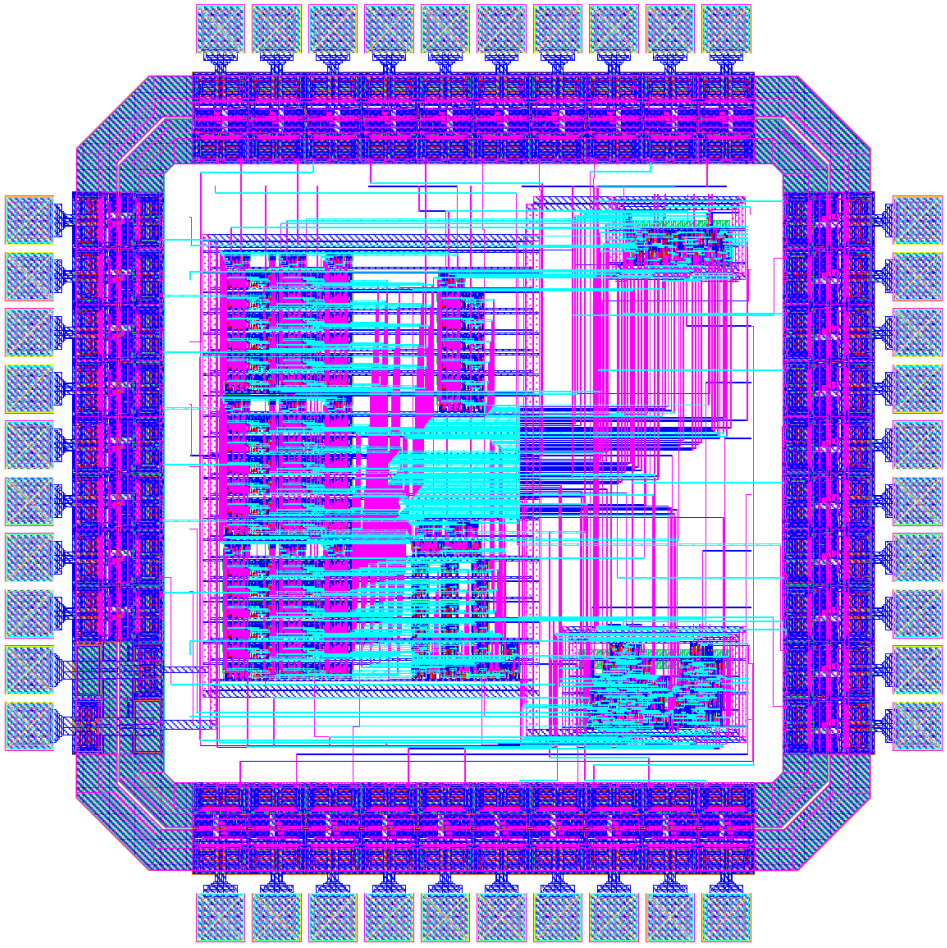
E158 Intro to CMOS VLSI Design

Alarm Clock



Sarah Yi & Samuel (Tae) Lee
4/19/2010

Introduction

The Alarm Clock chip includes the basic functions of an alarm clock such as a running clock time and alarm time that can be set, an a.m. or p.m. indicating output, an output for the alarm that is triggered when the two times match, and four seven bit outputs to connect to seven-segment LED displays for display. The user is able to set the clock time or the alarm time by incrementing either the hour or the minute. If the *apower* is on, the alarm will be triggered for the minute that the clock time and the alarm time match. If the *apower* is off, the alarm will never be triggered regardless of the time. The clock runs a 12-hour time and thus the a.m. or p.m. indicator is necessary. This time is expected to be displayed with four seven-segment LEDs and an extra LED to indicate a.m. or p.m.

This chip consists of three blocks: a datapath (custom), clockController (synthesized) and LEDdecoder (synthesized). The datapath consists of several cells that groups repetitive logic for regularity. It consists of a counter, three “alarm” cells, an “apmp” cell and a “buzzer” cell that together compares the clock time to the alarm time and output the time display and the alarm. The clockController encodes the control signals based on the user inputs and sends these signals to the datapath. It is in charge of enabling the counters to start counting resetting them when appropriate to best represent a functional clock. The LEDdecoder decodes the current display input from the datapath, whether it is the clock time or the alarm time, for the four seven-segment LED display outputs. The schematics and layout for these two cells were automatically generated from the Verilog code and auto-routed with SOC Ecnouter.

Specifications

This chip uses two phase overlapping clocks that run at the same speed in order to minimize race conditions. It also includes a reset feature to override all internal functions. The set signals, *cset* and *aset*, will allow the user to change the clock time and the alarm time respectively. In order to make these changes, either the *hr* or *min* signals must be high while simultaneously either of the set signals is held high. Both the “hr” and “min” inputs allow increments of once per clock cycle; however, no polling or debouncing modules has been implemented. The *apower* signal will control whether the alarm setting is on or off. For this chip, the alarm runs for a minute while the alarm time matches the clock time and will turn off automatically as soon as the clock time changes. Thus, *apower* will have to be set high in order for the alarm to ring at the correct time, and it will have to be set low in order to turn off the alarm within the minute. The output pins for *LED0*, *LED1*, *LED2* and *LED3*, each seven bits in width, should connect to four seven-segment LED displays for the minute one's digit, the minute ten's digit, the hour one's digit and the hour ten's digit respectively. The remaining pins, *ampm* and *buzz*, are both one bit outputs. The former indicates whether the time display is a.m. or p.m., and the latter indicates whether or not the alarm should be set off.

Type/ Direction	Description	Pin Name
Input	Pin used for the first clock	*ph1*
Input	Pin used for the second phase overlapping clock	*ph2*
Input	Pin used for reset	*reset*
Input/Output	Pin used for power connection	*vdd*
Input/Output	Pin used for ground connection	*gnd*
Input	Pin used for the set clock time control	*cset*
Input	Pin used for the set alarm time control	*aset*
Input	Pin used to turn on and off the alarm setting	*apower*
Input	Pin used to increment the hour of either the clock or the alarm time, depending on whether the *cset* or *aset* signal is high respectively	*hr*
Input	Pin used to increment the minute of either the clock or the alarm time, depending on whether the *cset* or *aset* signal is high respectively	*min*
Output	Pin used to indicate whether or not the clock time matches the alarm time	*buzz*
Output	Pin should be connected to the LED to display the time minute one's digit	*LED0[6:0]*
Output	Pin should be connected to the LED to display the time minute ten's digit	*LED1[6:0]*
Output	Pin should be connected to the LED to display the time hour one's digit	*LED2[6:0]*
Output	Pin should be connected to the LED to display the time hour ten's digit	*LED3[6:0]*
Output	Pin used to indicate either "a.m." or "p.m."	*ampm*

Figure 1: Table of all input and output pins for the Alarm Clock chip. (All inputs and outputs are one bit in bus width unless specified otherwise)

Floorplan

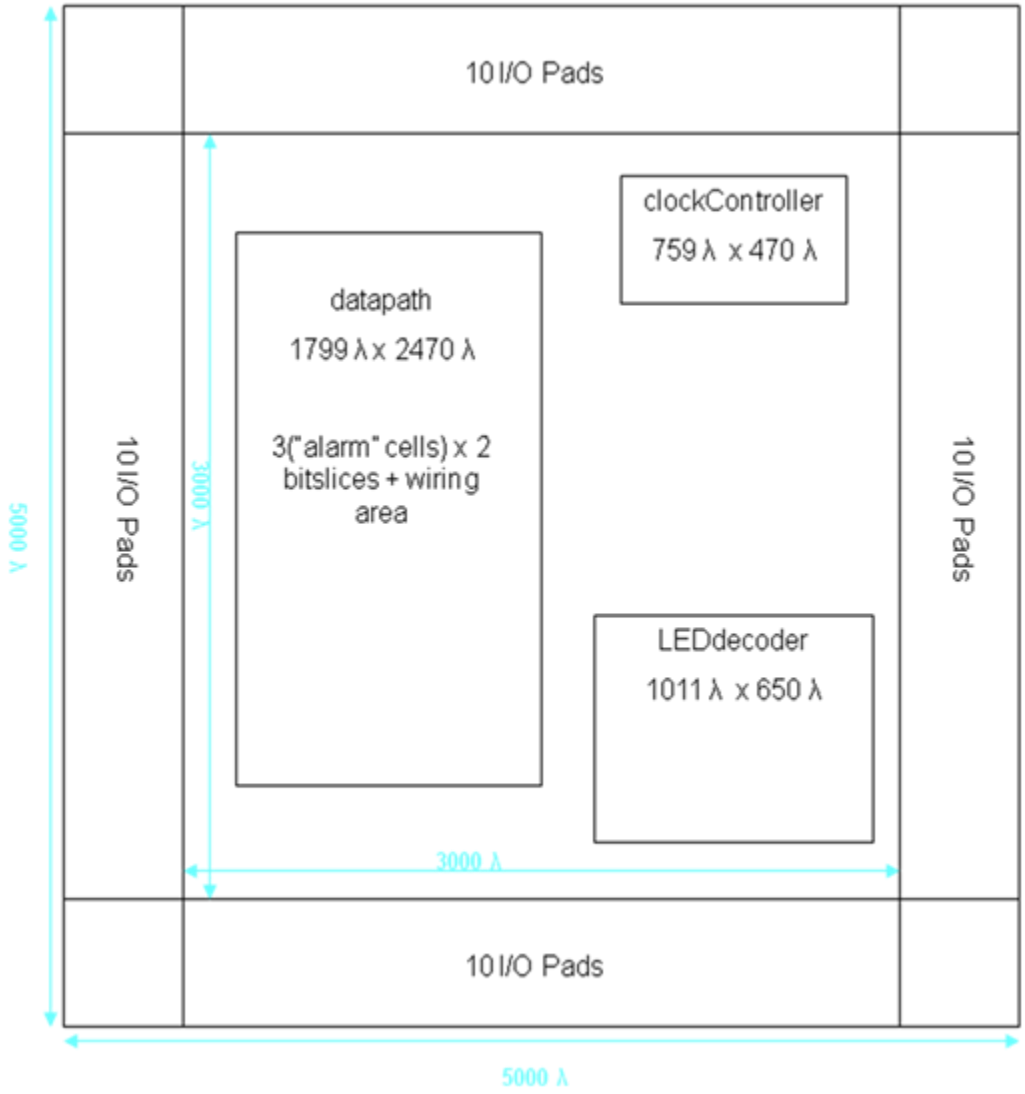


Figure 2: The Floorplan of the Alarm Clock Chip *(the extra area between the three cells corresponds to the wiring area for the layout).* This diagram is not drawn to scale.

