} \rightarrow
\end{aligned}
$$

1.3 Floorplan

There are three major portions of the design. The first is the datapath, which is described in Figure 1.1; the second is the synthesized controller; the third is the 8-bit unsigned multiplier. The datapath includes components such as the 8-bit registers, mux-4s, and an 18bit adder. The team estimated their sizes mainly using MIPS8 components from the mudd11 library, and came up with the estimated datapath size of 2500 x 1200 λ. The controller is fairly simple, so the team estimated its size at 1000 x 500 λ (half the mips8 controller). After researching 8-bit multipliers built by other designers, the team estimated the 8-bit multiplier to be 1000 x 1000 λ. These multipliers used radix 4 Booth encoding and were much smaller in area than our design. The team implemented an 8-bit unsigned multiplier using arrays of carry-save adders (CSA) and a carry propagate adder (CPA) described in the textbook. The team used a carry-lookahead adder modified from project 1 for the CPA. The comparison between the proposed floorplan and the final floorplan is shown below. Figure 1.3a is the floor plan of the proposal, and Figure 1.3b is the floor plan of the final design.



(A) Proposal floor plan (B) Final floor plan

FIGURE 1.3: Comparison of proposed floor plan and final floor plan

From the comparison above, it appears datapath was well-estimated. The final layout differed by 200 λ in width, and by 400 λ in height. One reason for the size difference was the implementation of the enable-resettable flops for the output signals. In the team's initial design, only resettable flops were used. Since the output signal flops operates at four-times the period than the input signal flops, the team originally planed to introduce a slower two-phase clock into the chip to trigger the output flops. However, a better design that uses enabled flops was implemented. As a result, the height of the datapah

increased. In addition, the final datapath layout includes surrounding power and ground rails.

The size of the controller turned out to be smaller than the estimation, which was what the team expected.

The size of the multiplier had the biggest change before and after the implementation of the chip. This is due to the fact that the team decided to switch design from a modified booth encoding radix-4 8-bit multiplier to a simpler CSA-CPA unsigned multiplier. The width of the multiplier turned out to be twice the original estimation. This is because the team moved the CPA adder to the right side of the multiplier instead of placing them below the CSA arrays. One reason that caused this change is that after the datapath was laid out and the controller was synthesized, the team realized there were not much space left in the height of the chip. Therefore, the team decided to move the CPA of the multiplier to the right side. As a result, the width of the multiplier increased to almost double the proposed plan. However, this provided a better placement for each component in the final layout of the chip.

Figure 1.4 contains the original sliceplan for the datapath and Figure 1.5 contains the actual sliceplan.

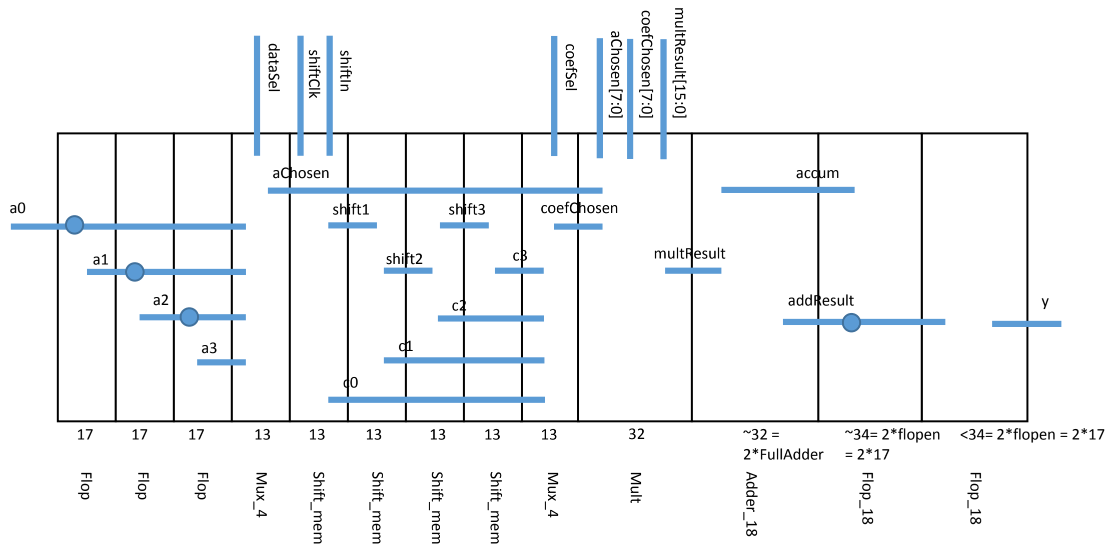


Figure 1.4: Original Sliceplan of the datapath

Figure 1.6 shows the assignments of the pins on the padframe:

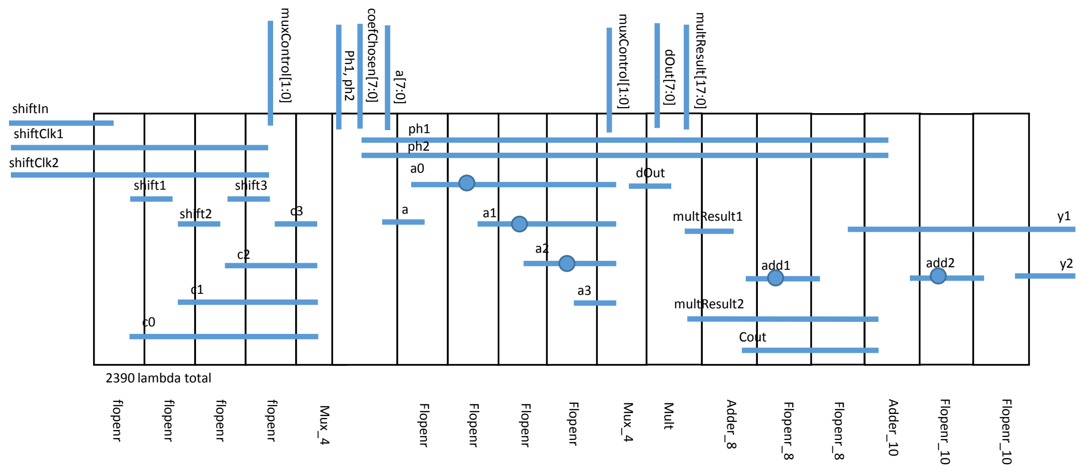


Figure 1.5: Updated Sliceplan of the datapath

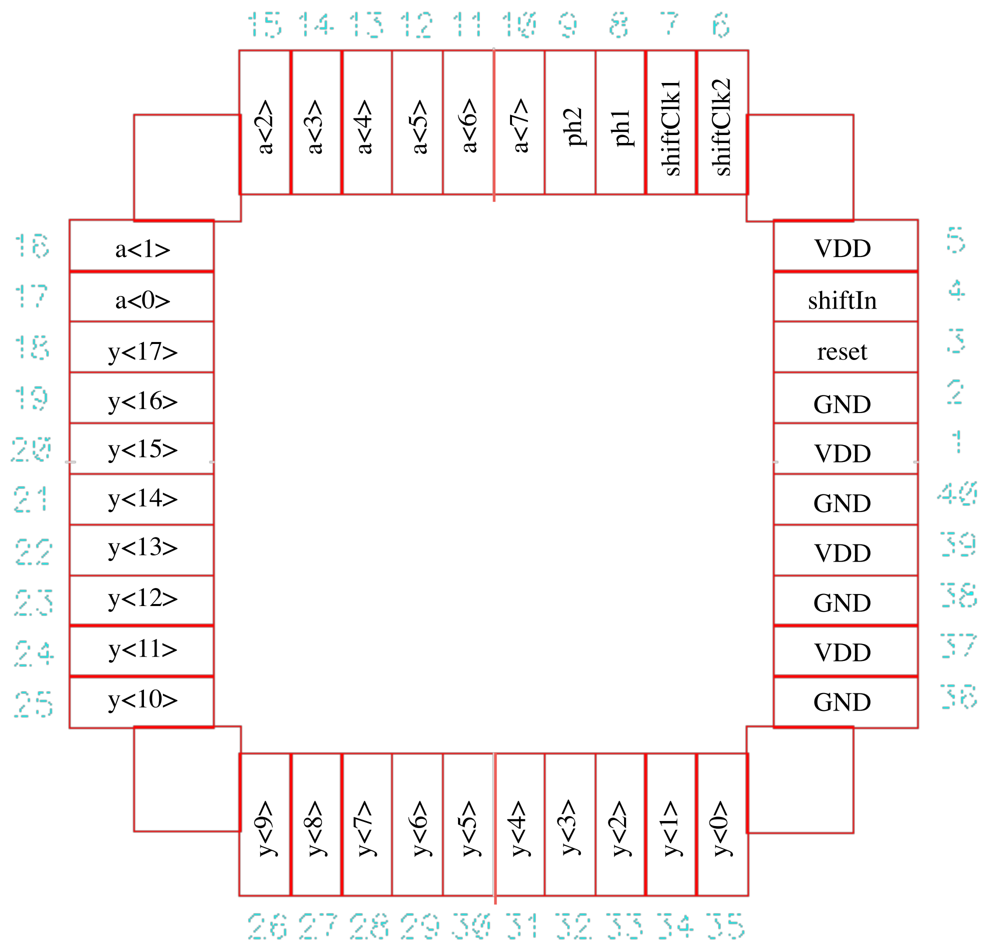


Figure 1.6: The pins assigned on the padframe

1.4 Verification

All components of our design passed our verification tests including the imported GDS file. The final schematic passes our verilog testbench and all layouts pass LVS and DRC. The verilog testbench uses directed and randomized tests. We tested the filter on randomized and directed sequences of a[7:0] and interesting coefficient values including zeros and all ones. The correct output for a given test vector was calculated by the testbench independently of the design, and the output was verified after each new input a[7:0]. We wrote scripts that generate the test vectors and shift sequences. These scripts and the verilog testbench are included in the appendices.

1.5 Postfabrication Test Plan

We plan to test the chip using the TestosterIC Device Under Test board. First, we will check the resistance between power and ground to make sure the chip isn't shorted. The resistance between power and ground should be much higher (10x +) than that between ground and ground. After checking the chip for shorts, we will use the TestosterIC DUT board to test the chip. We plan to convert our testbench into the appropriate format for the TestosterIC DUT board and setup a computer with the appropriate software to use the test board. If we are unable to set up the software and use the test board, then we will use an FPGA and bread board to stimulate the chip inputs and test it for functionality.

1.6 Design Time

The table below includes our design time for each portion of the project

Project Proposal	19 hours
Verilog	6 hours
Schematic	15 hours
Layout	18 hours
Total	**58 hours**

1.7 File Locations

The following is a list of all relevant file locations.