The datapath in the proposal (Appendix D) consisted of a grouped logic for the clock time. It proposed a series of registers, adders, multiplexers and comparators to run the clock time while comparing it to the alarm time. However, the final datapath has separated the logic into similar cells for regularity. Thus the two counters, mux2 and flopenr have been grouped to form the "alarm" cell which handles both the clock and alarm time. Three of these cells were used for the minute ones, the minute tens, and the hour, while the seconds was accounted for by the counter in the datapath. The "apmp" cell handles the a.m. and p.m. feature for both the clock time and the alarm time. Lastly, the "buzzer" cell in the datapath consists of comparators and

other logic gates to compare the clock and alarm time in order to determine the “buzz” output. Also, the cells in the datapath are stacked for compactness and also for the ease of wiring. Moreover, the proposed floorplan consisted of two blocks, a datapath and a decoder. However, in the final design, the original datapath has been separated into two different cells, “clockController” to handle all the control signals based on the user inputs, and “datapath” to handle the clock time, alarm time, and comparison. The original floorplan also did not account for the extra area needed for wiring and thus is much smaller in size compared to the actual floorplan. Lastly, the proposal called for one one-bit output and three seven-bit outputs for the time display, but the actual design incorporates four seven-bit outputs for regularity.

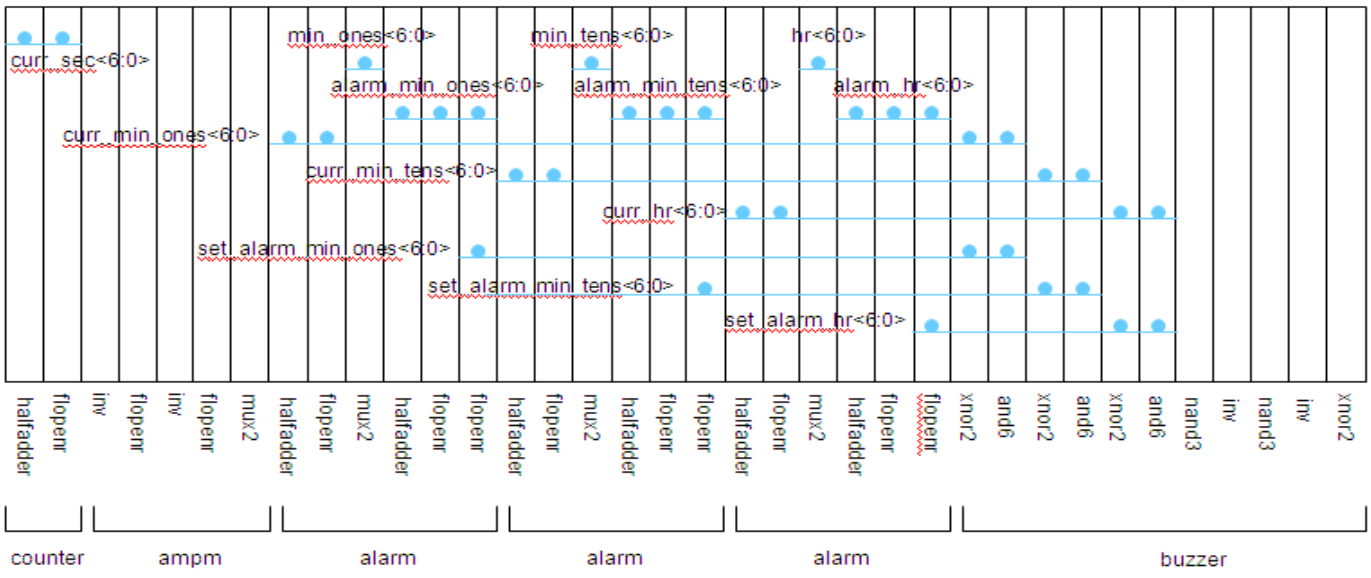


Figure 3: The slice plan for the datapath of the Alarm Clock chip. Only the seven-bit paths are shown for wiring locality.

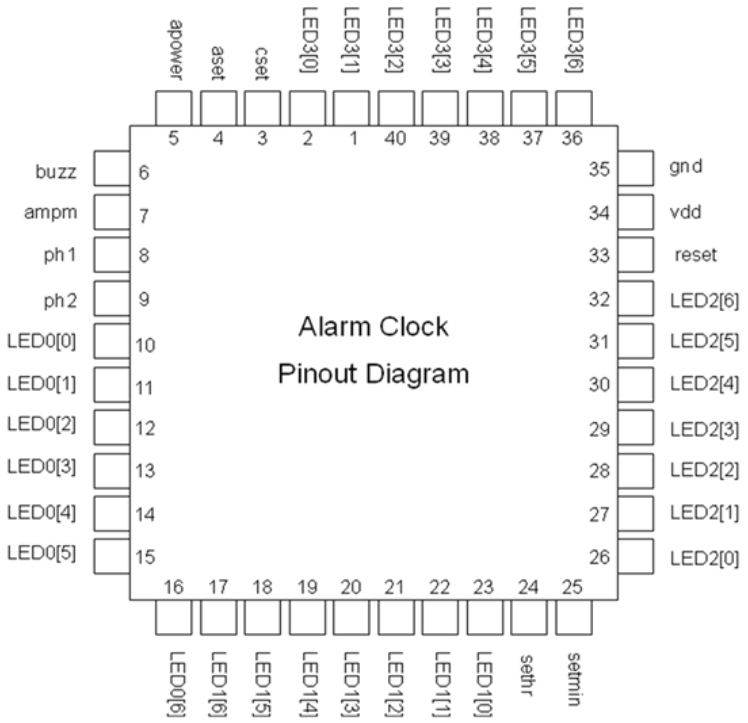


Figure 4: Pin-out Diagram of the Alarm Clock chip.

Verification

The Verilog code and the schematics (Appendix B) pass the testbench (Appendix A) without any errors. This testbench includes testvectors that simulate the basic functions of setting the clock time and the alarm time, running the clock through 24 hours, and setting off the alarm at the appropriate time. Both the layout and the CIF pass the DRC and LVS tests at the chip level. However, the clock speed remains an area of concern. Although the chip utilizes a two phase overlapping clocks to minimize the chances of clocking in an input twice or not clocking in the input, the clock speed contributes significantly to the possibility of these race conditions. Currently, the chip simulates correctly with two 1Hz clocks. However, this design can be better optimized for speed and usability by adjusting the clock speed simultaneously with the necessary clock skew. With high clock speed, however, bouncing may become an issue. Currently, a button press from the user would register only at1Hz, so although painfully slow, the presses would register without bouncing errors. However, with much faster clock speeds, bouncing may come into effect. Other areas of improvement include the optimization of the layout in terms of area. Since the datapath block was customized, a lot of space was set aside for the ease of wiring, but the design can allow for wiring over the cells to minimize this extra area. To enhance the design, additional features can be added such as a snooze button that temporarily silences the alarm that can ring again after a certain number of minutes pass and the ability to speed up the setting increments while the set button is held as seen in other general purpose alarm clocks.

Fabrication Test Plan

Post-fabrication testing is necessary to verify the absence of any design or manufacturing errors. In order to test this chip, four seven-segment LED displays should be connected to the appropriate output pins. The "ampm" and "buzz" output pins can be connected to some indicator such as an object that will either light up or make a sound when triggered. Since all the inputs are one bit in width, they can be connected to switches. The pin-out diagram can be seen in Figure 4. Then, the hardware can be manually tested by toggling the input switches and verifying that the resulting outputs are correct.

Design Time Summary

Design Level	**Estimated Time Spent**
Preliminary Design	10 hours
Verilog Code/Testbench	23 hours
Schematics	19 hours
Layouts	27 Hours

Figure 5: Summary of Design Times

File Locations

The Verilog code (alarmclock.v), testvectors (helpter.txt), synthesized results (_syn files), PDF chip plot (chip_layout.pdf & chip_schematic.pdf) and the PDF of the report (VLSI_Final_Project_Report.pdf) are contained in ~/IC_CAD/cadence/proj2files on chips. All Cadence libraries are found in ~/IC_CAD/cadence/proj2. CIF is found in ~/IC_CAD/cadence/alarmclock_cifin.