1. Verilog testbench and model:
   /home/mwaugaman/IC_CAD/cadence/e158proj2/testing/proj2.sv
2. Python script to generate testvectors:
   /home/mwaugaman/IC_CAD/cadence/e158proj2/testing/genTV.py
3. Synthesis results:
   /home/mwaugaman/IC_CAD/cadence/e158proj2/soc
4. Multiplier Library:
   /home/mwaugaman/IC_CAD/cadence/e158proj2/proj2_multiplier
5. Datpath, Controller, and Chip Library:
   /home/mwaugaman/IC_CAD/cadence/e158proj2/proj2
6. PDF of this report:
   /home/mwaugaman/IC_CAD/cadence/e158proj2/report/proj2.pdf
7. PDF chip plot:
   /home/mwaugaman/IC_CAD/cadence/e158proj2/report/proj2Plot.pdf
8. GDS file:
   /home/mwaugaman/IC_CAD/cadence/chip.gds

1.8 References

I. Huang, Da, Nassery, Afsaneh. "Modified Booth Encoding Radix-4 8-bit Multiplier. http://people.ee.duke.edu/ jmorizio/ece261/F08/projects/MULT.pdf

II. "Know About Modified Booth Algorithm". http://www.efxkits.us/modified-booth-algorithm/.

III. Minu, Thomas. "Design and Simulation of Radix-8 Booth Encoder Multiplier for Signed and Unsigned Numbers". http://www.ijirst.org/articles/IJIRSTV1I1008.pdf

Appendix A

Verilog Code

proj2.sv:

```
module datapath(input logic ph1, ph2, shiftIn, shiftClk1, shiftClk2,
        reset, enData, clearAccum,
        input logic[1:0]  muxControl,
        input logic[15:0] multR,
        input logic[7:0]  a,
        output logic[7:0] coef, dOut,
        output logic[17:0] y);
  `timescale 1ns / 100ps
  logic s1, s2, s3, cout;
  logic [7:0] a0, a1, a2, a3, c0, c1, c2, c3;
  logic [17:0] accumQ, accumD;

  // Mux's to select the coefficient and data
  mux4 #(8) m4d( a0, a1, a2, a3, muxControl, dOut);
  mux4 #(8) m4c(c0, c1, c2, c3, muxControl, coef);

  // 18 bit adder
  logic[7:0] test_multr;
  assign test_multr = {2'b00, multR[15:8]};
  adder_cout #(8) add8(multR[7:0], accumQ[7:0], accumD[7:0], cout);
  adder_cin #(10) add10({2'b00, multR[15:8]}, accumQ[17:8], cout, accumD
    [17:8]);

  // Data delay registers
  flopenr #(8) d0(ph1, ph2, reset, enData, a, a0);
  flopenr #(8) d1(ph1, ph2, reset, enData, a0, a1);
  flopenr #(8) d2(ph1, ph2, reset, enData, a1, a2);
  flopenr #(8) d3(ph1, ph2, reset, enData, a2, a3);
```

```
29    // Coefficient shift registers
      sflop_8 c0reg(shiftClk1, shiftClk2, shiftIn, c0);
31    sflop_8 c1reg(shiftClk1, shiftClk2, c0[7], c1);
      sflop_8 c2reg(shiftClk1, shiftClk2, c1[7], c2);
33    sflop_8 c3reg(shiftClk1, shiftClk2, c2[7], c3);

35    // Accumulate register
      flopr #(8) acc8(ph1, ph2, clearAccum, accumD[7:0], accumQ[7:0]);
37    flopr #(10) acc10(ph1, ph2, clearAccum, accumD[17:8], accumQ[17:8]);

39    // Output register
      flopenr #(8) r8(ph1, ph2, reset, enData, accumD[7:0], y[7:0]);
41    flopenr #(10) r10(ph1, ph2, reset, enData, accumD[17:8], y[17:8]);

43  endmodule

45
    module top(input logic ph1, ph2, shiftClk1, shiftClk2,
47                 shiftIn, reset,
               input logic [7:0] a,
49             output logic [17:0] y);

51      logic clearAccum, enData;
        logic[1:0] muxControl;
53      logic[7:0] coef, dOut;
        logic[15:0] multR;
55
        datapath dp(.*);
57      controller c(.*);
        multiplier #(8) m(coef, dOut, multR);
59  endmodule

61  module adder
      #(parameter width = 18)
63    (input logic [width-1:0] a, b,
      output logic [width-1:0] y);
65
      assign y = a + b;
67  endmodule

69  module adder_cin
      #(parameter width = 18)
71    (input logic [width-1:0] a, b,
      input logic cin,
73    output logic [width-1:0] y);

75    assign y = a + b + cin;
    endmodule
```

```

module adder_cout
  #(parameter width = 18)
  (input logic [width-1:0] a, b,
  output logic [width-1:0] y,
  output logic cout);

  assign {cout, y} = a + b;
endmodule

// states and instructions
typedef enum logic [1:0] {STATE_1 = 2'b00,
              STATE_2 = 2'b01,
              STATE_3 = 2'b10,
              STATE_4 = 2'b11
              } statetype;

module controller(input logic ph1, ph2, reset,
                  output logic enData, clearAccum,
                  output logic[1:0] muxControl);


  // State registers
  statetype state;

  statelogic statelog(ph1, ph2, reset, state);
  outputlogic outputlog(state, ph2, enData, clearAccum, muxControl);

endmodule


module statelogic(input  logic     ph1, ph2, reset,
                  output statetype state);


      statetype nextstate;
      logic [1:0] state_logic;
      //
    flopr #(2) statereg(ph1, ph2, reset, nextstate, state_logic);
    assign state = statetype'(state_logic);

    // Next state logic
    always_comb
      begin
        case (state)
          STATE_1 : nextstate = STATE_2 ;
        STATE_2 : nextstate = STATE_3 ;
        STATE_3 : nextstate = STATE_4 ;
```

```
125         STATE_4 : nextstate = STATE_1 ;
              default: nextstate = STATE_1; // should never happen
127         endcase
          end
129 endmodule

131 module outputlogic(input statetype state,
               input logic ph2,
133                        output logic enData, clearAccum,
                           output logic[1:0] muxControl);
135
      // Output logic
137   always_comb
        begin
139     case (state)
          STATE_1 :
141       begin
            muxControl = 2'b00;
143         enData = 0;
            clearAccum = 0;
145       end
          STATE_2 :
147       begin
            muxControl = 2'b01;
149         enData = 0;
            clearAccum = 0;
151       end
          STATE_3 :
153       begin
            muxControl = 2'b10;
155         enData = 0;
            clearAccum = 0;
157       end
          STATE_4 :
159       begin
            muxControl = 2'b11;
161         enData = 1;
            clearAccum = 1;
163       end
      default: muxControl = 2'b00;
165     endcase
        end
167 endmodule

169 module flop #(parameter WIDTH = 8)
                  (input  logic             ph1, ph2,
171                input  logic [WIDTH-1:0] d,
                   output logic [WIDTH-1:0] q);
```

```
173

175   logic [WIDTH-1:0] mid;

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

181 module flopr #(parameter WIDTH = 8)
                 (input  logic             ph1, ph2, reset,
183               input  logic [WIDTH-1:0] d,
                  output logic [WIDTH-1:0] q);
185
      logic [WIDTH-1:0] d2, resetval;
187
      assign resetval = 0;
189
      mux2 #(WIDTH) enrmux(d, resetval, reset, d2);
191   flop #(WIDTH) f(ph1, ph2, d2, q);
    endmodule
193
    module flopenr #(parameter WIDTH = 8)
195                (input  logic             ph1, ph2, reset, en,
                    input  logic [WIDTH-1:0] d,
197                 output logic [WIDTH-1:0] q);

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

201   assign resetval = 0;

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

207 module latch #(parameter WIDTH = 8)
               (input  logic             ph,
209             input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);
211
      always_latch
213     if (ph) q <= d;
    endmodule
215

217 module multiplier
      #(parameter width = 8)
219   (input logic [width-1:0] a, b,
      output logic [2*width-1:0] y);
```

```
221   assign y = a*b;
      // Write out the custom logic
223
    endmodule
225

227 module mux2
      #(parameter width = 8)
229   (input logic [width-1:0] d0, d1,
      input logic s,
231   output logic [width-1:0] y);

233   assign y = s ? d1 : d0;
    endmodule
235
    module mux4
237   #(parameter width = 8)
      (input logic [width-1:0] d0, d1, d2, d3,
239   input logic [1:0] s,
      output logic [width-1:0] y);
241
      logic [width-1:0] low, hi;
243   mux2 #(width) lowmux(d0, d1, s[0], low);
      mux2 #(width) himux(d2, d3, s[0], hi);
245   mux2 #(width) outmux(low, hi, s[1], y);
    endmodule
247
    module mux3 #(parameter WIDTH = 8)
249               (input  logic [WIDTH-1:0] d0, d1, d2,
                   input  logic [1:0]       s,
251                output logic [WIDTH-1:0] y);

253   always_comb
        casez (s)
255       2'b00: y = d0;
          2'b01: y = d1;
257       2'b1?: y = d2;
        endcase
259 endmodule

261 module sflop_8
      (input logic clk1, clk2,
263    input logic sin,
       output logic[7:0] q);
265
      logic[7:0] mid;
267   flop #(1) sreg0(clk1, clk2, sin,  q[0]);
      flop #(1) sreg1(clk1, clk2, q[0], q[1]);
```

```
  flop #(1) sreg2(clk1, clk2, q[1], q[2]);
  flop #(1) sreg3(clk1, clk2, q[2], q[3]);
  flop #(1) sreg4(clk1, clk2, q[3], q[4]);
  flop #(1) sreg5(clk1, clk2, q[4], q[5]);
  flop #(1) sreg6(clk1, clk2, q[5], q[6]);
  flop #(1) sreg7(clk1, clk2, q[6], q[7]);
endmodule
```