Appendix A: Verilog Code

alarmclock.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: CMOS VLSI Design
// Engineer: Samuel (Tae) Lee & Sarah Yi
// E-mail: tlee@hmc.edu or syi@hmc.edu
//
// Create Date:    23:12:44 03/21/2010
// Design Name: Alarm Clock
// Module Name:    alarmclock & submodules
//////////////////////////////////////////////////////////////////////////////////
module alarmclock(input ph1, ph2, reset,
                  input cset, aset, apower,
                  input sethr, setmin,
                  output buzz, ampm,
                  output [6:0] LED0, LED1, LED2, LED3);

        wire [5:0] min_ones;
        wire [5:0] min_tens;
        wire [5:0] hr;
        wire [5:0] curr_sec, curr_min_ones, curr_min_tens, curr_hr;
        wire [5:0] alarm_min_ones, alarm_min_tens, alarm_hr;
        wire sec_en, min_ones_en, min_tens_en, hr_en;
        wire sec_reset, min_ones_reset, min_tens_reset, hr_reset;
        wire alarm_min_ones_en, alarm_min_tens_en, alarm_hr_en;
        wire alarm_min_ones_reset, alarm_min_tens_reset, alarm_hr_reset;
        wire curr_ampm_en, alarm_ampm_en;

        // datatpath module call
        datapath DP( ph1, ph2, reset, aset, apower, curr_ampm_en, alarm_ampm_en, sec_en,
                sec_reset, min_ones_en, min_ones_reset, min_tens_en, min_tens_reset, hr_en,
                hr_reset, alarm_min_ones_en, alarm_min_ones_reset, alarm_min_tens_en,
                alarm_min_tens_reset, alarm_hr_en, alarm_hr_reset, buzz, ampm, curr_sec,
                curr_min_ones, curr_min_tens, curr_hr, alarm_min_ones, alarm_min_tens, alarm_hr,
                hr, min_ones, min_tens);

        // controller module call
        clockController controller( reset, cset, aset, setmin, sethr, curr_sec, curr_min_ones,
                curr_min_tens, curr_hr, alarm_min_ones, alarm_min_tens, alarm_hr, curr_ampm_en,
                alarm_ampm_en, sec_en, min_ones_en, min_tens_en, hr_en, sec_reset, min_ones_reset,
                min_tens_reset, hr_reset, alarm_min_ones_en, alarm_min_tens_en, alarm_hr_en,
                alarm_min_ones_reset, alarm_min_tens_reset, alarm_hr_reset);

        // LED decoder module call
        LEDdecoder leds( hr, min_ones, min_tens, LED2, LED3, LED0, LED1);

endmodule


module datapath(input  ph1, ph2, reset, aset, apower,
                input  curr_ampm_en, alarm_ampm_en,
                input  sec_en, sec_reset,
                input  min_ones_en, min_ones_reset,
                input  min_tens_en, min_tens_reset,
                input  hr_en, hr_reset,
                input  alarm_min_ones_en, alarm_min_ones_reset,
                input  alarm_min_tens_en, alarm_min_tens_reset,
                input  alarm_hr_en, alarm_hr_reset,
                output buzz,
               output ampm,
               output [5:0] curr_sec, curr_min_ones,curr_min_tens,curr_hr,
               output [5:0] alarm_min_ones, alarm_min_tens, alarm_hr,
               output [5:0] hr, min_ones,
               output [5:0] min_tens);


        wire alarm_ampm, curr_ampm;
```

```
        wire [5:0] set_alarm_min_ones;
        wire [5:0] set_alarm_min_tens;
        wire [5:0] set_alarm_hr;

        // switch current time ampm at 12
        flopenr #(1) curr_ampmflop(ph1, ph2, reset, curr_ampm_en, ~curr_ampm, curr_ampm);

        // counters for current time
        counter #(6) clock_sec(ph1, ph2, sec_en, sec_reset, curr_sec);
        counter #(6) clock_min_ones(ph1, ph2, min_ones_en, min_ones_reset, curr_min_ones);
        counter #(6) clock_min_tens(ph1, ph2, min_tens_en, min_tens_reset, curr_min_tens);
        counter #(6) clock_hour(ph1, ph2, hr_en, hr_reset, curr_hr);

        // switch alarm ampm at 12
        flopenr #(1) alarm_ampmflop(ph1, ph2, reset, alarm_ampm_en, ~alarm_ampm, alarm_ampm);

        // counters for alarm time, enabled by buton presses
        counter #(6) alarm_min_ones_set(ph1, ph2, alarm_min_ones_en, alarm_min_ones_reset,
                                        alarm_min_ones);
        counter #(6) alarm_min_tens_set(ph1, ph2, alarm_min_tens_en, alarm_min_tens_reset,
                                       alarm_min_tens);
        counter #(6) alarm_hour_set(ph1, ph2, alarm_hr_en, alarm_hr_reset, alarm_hr);

        // stored alarm time
        flopenr #(6) alarm_min_ones_flop(ph1,ph2,reset,asset,alarm_min_ones,set_alarm_min_ones);
        flopenr #(6) alarm_min_tens_flop(ph1,ph2,reset,aset, alarm_min_tens,set_alarm_min_tens);
        flopenr #(6) alarm_hr_flop(ph1, ph2, reset, aset, alarm_hr, set_alarm_hr);

        // set buzzer
        buzz_control buzz_set(curr_min_ones, curr_min_tens, curr_hr, set_alarm_min_ones,
                              set_alarm_min_tens, set_alarm_hr, curr_ampm, alarm_ampm, apower,
                              buzz);

        // choose which time (current or alarm) to display
        mux2 #(6) display_min_ones(curr_min_ones, alarm_min_ones, aset, min_ones);
        mux2 #(6) display_min_tens(curr_min_tens, alarm_min_tens, aset, min_tens);
        mux2 #(6) display_hr(curr_hr, alarm_hr, aset, hr);
        mux2 #(1) display_ampm(curr_ampm, alarm_ampm, aset, ampm);

endmodule


module buzz_control(input  [5:0] curr_min_ones, curr_min_tens, curr_hr,
                    input  [5:0] set_alarm_min_ones,set_alarm_min_tens, set_alarm_hr,
                    input        curr_ampm, alarm_ampm, apower,
                    output       buzz);

        wire min_ones_equal, min_tens_equal, hr_equal, time_equal, ampm_equal;

        // check if the time is equal to the set alarm time and set off buzz
        comparator #(6) min_ones_comparator(curr_min_ones, set_alarm_min_ones, min_ones_equal);
        comparator #(6) min_tens_comparator(curr_min_tens, set_alarm_min_tens, min_tens_equal);
        comparator #(6) hr_comparator(curr_hr, set_alarm_hr, hr_equal);
        and3 #(1) buzz_and3(min_ones_equal, min_tens_equal, hr_equal, time_equal);
        xnor2 #(1) buzz_xnor2(curr_ampm, alarm_ampm, ampm_equal);
        and3 #(1) buzz_and2(time_equal, ampm_equal, apower, buzz);

endmodule


module clockController(input reset, cset, aset, setmin, sethr,
                       input [5:0] curr_sec,
                       input [5:0] curr_min_ones,
                       input [5:0] curr_min_tens,
                       input [5:0] curr_hr,
                       input [5:0] alarm_min_ones,
                       input [5:0] alarm_min_tens,
                       input [5:0] alarm_hr,
                       output curr_ampm_en, alarm_ampm_en,
                       output sec_en, min_ones_en, min_tens_en, hr_en,
```

```
                output sec_reset, min_ones_reset, min_tens_reset, hr_reset,
                output alarm_min_ones_en, alarm_min_tens_en, alarm_hr_en,
                output alarm_min_ones_reset, alarm_min_tens_reset, alarm_hr_reset);

    parameter max_min_ones = 6'b00_1001;  //9
    parameter max_min_tens = 6'b00_0101;  //5
    parameter max_sec = 6'b11_1011;              //59
    parameter max_hr = 6'b00_1011;               //11

    wire sec_end, min_ones_end, min_tens_end, hr_end;
    wire alarm_min_ones_end, alarm_min_tens_end, alarm_hr_end;
    wire run_min_ones_en, set_min_ones_en;
    wire run_hr_en, set_hr_en;
    wire run_curr_ampm_en, set_curr_ampm_en;
    wire sec_reset_mid;

    // check if each digit is at its max
    comparator #(6) sec_compare(curr_sec, max_sec, sec_end);
    comparator #(6) min_ones_compare(curr_min_ones, max_min_ones, min_ones_end);
    comparator #(6) min_tens_compare(curr_min_tens, max_min_tens, min_tens_end);
    comparator #(6) hr_compare(curr_hr, max_hr, hr_end);

    comparator #(6) alarm_min_ones_reset_compare(alarm_min_ones, max_min_ones,
                                                 alarm_min_ones_end);
    comparator #(6) alarm_min_tens_reset_compare(alarm_min_tens, max_min_tens,
                                                 alarm_min_tens_end);
    comparator #(6) alarm_hour(alarm_hr, max_hr, alarm_hr_end);


    // enable seconds when clock is not being set
    inv #(1) sec_en_inv(cset, sec_en);

    //enable for minute ones digit to increment
    and2 #(1) run_min_ones_en_and2(sec_en, sec_end, run_min_ones_en);
    and2 #(1) set_min_ones_en_and2(cset, setmin, set_min_ones_en);
    or2 #(1) min_ones_en_or2(run_min_ones_en, set_min_ones_en, min_ones_en);

    //enable for minute tens digit to increment
    and2 #(1) min_tens_en_and2(min_ones_end, min_ones_en, min_tens_en);

    //enable for hour to increment
    and3 #(1) run_hr_en_and3(sec_en, min_tens_end, min_tens_en, run_hr_en);
    and2 #(1) set_hr_en_and2(cset, sethr, set_hr_en);
    or2 #(1) hr_en_or2(run_hr_en, set_hr_en, hr_en);

    //enable for current ampmp to switch
    and3 #(1) run_curr_ampm_en_and3(sec_en, hr_end, hr_en, run_curr_ampm_en);
    and3 #(1) set_curr_ampm_en_and3(cset, sethr, hr_end, set_curr_ampm_en);
    or2 #(1) curr_ampm_en_or2(run_curr_ampm_en, set_curr_ampm_en, curr_ampm_en);

    //enables for when setting alarm time
    and2 #(1) alarm_min_ones_en_and2(aset, setmin, alarm_min_ones_en);
    and3 #(1) alarm_min_tens_en_and3(aset, alarm_min_ones_end, alarm_min_ones_en,
                                     alarm_min_tens_en);
    and2 #(1) alarm_hr_en_and2(aset, sethr, alarm_hr_en);
    and3 #(1) alarm_ampm_en_and3(aset, sethr, alarm_hr_end, alarm_ampm_en);

    //reset logic
    and2 #(1) sec_reset_and2(sec_end, sec_en, sec_reset_mid);
    or3 #(1) sec_reset_or3(sec_reset_mid, cset, reset, sec_reset);
    a2o #(1) min_ones_reset_a2o(min_ones_end, min_ones_en, reset, min_ones_reset);
    a2o #(1) min_tens_reset_a2o(min_tens_end, min_tens_en, reset, min_tens_reset);
    a2o #(1) hr_reset_a2o(hr_end, hr_en, reset, hr_reset);
    a2o #(1) alarm_min_ones_reset_a2o(alarm_min_ones_end, alarm_min_ones_en, reset,
                                      alarm_min_ones_reset);
    a2o #(1) alarm_min_tens_reset_a2o(alarm_min_tens_end, alarm_min_tens_en, reset,
                                      alarm_min_tens_reset);
    a2o #(1) alarm_hr_reset_a2o(alarm_hr_end, alarm_hr_en, reset, alarm_hr_reset);

endmodule
```

```
module LEDdecoder(input  [5:0] hr,
                  input  [5:0] min_ones,
                  input  [5:0] min_tens,
                  output [6:0] hr_ones_segs, hr_tens_segs, min_ones_segs, min_tens_segs);

        // call sevenseg or sevenseg_hr to decode each number for SSD
        sevenseg min_ones_segments(min_ones, min_ones_segs);
        sevenseg min_tens_segments(min_tens, min_tens_segs);
        sevenseg_hr hr_segments(hr, hr_ones_segs, hr_tens_segs);

endmodule


module sevenseg(input    [5:0] data,
                output reg [6:0] segments);

        always @(*)
                case (data)
                                        // abc_defg
                        0: segments = 7'b111_1110;
                        1: segments = 7'b011_0000;
                        2: segments = 7'b110_1101;
                        3: segments = 7'b111_1001;
                        4: segments = 7'b011_0011;
                        5: segments = 7'b101_1011;
                        6: segments = 7'b101_1111;
                        7: segments = 7'b111_0000;
                        8: segments = 7'b111_1111;
                        9: segments = 7'b111_1011;
                        default: segments = 7'b000_0000; // required
                endcase

endmodule


module sevenseg_hr(input      [5:0] data,
                   output reg [6:0] segments_ones, segments_tens);

        // seven segment display for both digits of the hour
        always @(*)
                case (data)
                        0:
                                begin
                                segments_ones = 7'b110_1101;
                                segments_tens = 7'b011_0000;
                                end
                        1:
                                begin
                                segments_ones = 7'b011_0000;
                                segments_tens = 7'b000_0000;
                                end
                        2:
                                begin
                                segments_ones = 7'b110_1101;
                                segments_tens = 7'b000_0000;
                                end
                        3:
                                begin
                                segments_ones = 7'b111_1001;
                                segments_tens = 7'b000_0000;
                                end
                        4:
                                begin
                                segments_ones = 7'b011_0011;
                                segments_tens = 7'b000_0000;
                                end
                        5:
                                begin
                                segments_ones = 7'b101_1011;
                                segments_tens = 7'b000_0000;
                                end
```

```
                        6:      begin
                                segments_ones = 7'b101_1111;
                                segments_tens = 7'b000_0000;
                                end
                        7:      begin
                                segments_ones = 7'b111_0000;
                                segments_tens = 7'b000_0000;
                                end
                        8:
                                begin
                                segments_ones = 7'b111_1111;
                                segments_tens = 7'b000_0000;
                                end
                        9:
                                begin
                                segments_ones = 7'b111_1011;
                                segments_tens = 7'b000_0000;
                                end
                        10:
                                begin
                                segments_ones = 7'b111_1110;
                                segments_tens = 7'b011_0000;
                                end
                        11:
                                begin
                                segments_ones = 7'b011_0000;
                                segments_tens = 7'b011_0000;
                                end
                        default:
                                begin
                                segments_ones = 7'b000_0000;
                                segments_tens = 7'b000_0000;
                                end
                endcase

endmodule


module counter # (parameter WIDTH = 8)
                 (input              ph1, ph2, enable, reset,
                  output [WIDTH-1:0] curr_num);

        wire [WIDTH-1:0] next_num;

        incrementer #(WIDTH) increment(curr_num, next_num);
        flopenr #(WIDTH) counter_flop(ph1, ph2, reset, enable, next_num, curr_num);

endmodule


module incrementer # (parameter WIDTH = 8)
                     (input  [WIDTH-1:0] a,
                      output [WIDTH-1:0] y);

        assign y = a + 1;

endmodule


module comparator #(parameter WIDTH=8)
                   (input  [WIDTH-1:0] d0, d1,
                    output             equal);

        assign equal = (d0==d1);

endmodule


module inv # (parameter WIDTH = 8)
              (input [WIDTH-1:0] a,
               output [WIDTH-1:0] y);
```

```
        assign y = ~a;

endmodule


module or2 # (parameter WIDTH = 8)
             (input  [WIDTH-1:0] a, b,
              output [WIDTH-1:0] y);

        assign y = a|b;

endmodule


module or3 # (parameter WIDTH = 8)
             (input  [WIDTH-1:0] a, b, c,
              output [WIDTH-1:0] y);

        assign y = a|b|c;

endmodule


module and2 # (parameter WIDTH = 8)
              (input  [WIDTH-1:0] a, b,
               output [WIDTH-1:0] y);

        assign y = a&b;

endmodule


module and3 # (parameter WIDTH = 8 )
              (input  [WIDTH-1:0] a, b, c,
               output [WIDTH-1:0] y);

        assign y = a&b&c;

endmodule


module xnor2 # (parameter WIDTH = 8)
               (input  [WIDTH-1:0] a, b,
                output [WIDTH-1:0] y);

        assign y = a~^b;

endmodule


module a2o # (parameter WIDTH = 8)
             (input  [WIDTH-1:0] a, b, c,
              output [WIDTH-1:0] y);

        assign y = (a&b)|c;

endmodule


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
```

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

  wire [WIDTH-1:0] d2;

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

endmodule


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


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

  always @ ( * )
    if (ph) q <= d;

endmodule


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

  assign y = s ? d1 : d0;

endmodule


module mux3 #(parameter WIDTH = 8)
             (input     [WIDTH-1:0] d0, d1, d2,
              input     [1:0]       s,
              output reg [WIDTH-1:0] y);

       always @ ( * )
              case (s)
                     2'b00:  y <= d0;
                     2'b01:  y <= d1;
                     2'b10:  y <= d2;
                     2'b11:  y <= d2;
                     default:       y <= d0;
              endcase
endmodule
```

alarmclock_tb.sv

```
// testbench for testing
//`include "alarmclock.v"
`timescale 1ns / 1ps
module testbench();

  reg               ph1, ph2, reset;
  reg               cset, aset, apower;
  reg               sethr, setmin;
  wire              buzz, ampm;
  reg               buzzexp, ampmexp;
  wire [6:0]        LED0, LED1, LED2, LED3;
  reg [6:0]         LED0exp, LED1exp, LED2exp, LED3exp;
  integer           vectornum, errors;
  reg [34:0]        testvectors[0:150000];

  // instantiate devices to be tested
  // .* notation instantiates all ports in the mips module
  // with the correspondingly named signals in this module
  // alarmclock #(WIDTH,REGBITS) dut(.*);
  /* alarmclock run(ph1, ph2, reset, cset, aset, apower, sethr, setmin, buzz, ampm, LED0, LED1,
LED2, LED3);
       core run( LED0, LED1, LED2, LED3, ampm, buzz, apower, aset, cset,
     ph1, ph2, reset, sethr, setmin ); */

     chip run( LED0, LED1, LED2, LED3, ampm, buzz, apower, aset, cset, ph1, ph2, reset, sethr,
setmin );


  // initialize test
  initial
    begin
      reset <= 1; #20; reset <= 0;
    end

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


 // load vectors at start
 initial begin
   $dumpfile("alarmclock.vcd");                // where to dump the results
   $dumpvars(1, ph1, ph2, reset, cset, aset, apower, sethr, setmin, buzz, ampm, LED0, LED1, LED2,
LED3);
   $readmemb("helper.txt", testvectors);       // load test vectors
   vectornum = 0; errors = 0;
   reset = 1; #20 reset = 0;                  // hold reset before starting
 end


 // apply test vectors
  always @(posedge ph1)
    begin
      #1; {apower, aset, cset, sethr, setmin, LED3exp, LED2exp, LED1exp, LED0exp, ampmexp,
buzzexp} = testvectors[vectornum];
      //$display("apower = %b, aset = %b, cset = %b, sethr = %b, setmin = %b", apower, aset, cset,
sethr, setmin);
      //$display("loaded vector %0d as: %b", vectornum, testvectors[vectornum]);
    end


  // check results
  always @(negedge ph1)
    if (!reset) begin                          // skip during reset
```

```
      if (LED3 !== LED3exp | LED2 !== LED2exp | LED1 !== LED1exp | LED0 !== LED0exp | ampm !==
ampmexp | buzz !== buzzexp) begin
      $display("Error: LED3 = %b (expected %b), LED2 = %b (expected %b), LED1 = %b (expected %b),
LED0 = %b (expected %b), ampm = %b (expected %b), buzz = %b (expected %b)",
         LED3, LED3exp, LED2, LED2exp, LED1, LED1exp, LED0, LED0exp, ampm, ampmexp, buzz,
buzzexp);
      errors = errors + 1;
    end
    vectornum = vectornum + 1;
    if (testvectors[vectornum] === 35'bx) begin
      $display("%d tests completed with %d errors",
         vectornum, errors);
      $dumpflush;
//      #1000; $stop;
      $finish;
    end
  end
endmodule
```