testbench.sv:

```
module testbench();

  logic ph1, ph2, reset, shiftIn, shiftClk1, shiftClk2;
  logic [7:0] a, a0, a1, a2, a3, testa;
  logic [17:0] y;
  logic [17:0] result;

  // Example testvector
  // tv =       0    _    0     _00000000
  //      shiftClkEn, shiftIn,      a

  // Store the list of testvectors
  logic [9:0] testvector[2097151:0];
  logic [15:0] vecnum; // index of the current testvector
  logic[9:0] tv, currVec, t0, t1, t2, t3;
  logic [31:0] errors;
  logic [31:0] correct;

  // Calculate the correct value given the test vectors
  always_comb
  begin
    currVec = testvector[vecnum];
    t0 = testvector[vecnum -8];
    t1 = testvector[vecnum -12];
    t2 = testvector[vecnum -16];
    t3 = testvector[vecnum -20];
    a0 = t0[7:0];
    a1 = t1[7:0];
    a2 = t2[7:0];
    a3 = t3[7:0];
    result = dut.dp.c0 * a0 + dut.dp.c1 * a1 + dut.dp.c2 * a2 + dut.dp.c3 *
     a3;
  end

  // instantiate device to be tested
```

```
35  top dut(.*);

37  // initialize test
    initial
39    begin
        // C:\Users\maxwaug\Google Drive\E 158\proj2\SourceTree\testing
41      // $readmemb("C:/Users/maxwaug/Google Drive/E 158/proj2/SourceTree/
      testing/t1.v", testvector);
        $readmemb("D:/Max/Google Drive/E 158/proj2/SourceTree/testing/t1.v",
      testvector);
43      vecnum = 0;
        errors = 0;
45      correct = 0;
        reset <= 1; # 20; reset <= 0;
47    end

49  // generate clock to sequence tests
    always
51    begin
       ph1 = 0; ph2 = 0; #1;
53     ph1 = 1; # 4;
       ph1 = 0; #1;
55     ph2 = 1; # 4;
      end
57
    // Check results on each new data cycle
59  always @(posedge dut.dp.enData )
    begin
61    if(vecnum > (20 + 42)) begin
        if( result !== y ) begin
63        $display("Expected %d, actual %d", result, y);
          $display("c0 %d, c1 %d, c2 %d, c3 %d",
65          dut.dp.c0, dut.dp.c1, dut.dp.c2, dut.dp.c3);
          $display("a0 %d, a1 %d, a2 %d, a3 %d", a0, a1, a2, a3);
67        $display("vecnum %d",vecnum);
          $display("@%0dps",$time);
69        errors = errors +1;
        end else begin
71        correct = correct + 1;
        end
73    end
    end
75
    logic shiftClkEn;
77  // Make the second shift clock follow the first
    always_comb
79  begin
      shiftClk1 = shiftClkEn & ph1;
```

```
        shiftClk2 = shiftClkEn & ph2;
    end

    // Increment through the testvectors
    always @(negedge ph1)
    begin
      if(vecnum > 2000)begin
        $display("correct %d", correct);
        $display("incorrect %d", errors);
        $display("vecnum %d", vecnum);
        $finish;
      end else begin
        tv = testvector[vecnum];
        a = tv[7:0];
        shiftIn = tv[8];
        shiftClkEn = tv[9];
        vecnum = vecnum +1;
      end
    end

endmodule
```

(part of) testvectors:

```
0_0_00000000
0_0_00000000
0_0_00000000
0_0_00000000
0_0_00000000
0_0_00000000
0_0_00000000
0_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_1_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_1_00000000
```

```
1_0_00000000
1_1_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_1_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_0_00000000
1_1_00000000
1_0_00000000
1_1_00000000
0_0_01101101
0_0_01101101
0_0_01101101
0_0_01101101
0_0_00111111
0_0_00111111
0_0_00111111
0_0_00111111
0_0_10011011
0_0_10011011
0_0_10011011
0_0_10011011
0_0_10100000
0_0_10100000
0_0_10100000
0_0_10100000
0_0_11100101
0_0_11100101
0_0_11100101
0_0_11100101
0_0_10110001
0_0_10110001
0_0_10110001
0_0_10110001
0_0_01011001
0_0_01011001
0_0_01011001
0_0_01011001
0_0_10000011
```

```
0_0_10000011
0_0_10000011
0_0_10000011
0_0_10111110
0_0_10111110
0_0_10111110
0_0_10111110
0_0_01001011
0_0_01001011
0_0_01001011
0_0_01001011
0_0_01111000
0_0_01111000
0_0_01111000
0_0_01111000
0_0_01111111
0_0_01111111
0_0_01111111
0_0_01111111
0_0_00001100
0_0_00001100
0_0_00001100
0_0_00001100
0_0_11010100
0_0_11010100
0_0_11010100
0_0_11010100
0_0_00110110
0_0_00110110
0_0_00110110
0_0_00110110
0_0_00011001
0_0_00011001
0_0_00011001
0_0_00011001
0_0_01000110
0_0_01000110
0_0_01000110
0_0_01000110
0_0_01000011
0_0_01000011
0_0_01000011
0_0_01000011
0_0_00100001
```