Appendix B: Schematics

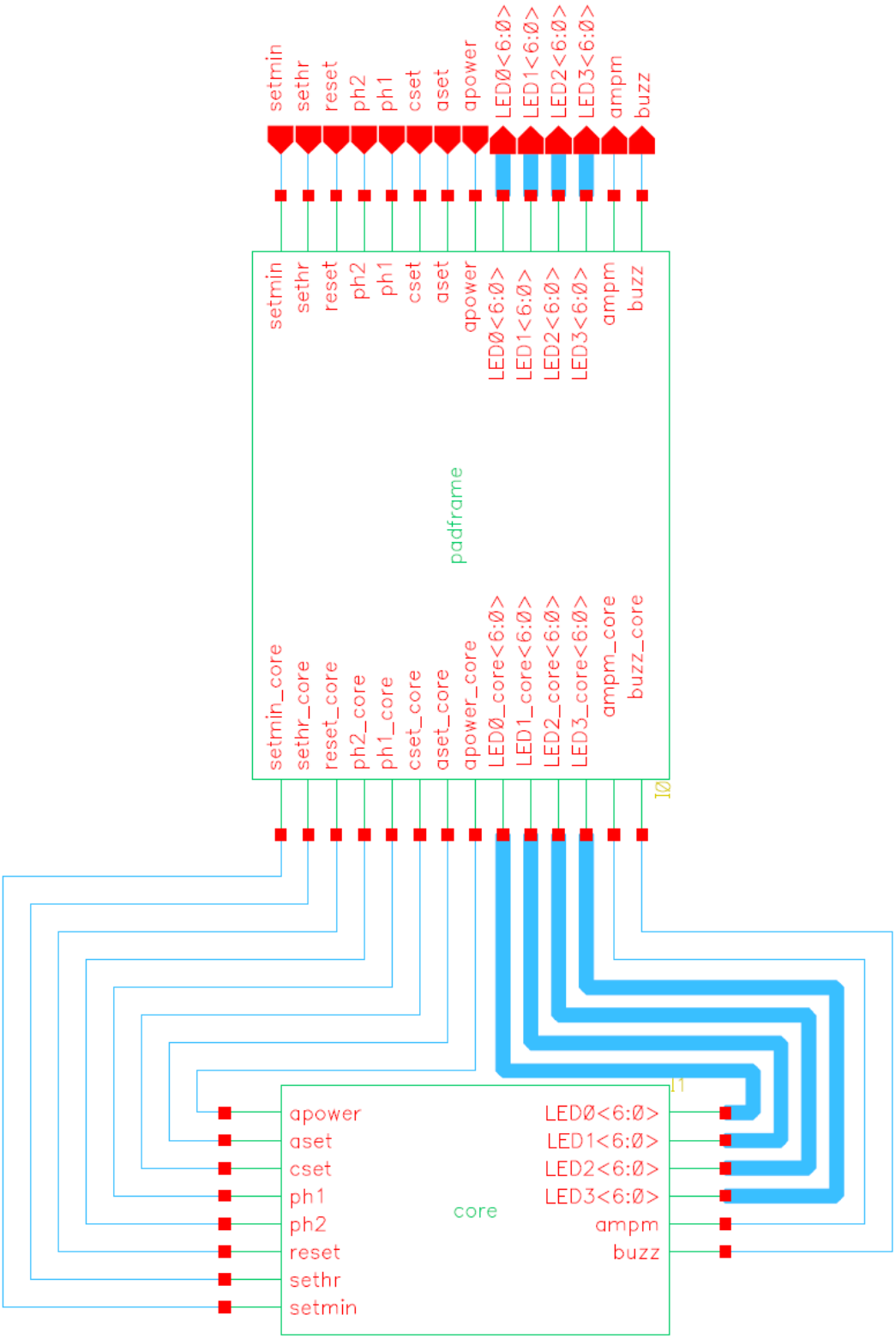


Figure 6: Chip Schematic

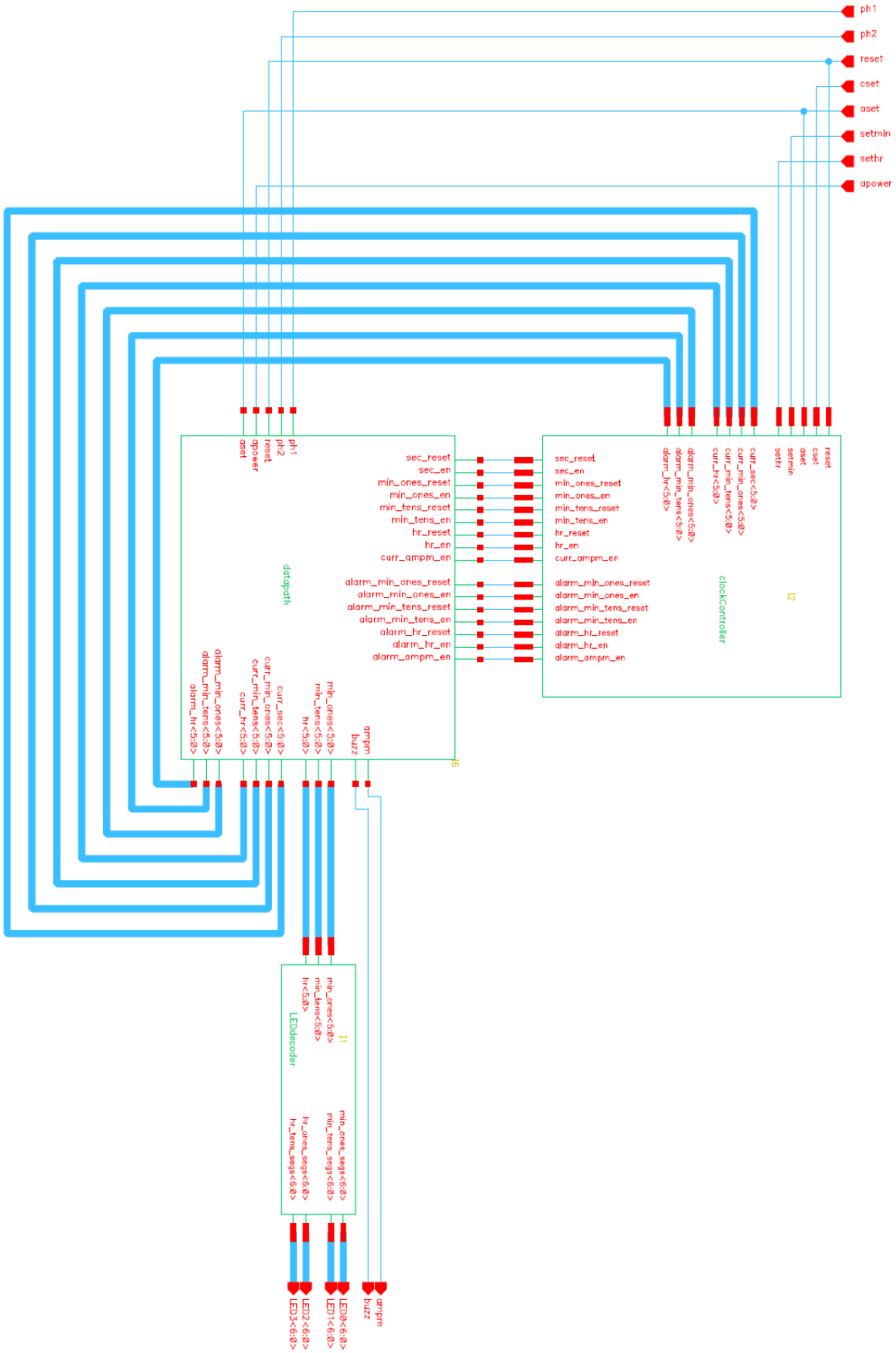


Figure 7: Core Schematic (Rotated 90 Degrees)

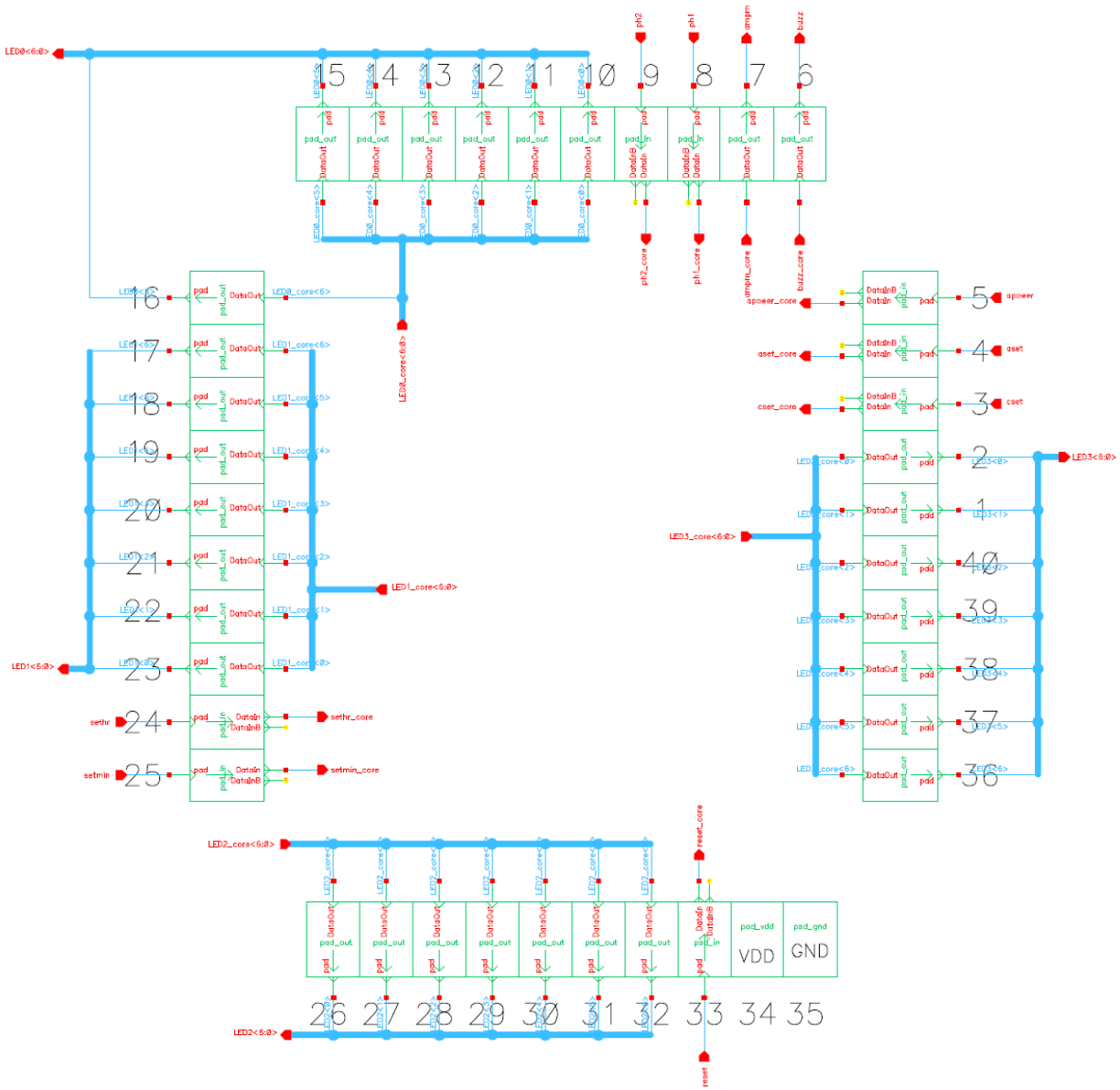


Figure 8: Padframe Schematic

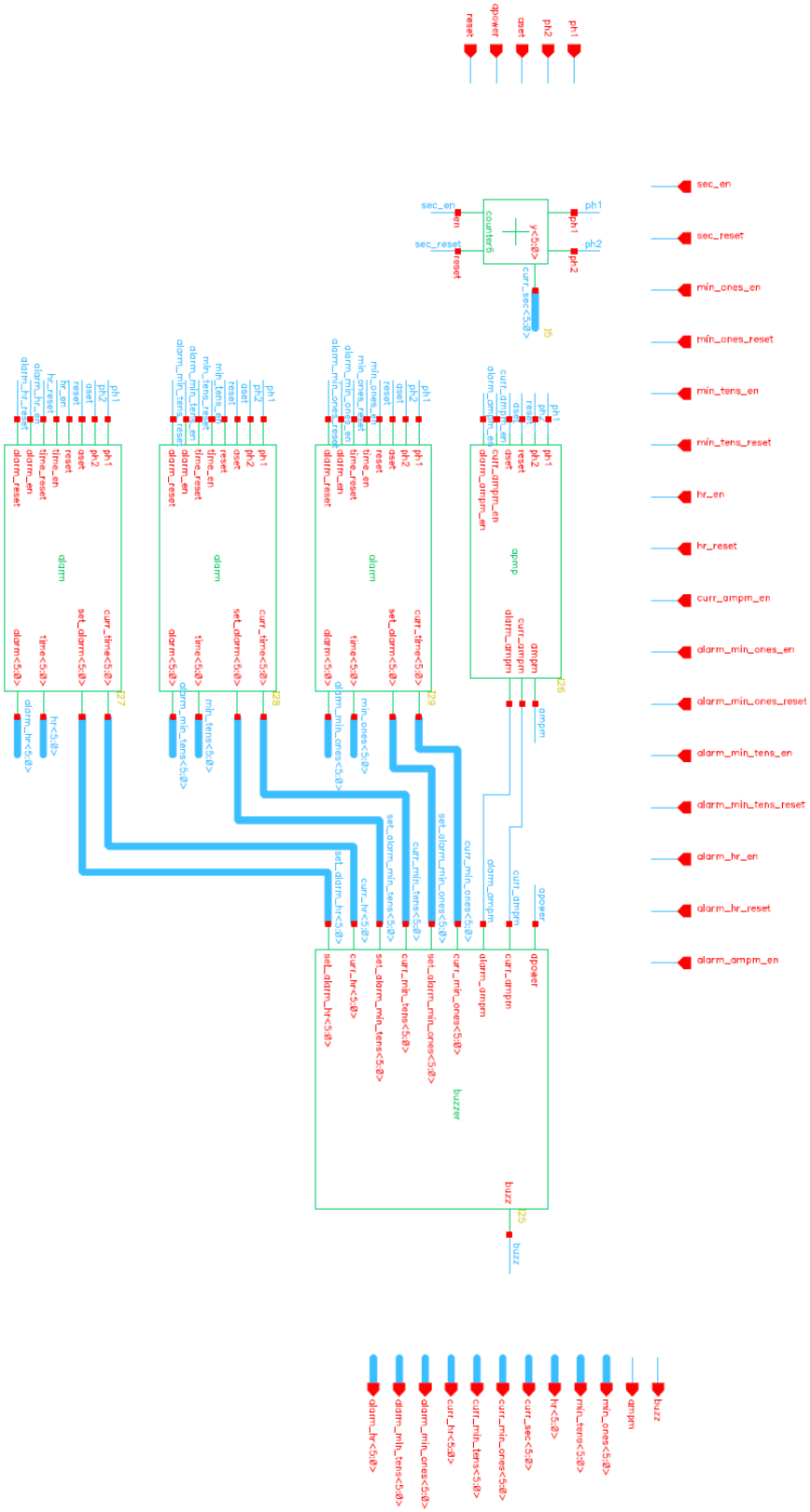


Figure 9: Datapath Schematic

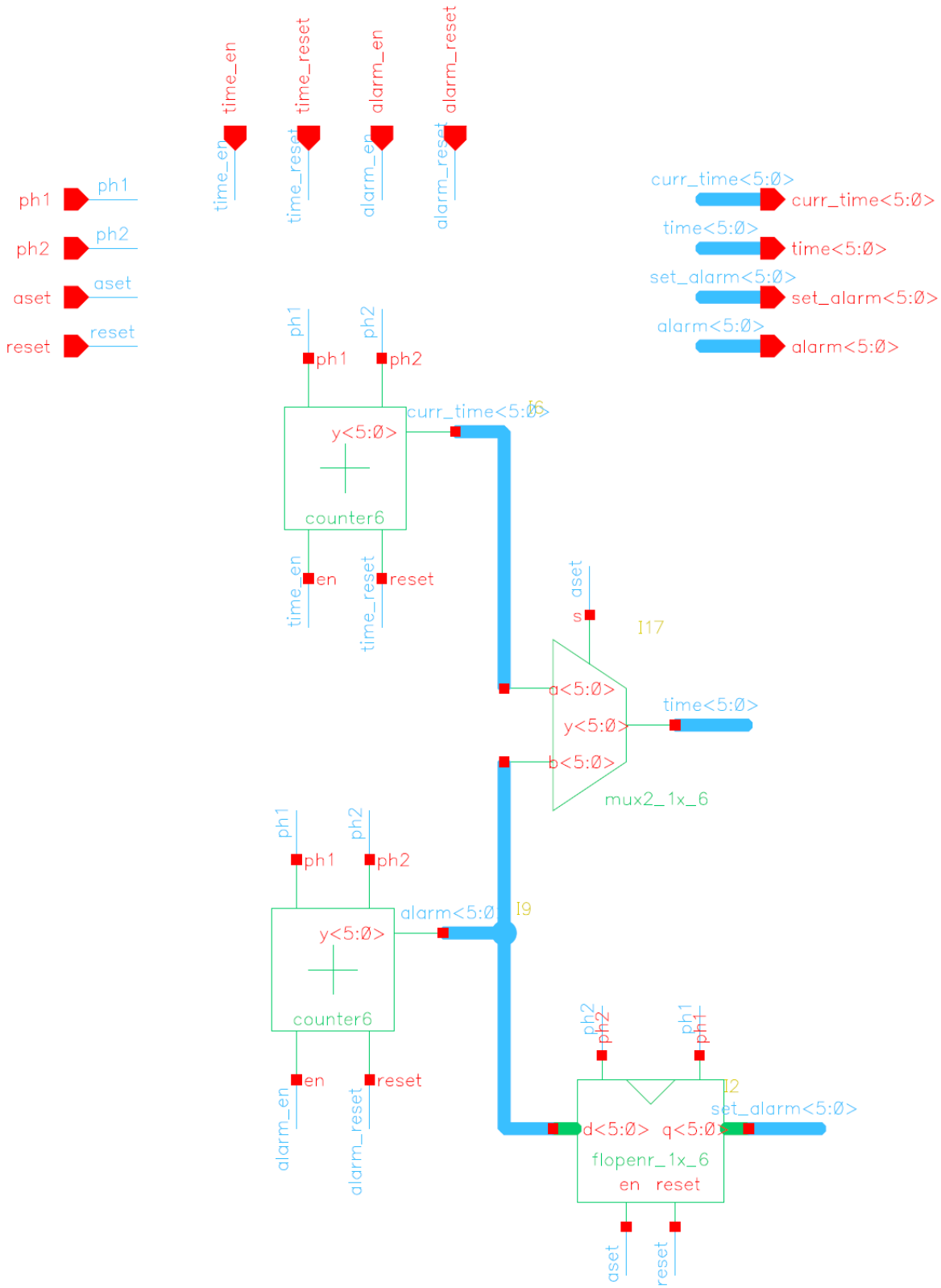