Appendix B

Schematics

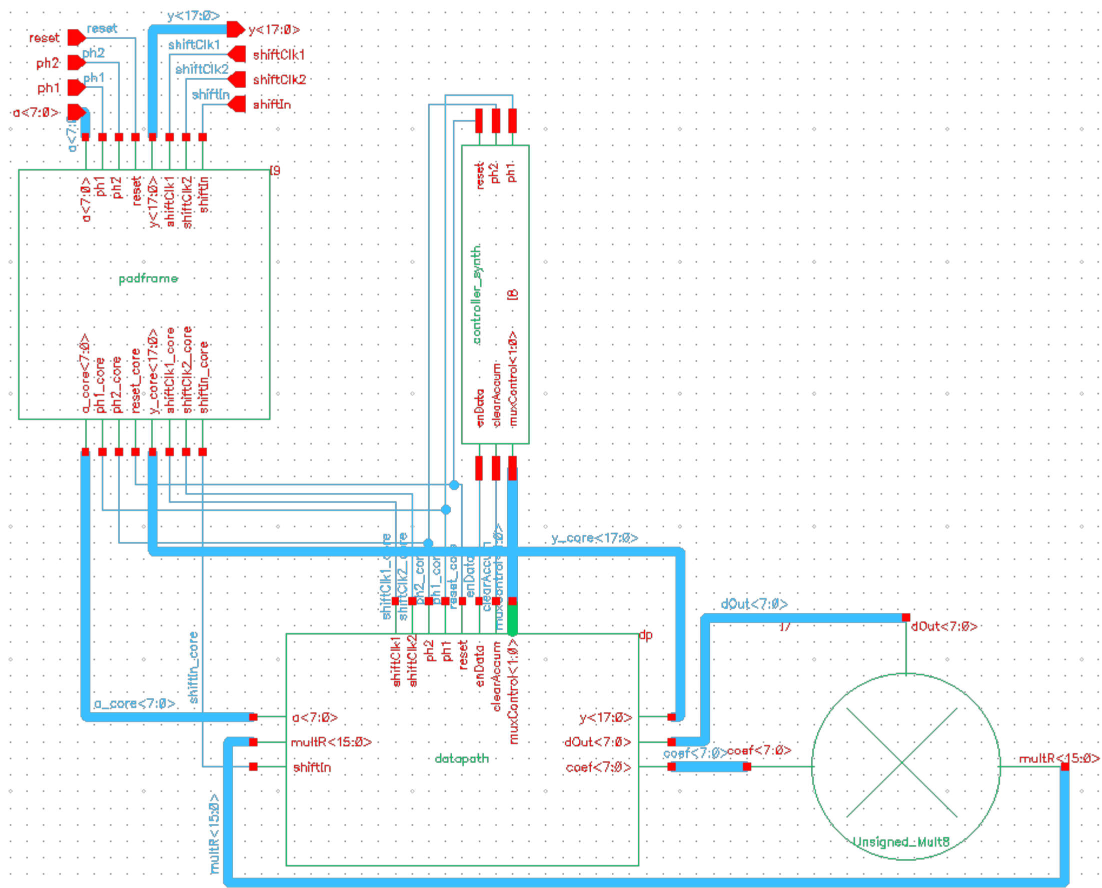


Figure B.1: Chip schematics

Figure B.2: Padframe schematics

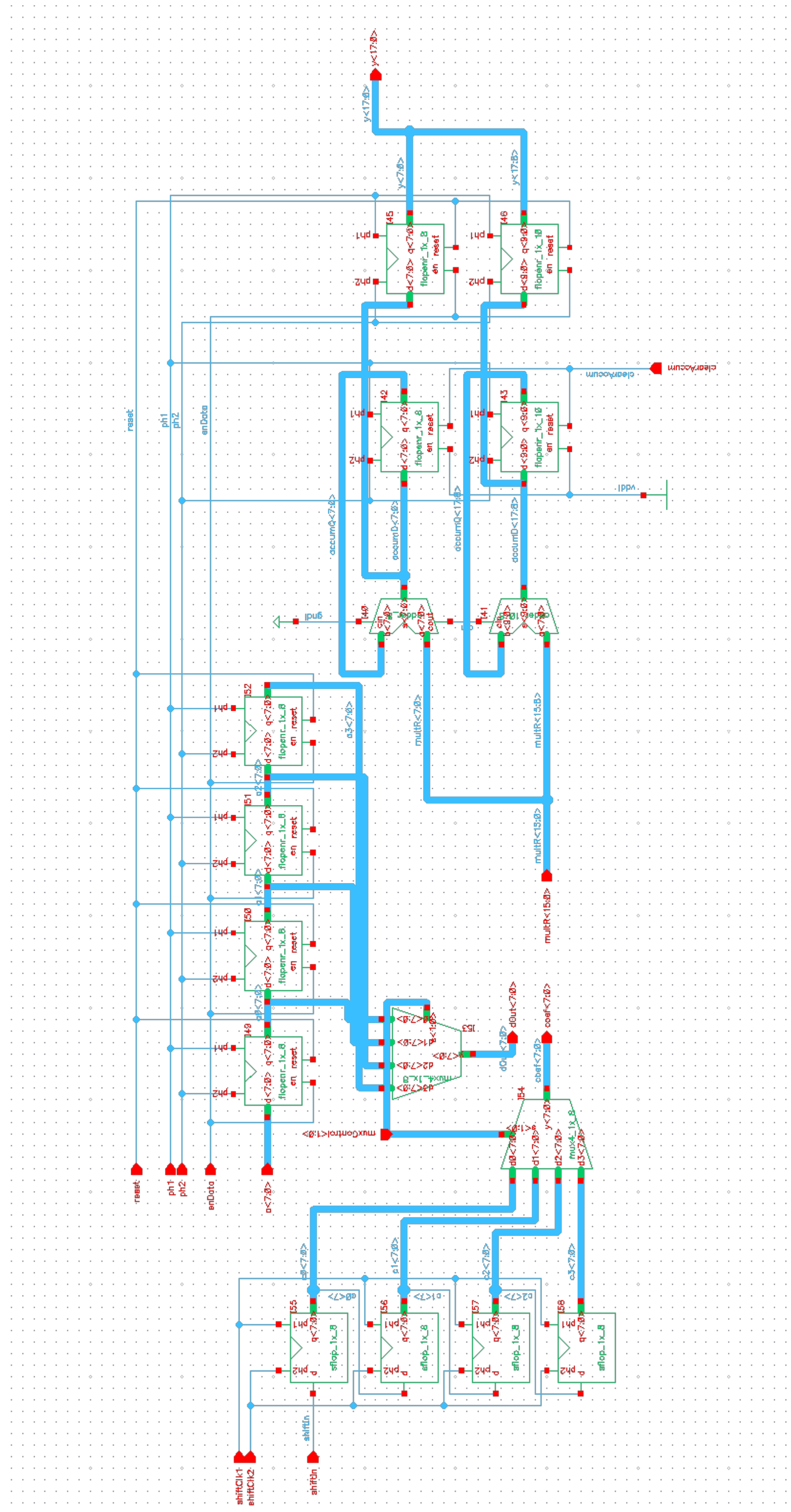


FIGURE B.3: Datapath schematics

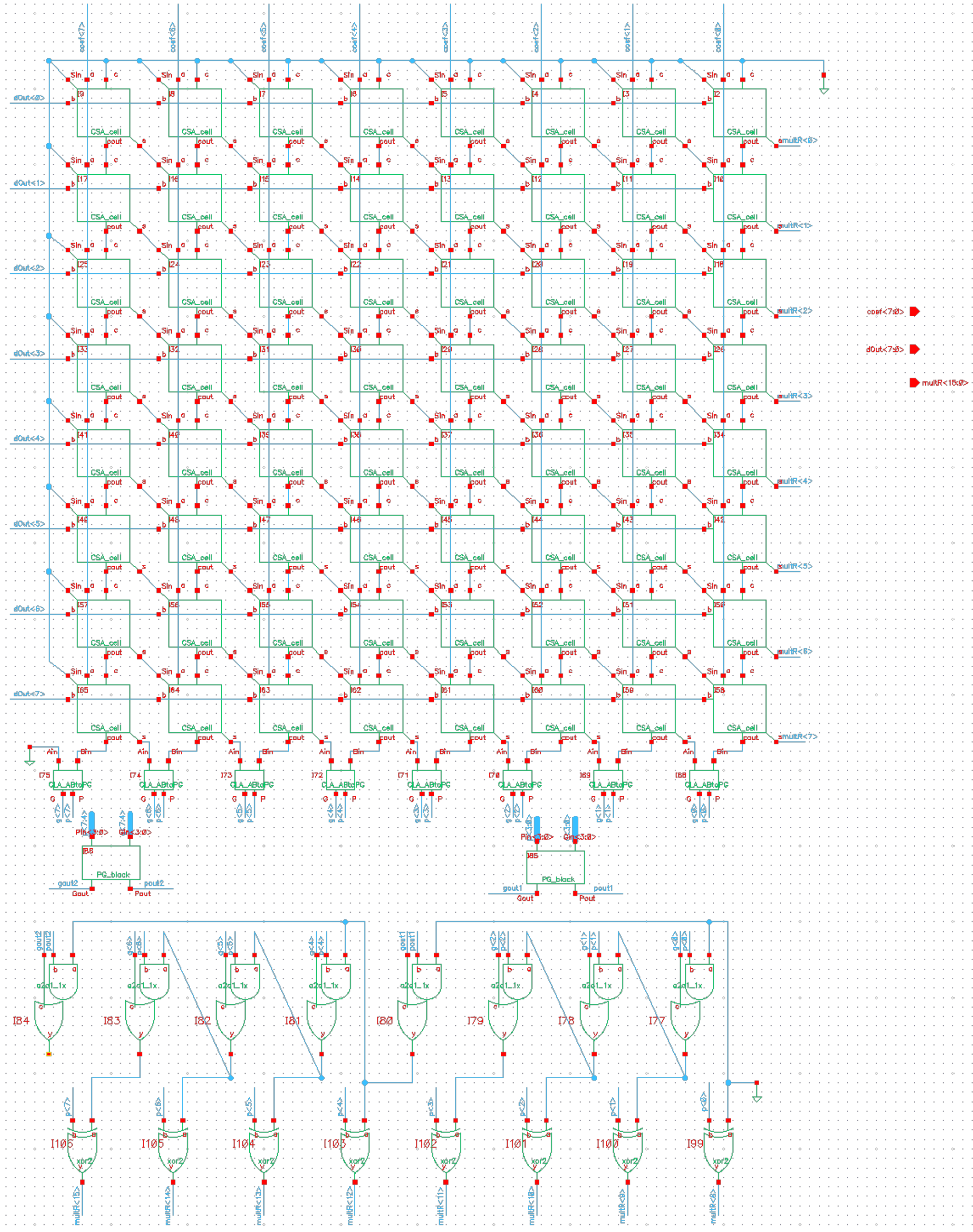


Figure B.4: 8-bit multiplier schematics

Figure B.5: Synthesize controller schematics

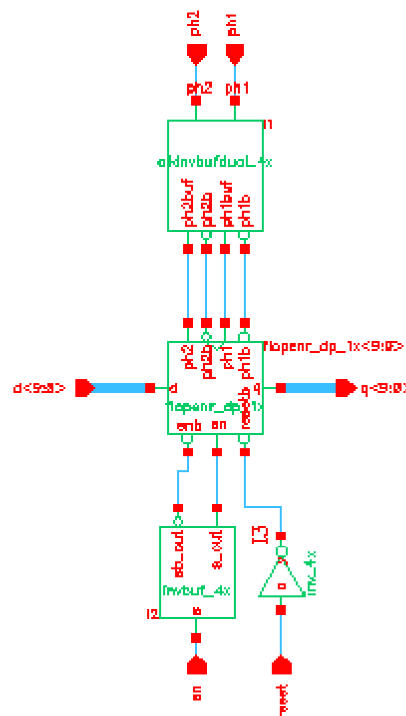


Figure B.6: 10-bit enable flop schematics

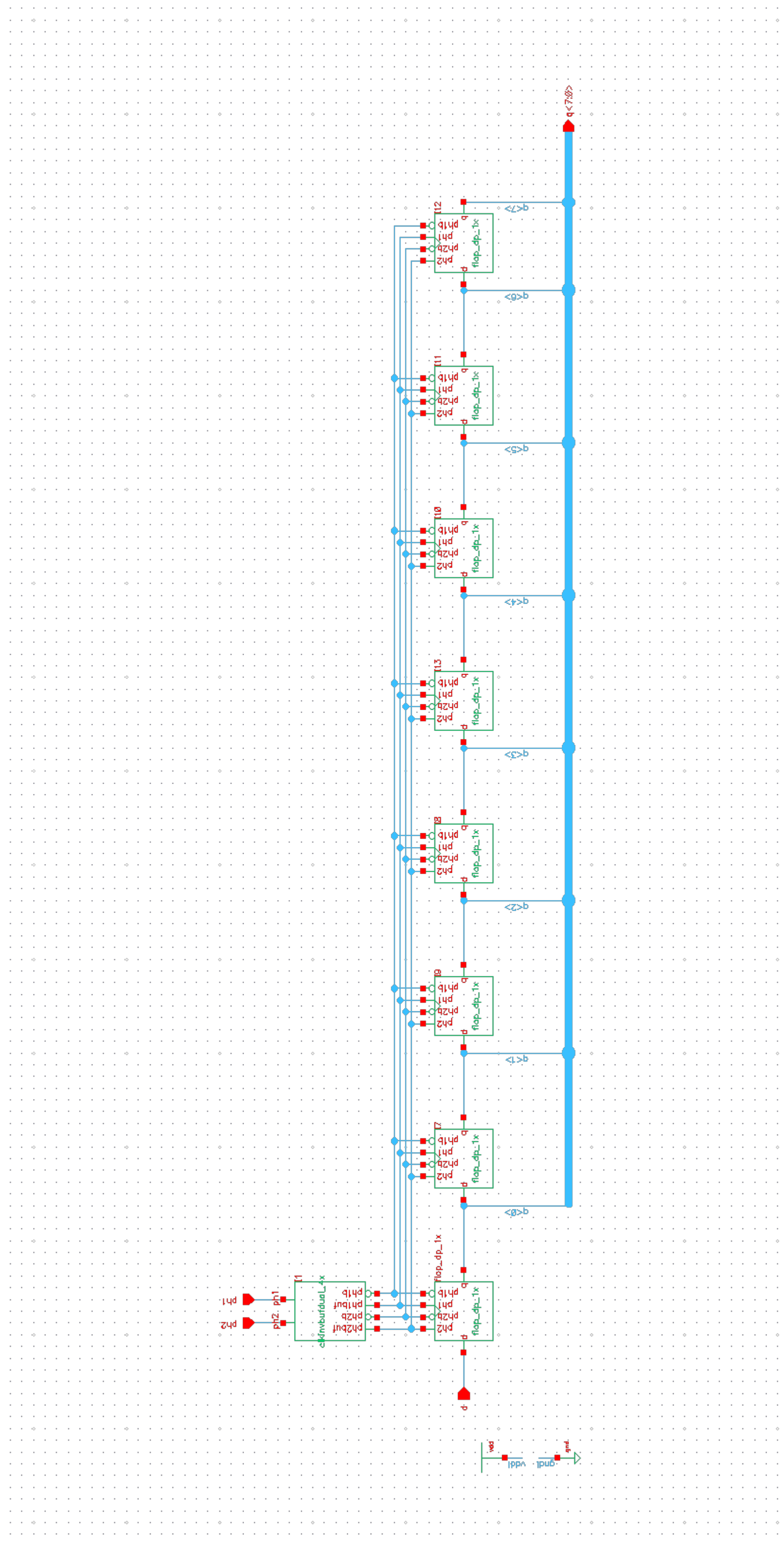


Figure B.7: 8-bit resettable flop schematics

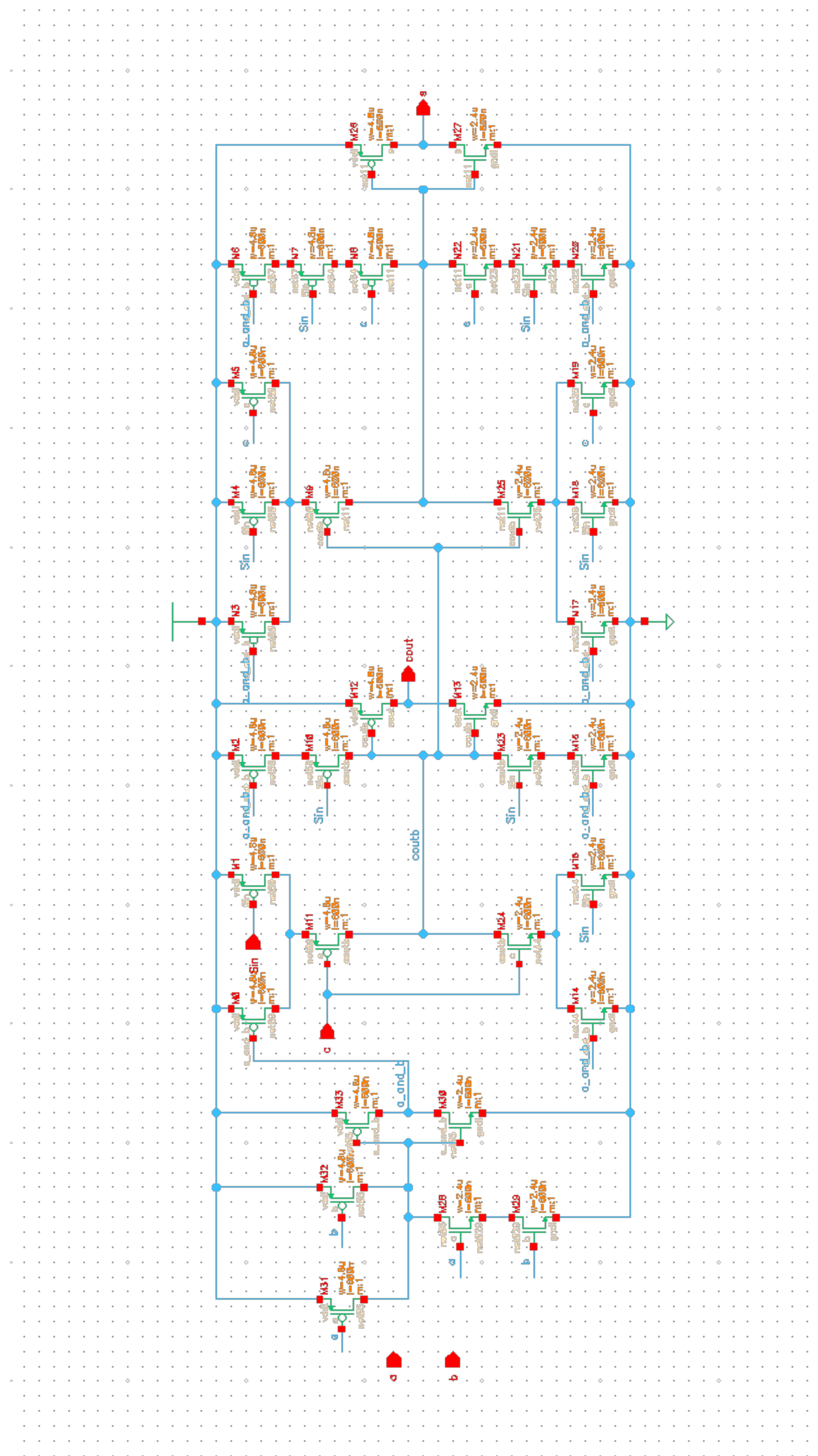


Figure B.8: CSA cell schematics

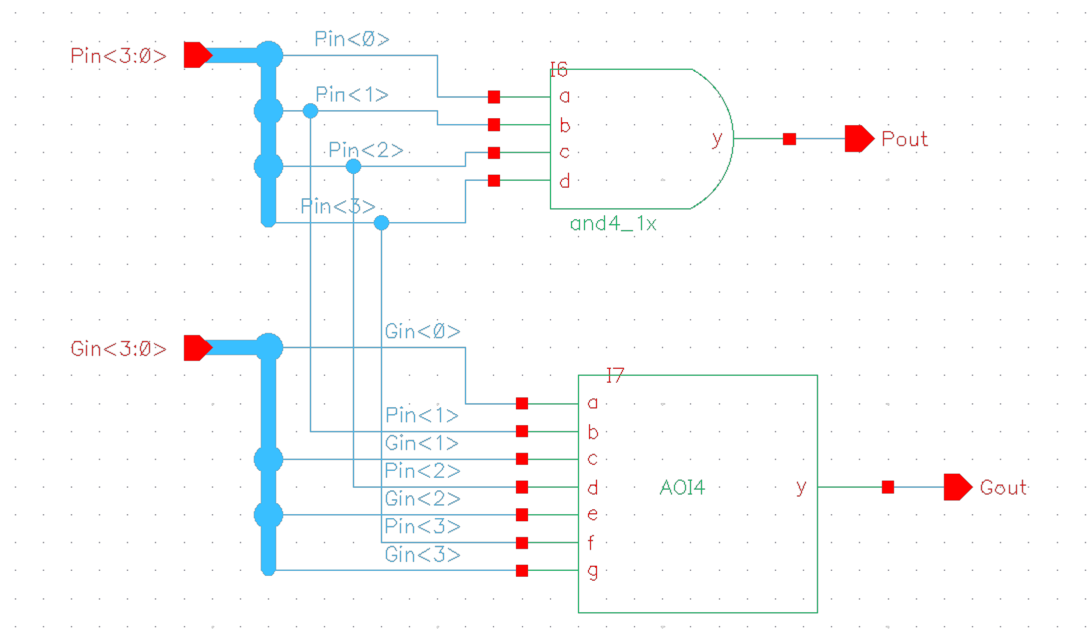


FIGURE B.9: PG cell schematics

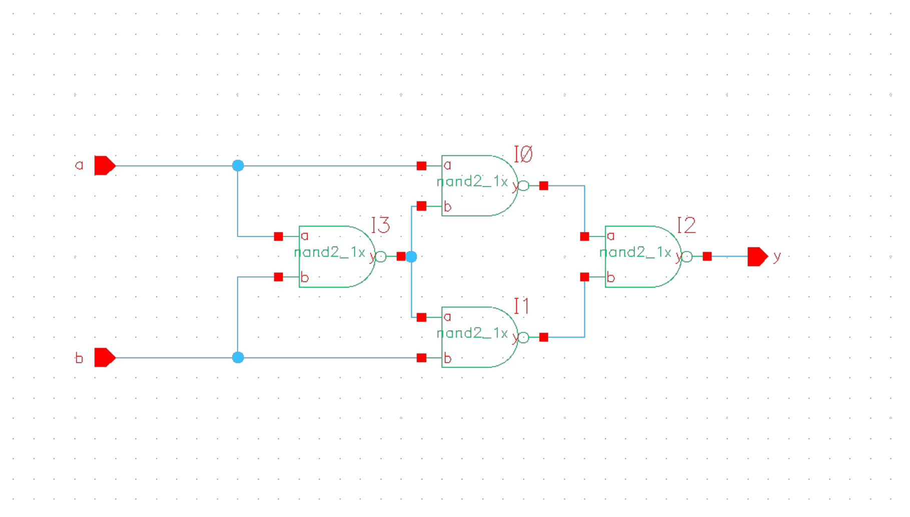


FIGURE B.10: XOR2 schematics

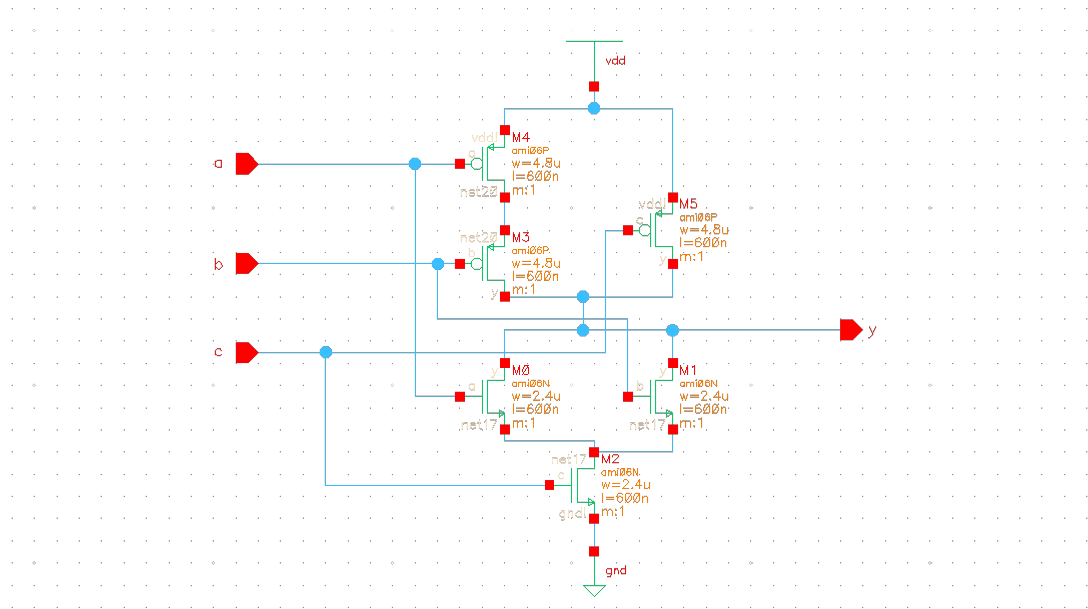


FIGURE B.11: OAI schematics

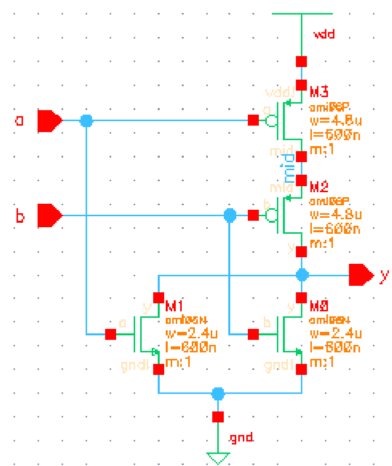


FIGURE B.12: NOR2 schematics