Figure 40: Alarm Schematic

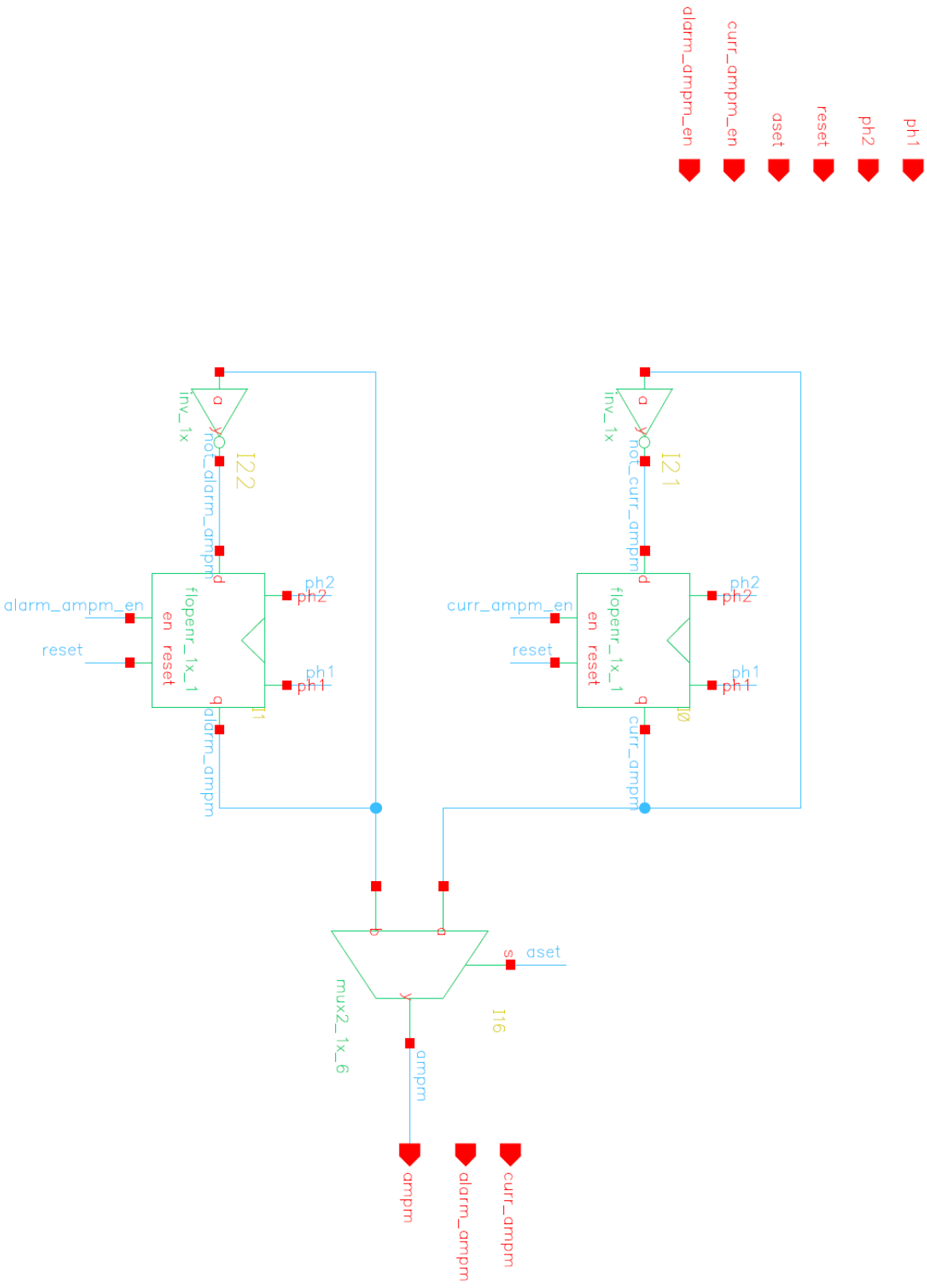


Figure 51: ampm Schematic (Rotated 90 Degrees)

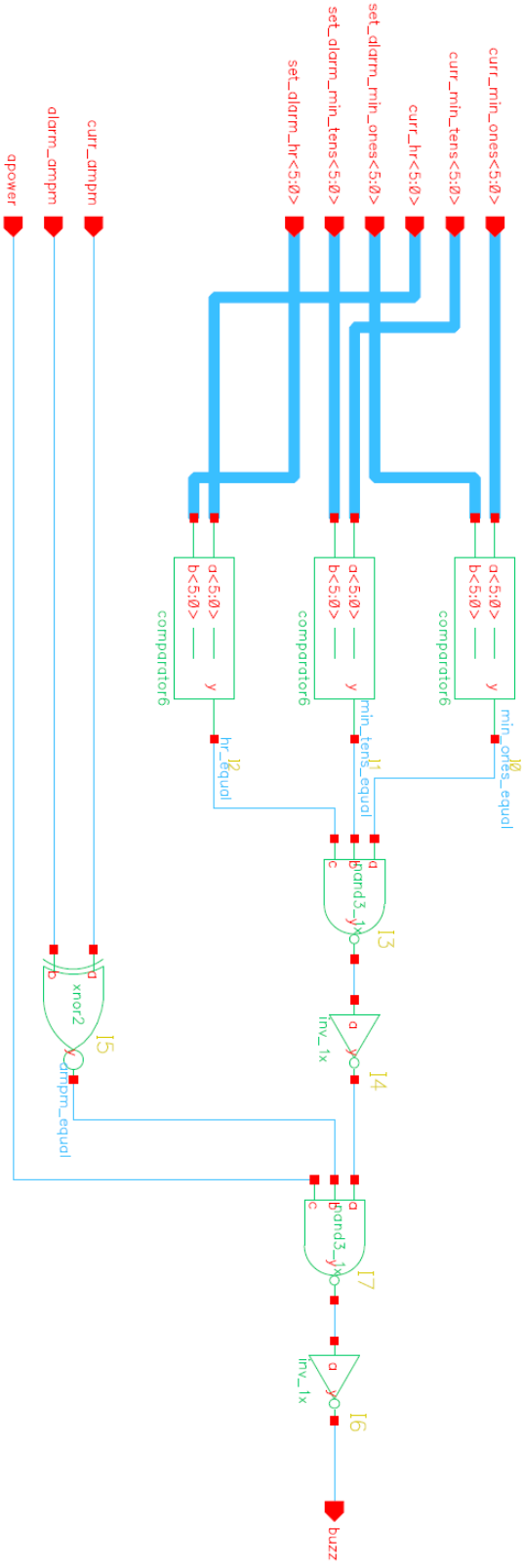


Figure 12: Buzzer Schematic (Rotated 90 Degrees)