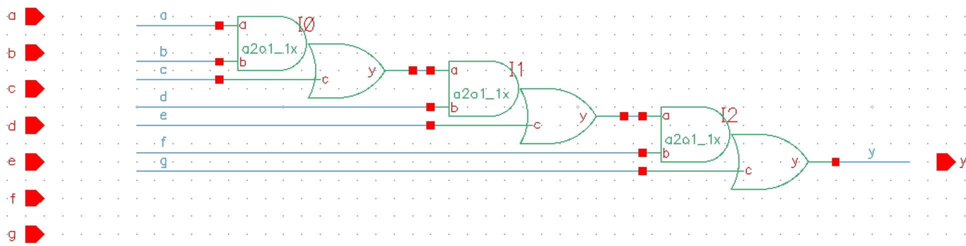


FIGURE B.13: AOI4 schematics

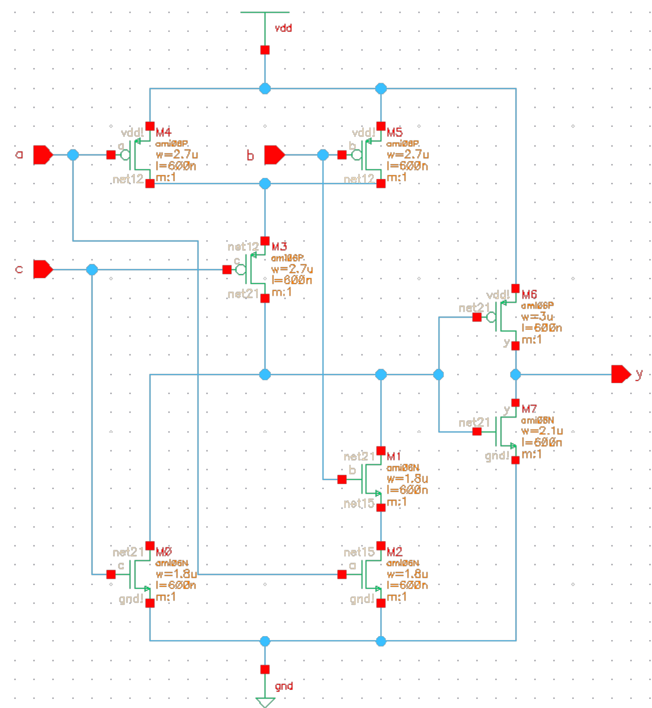


FIGURE B.14: A2o1_1x schematics

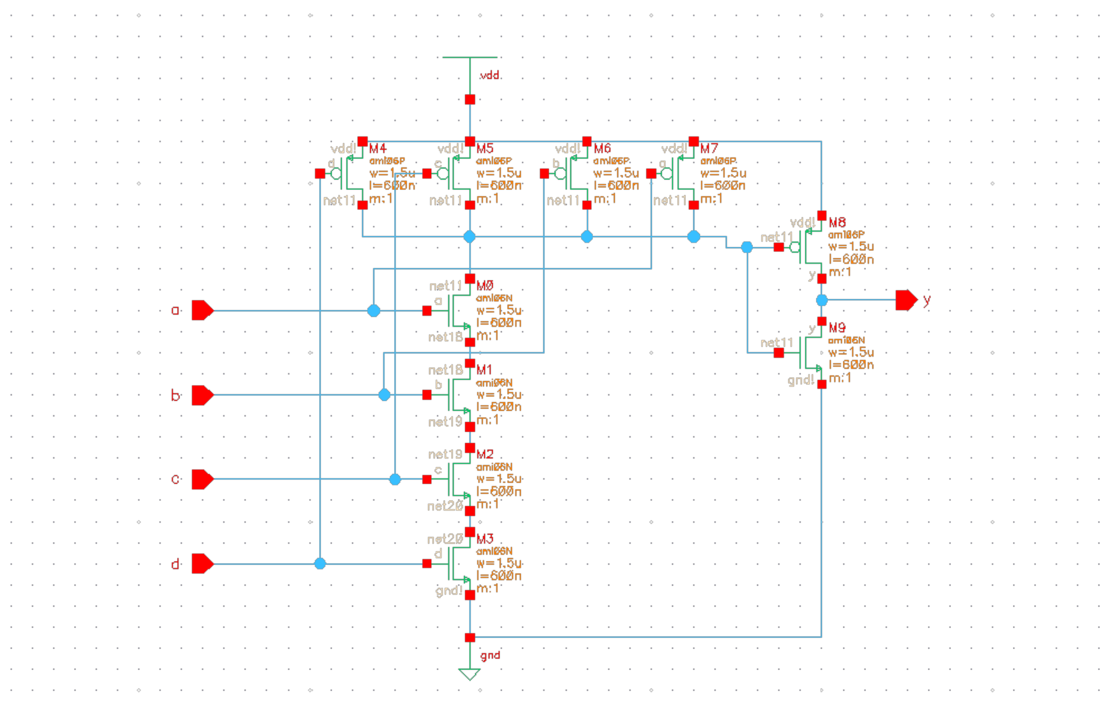


FIGURE B.15: AND4 schematics

Appendix C

Layout

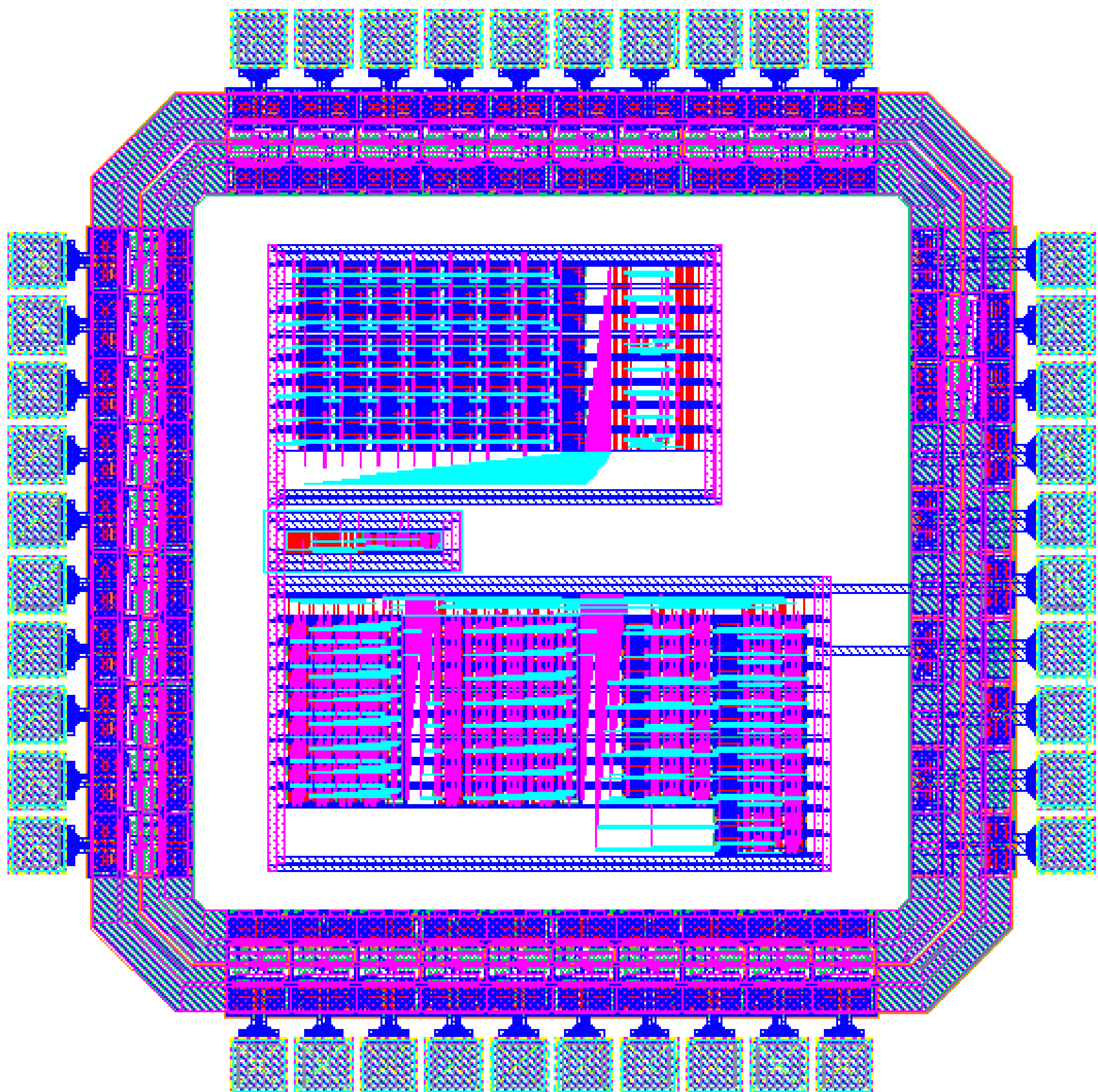


Figure C.1: Chip layout

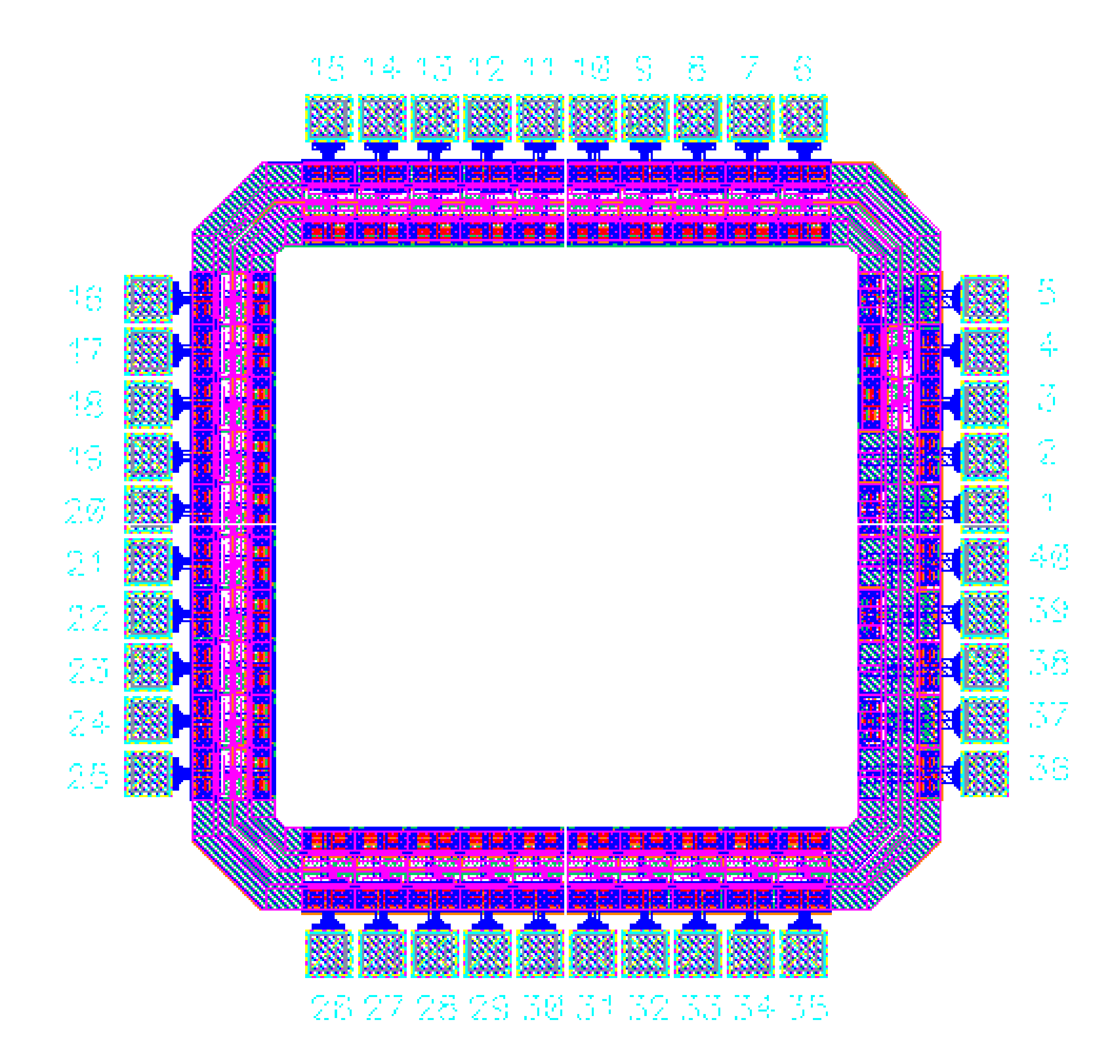


FIGURE C.2: Padframe layout

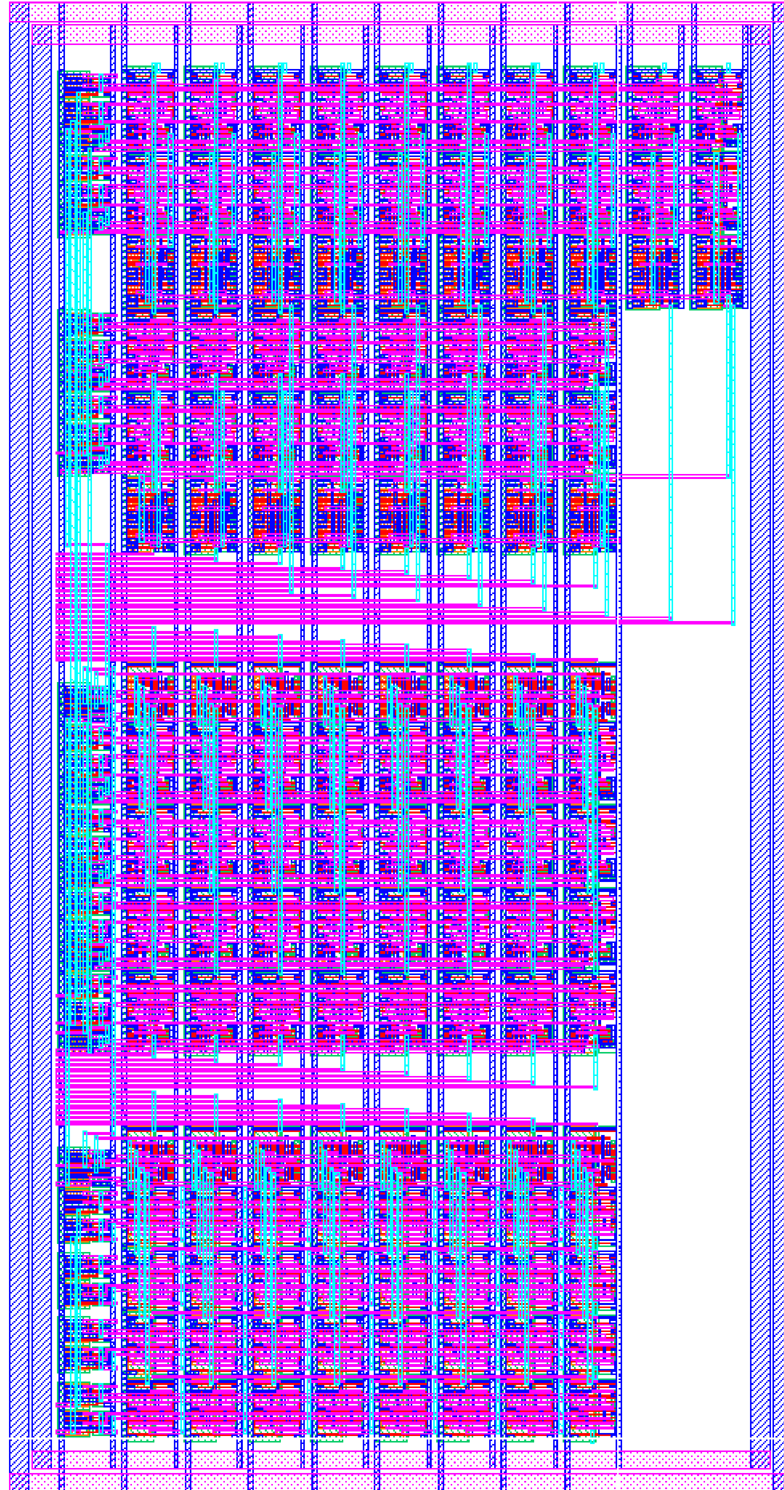


FIGURE C.3: Datapath layout

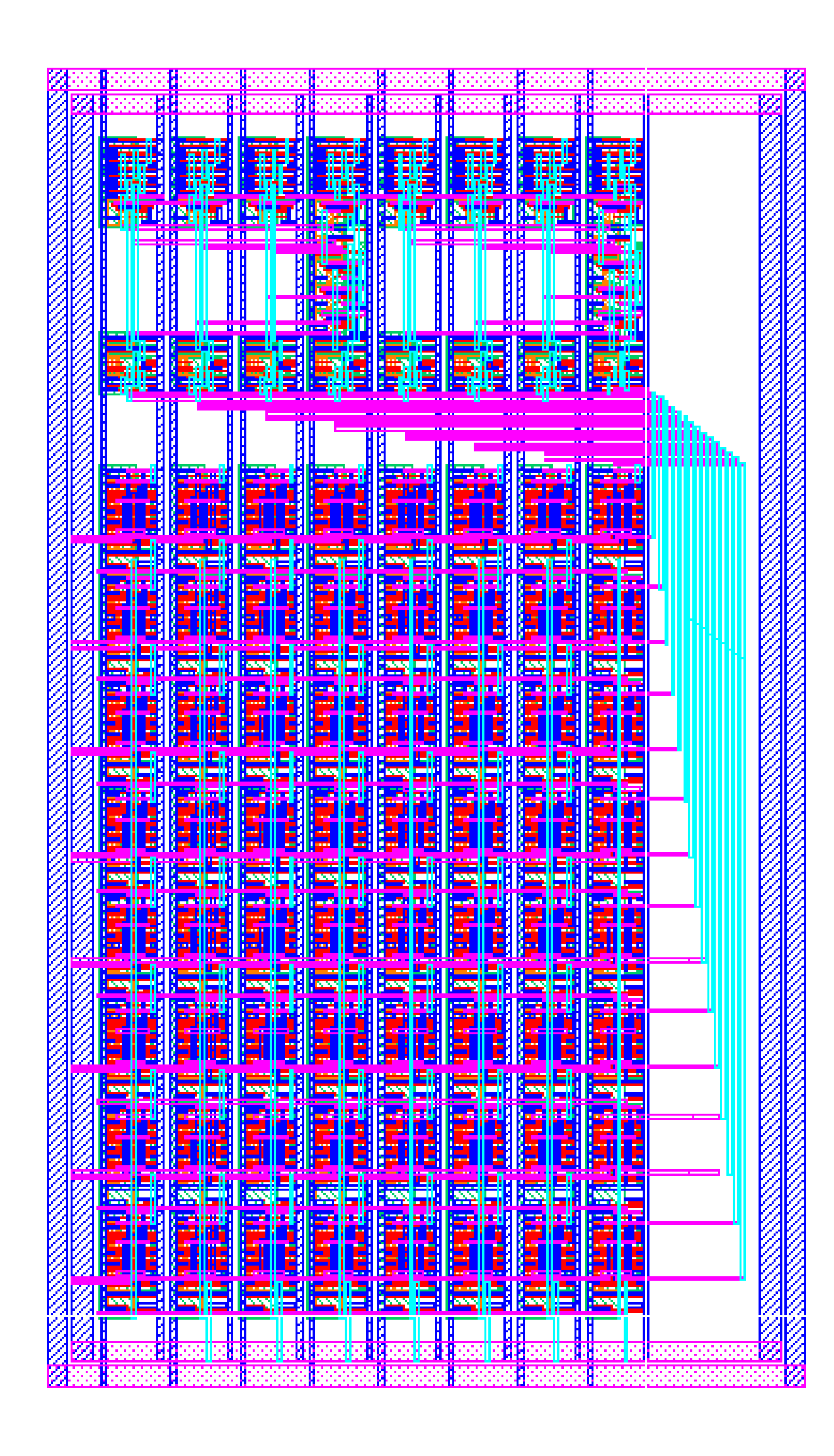


FIGURE C.4: 8-bit multiplier layout

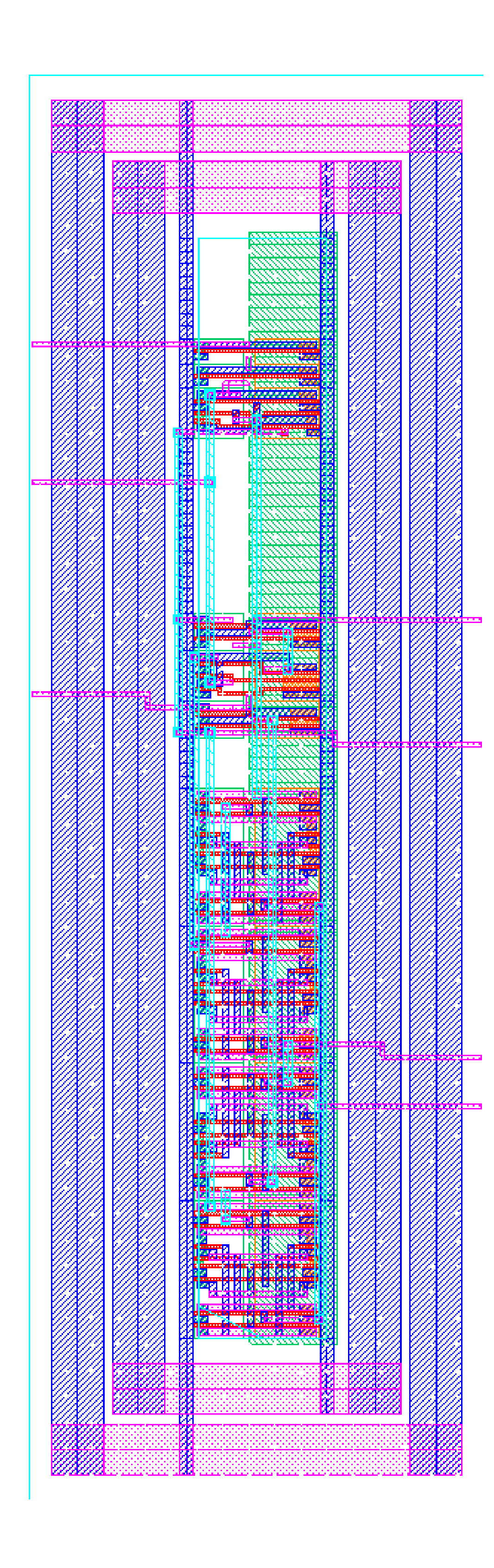


Figure C.5: Synthesize controller layout

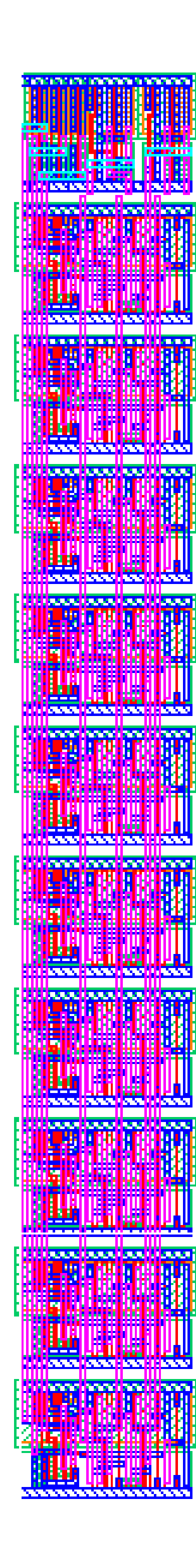


Figure C.6: 10-bit enable flop layout

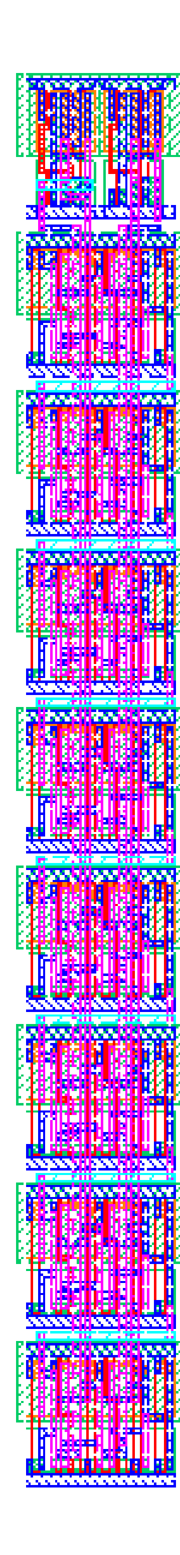


FIGURE C.7: 8-bit resettable flop layout

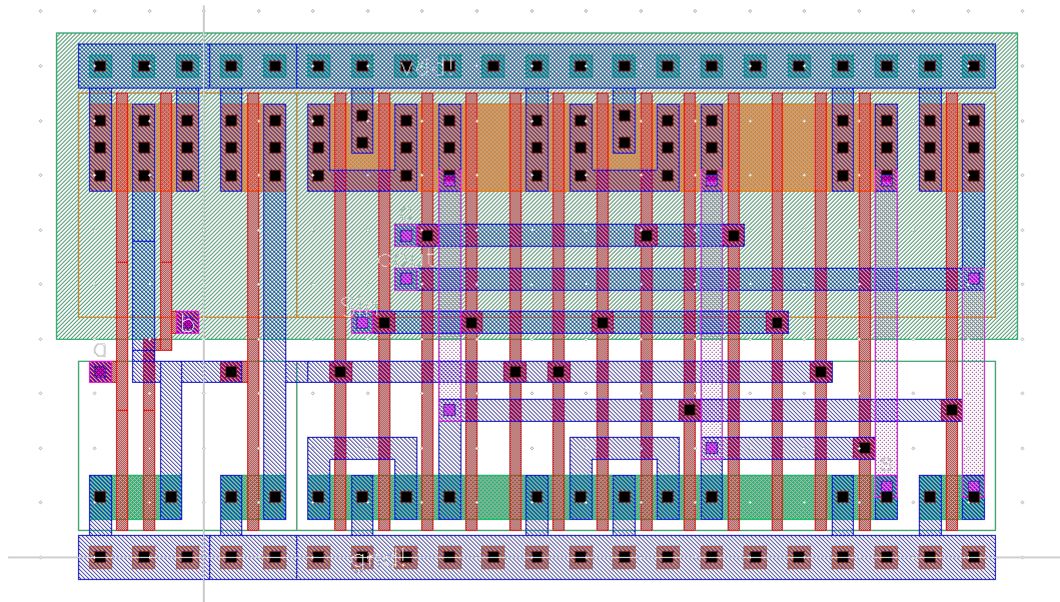


FIGURE C.8: CSA cell layout

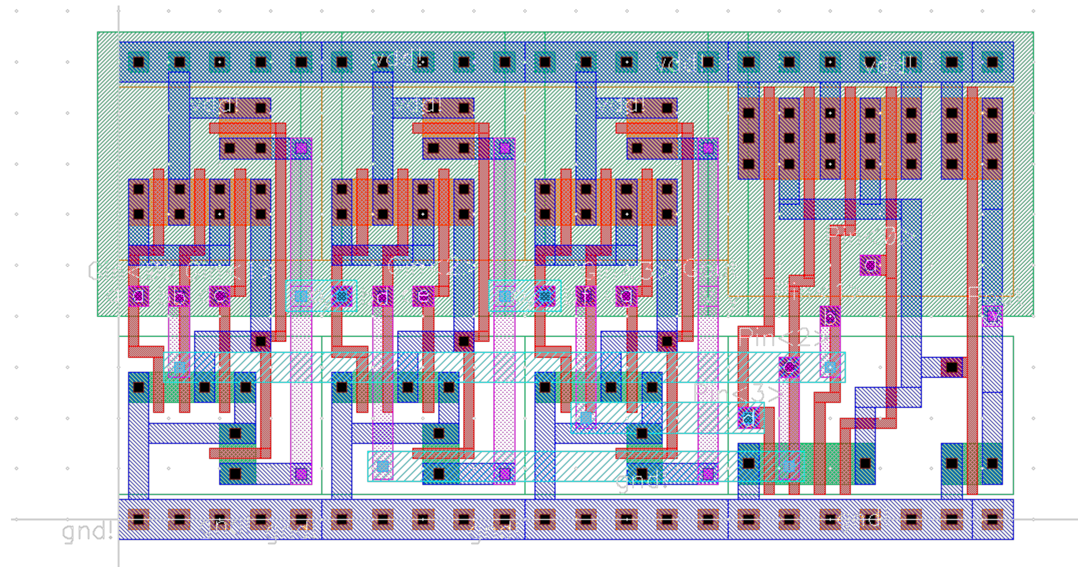


Figure C.9: PG cell layout

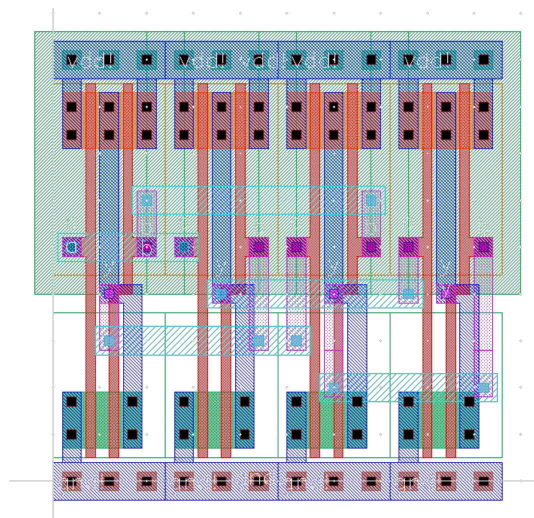


Figure C.10: XOR2 layout

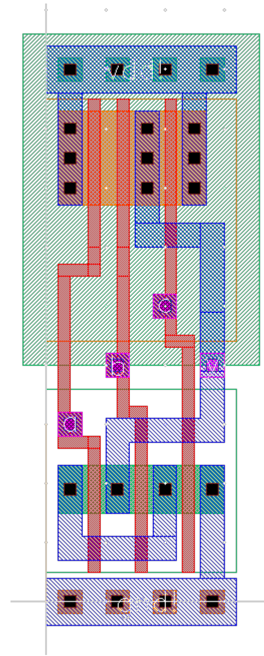


FIGURE C.11: OAI layout

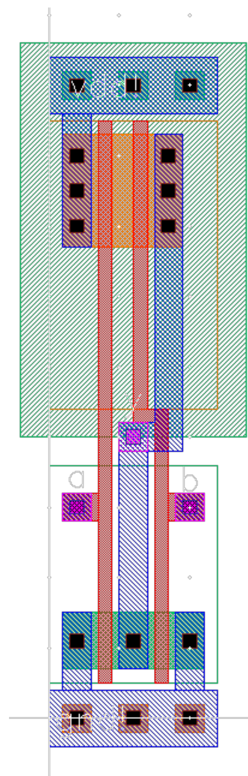


FIGURE C.12: NOR2 layout

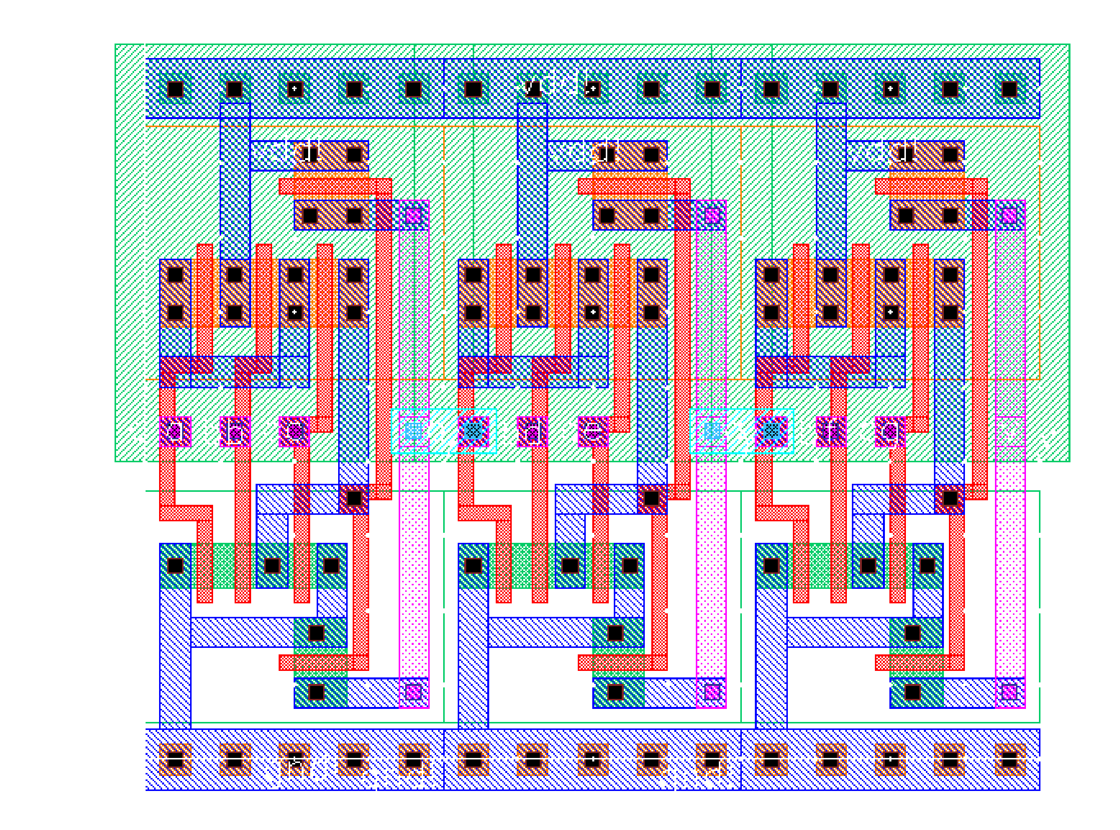


Figure C.13: AOI4 layout

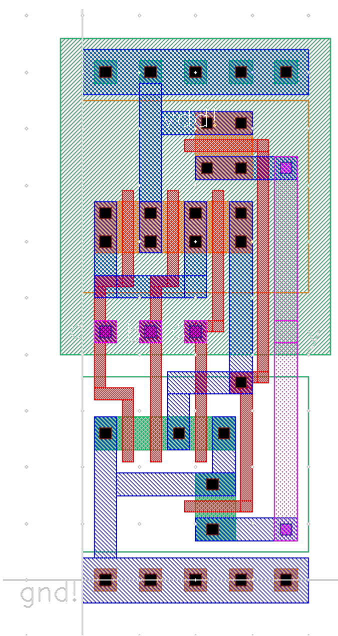


Figure C.14: A2o1_1x layout

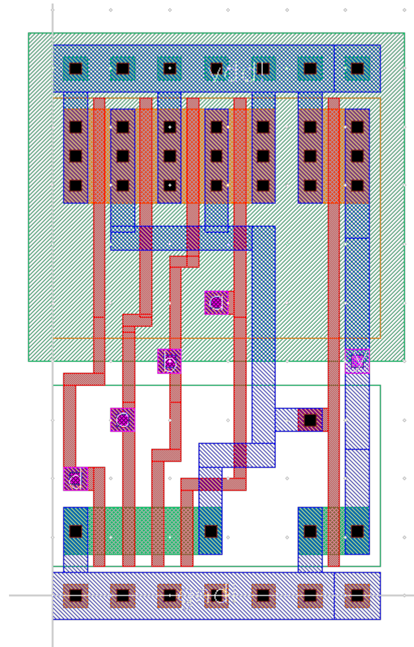


FIGURE C.15: AND4 layout