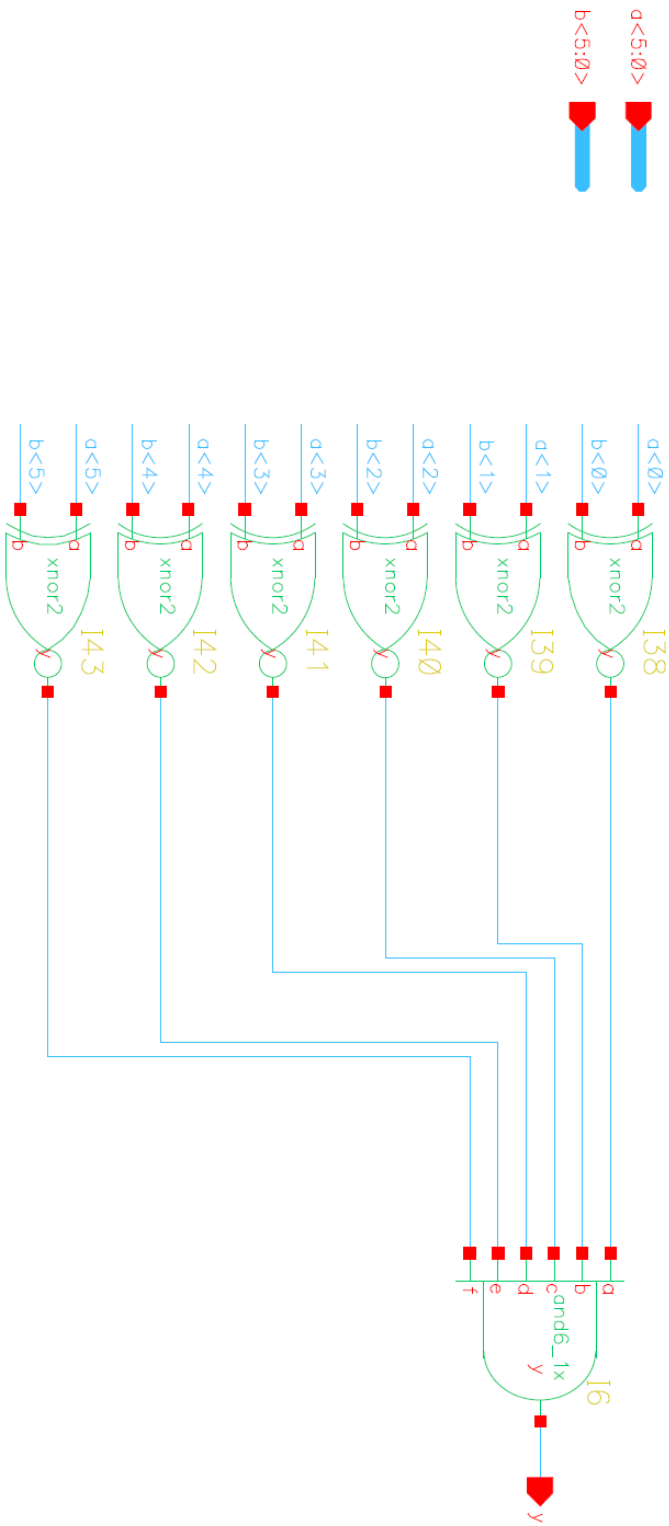


Figure 13: Comparator6 Schematic (Rotated 90 Degrees)

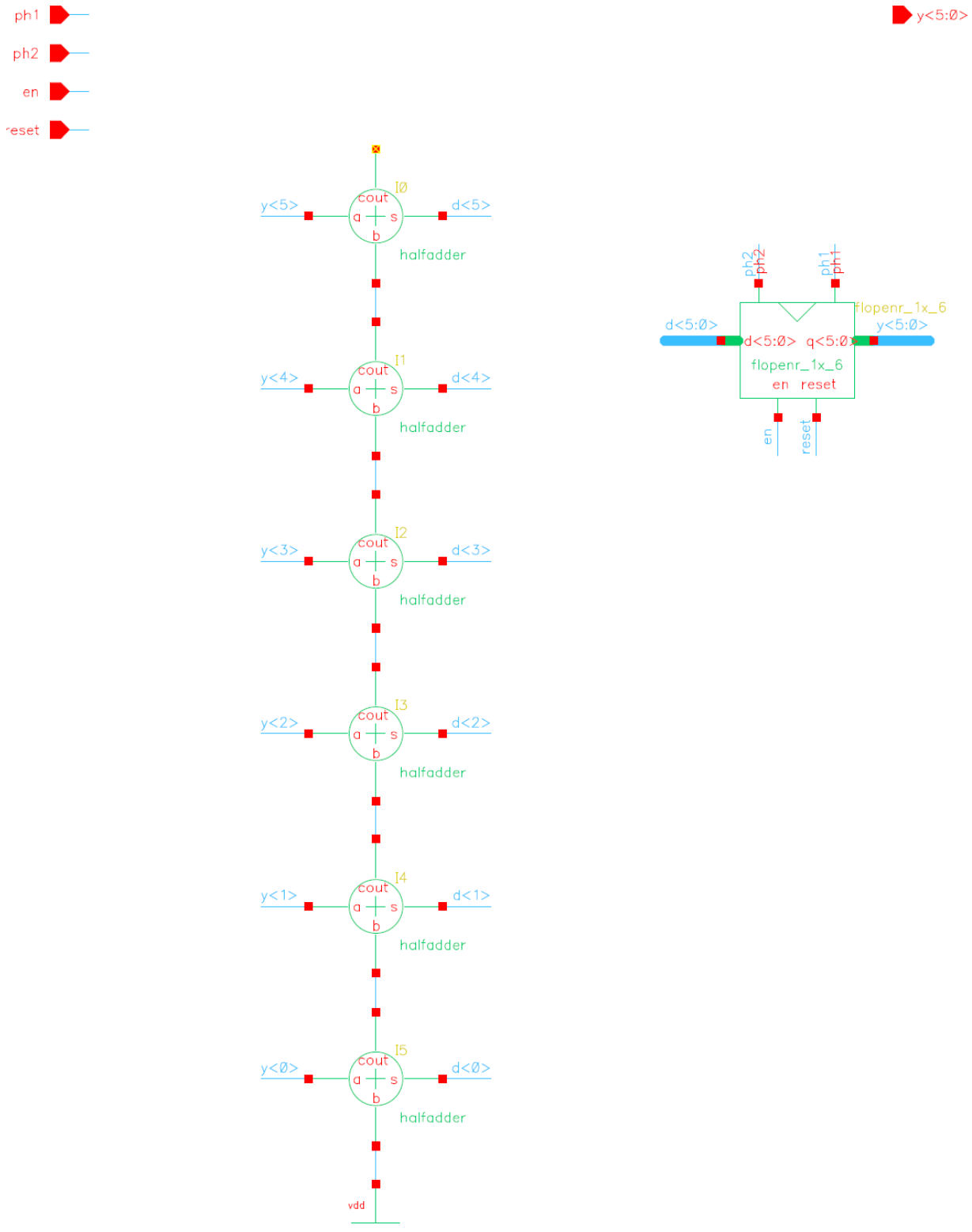


Figure 14: Counter6 Schematic

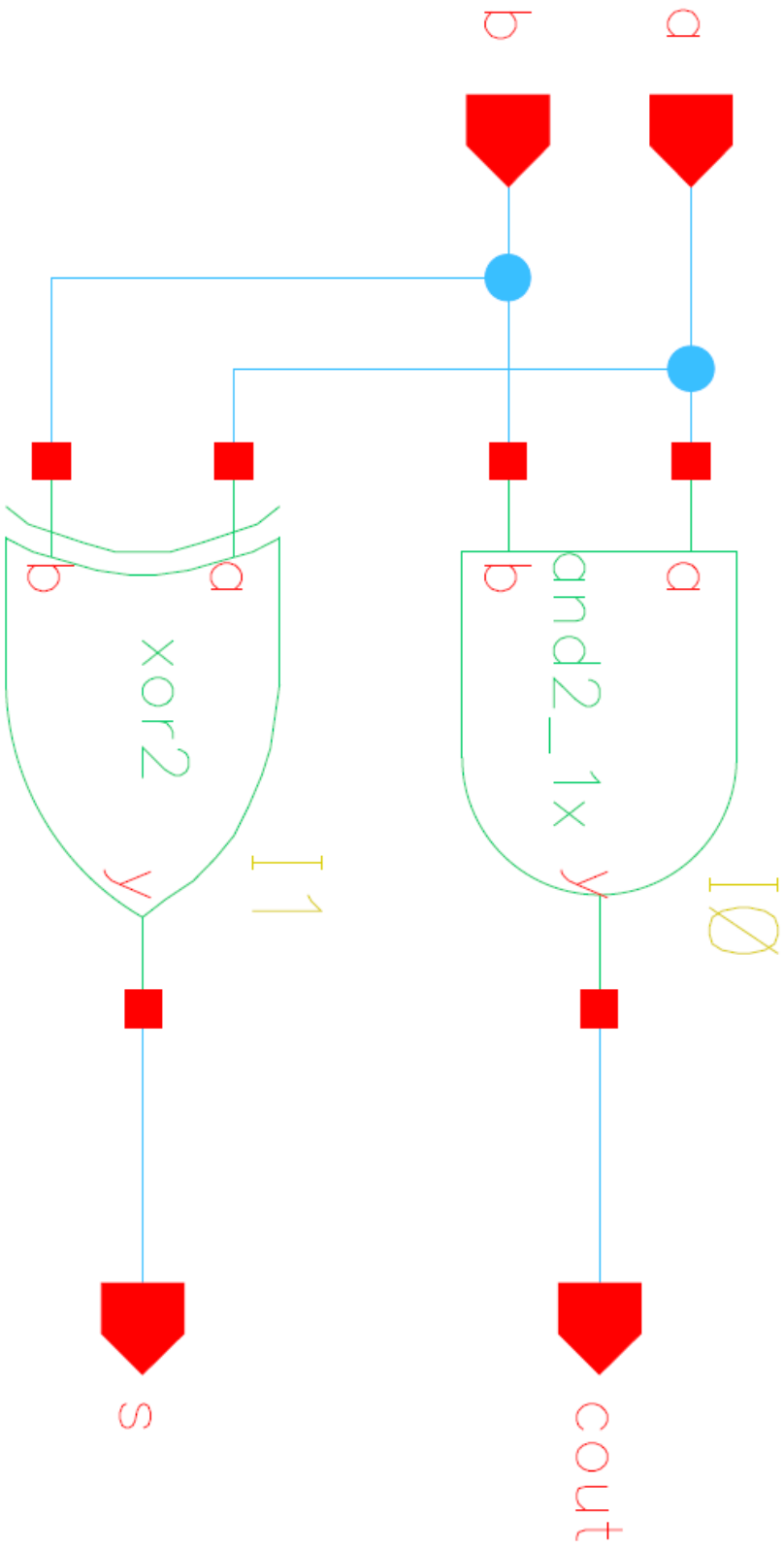


Figure 15: Half-Adder Schematic (Rotated 90 Degrees)

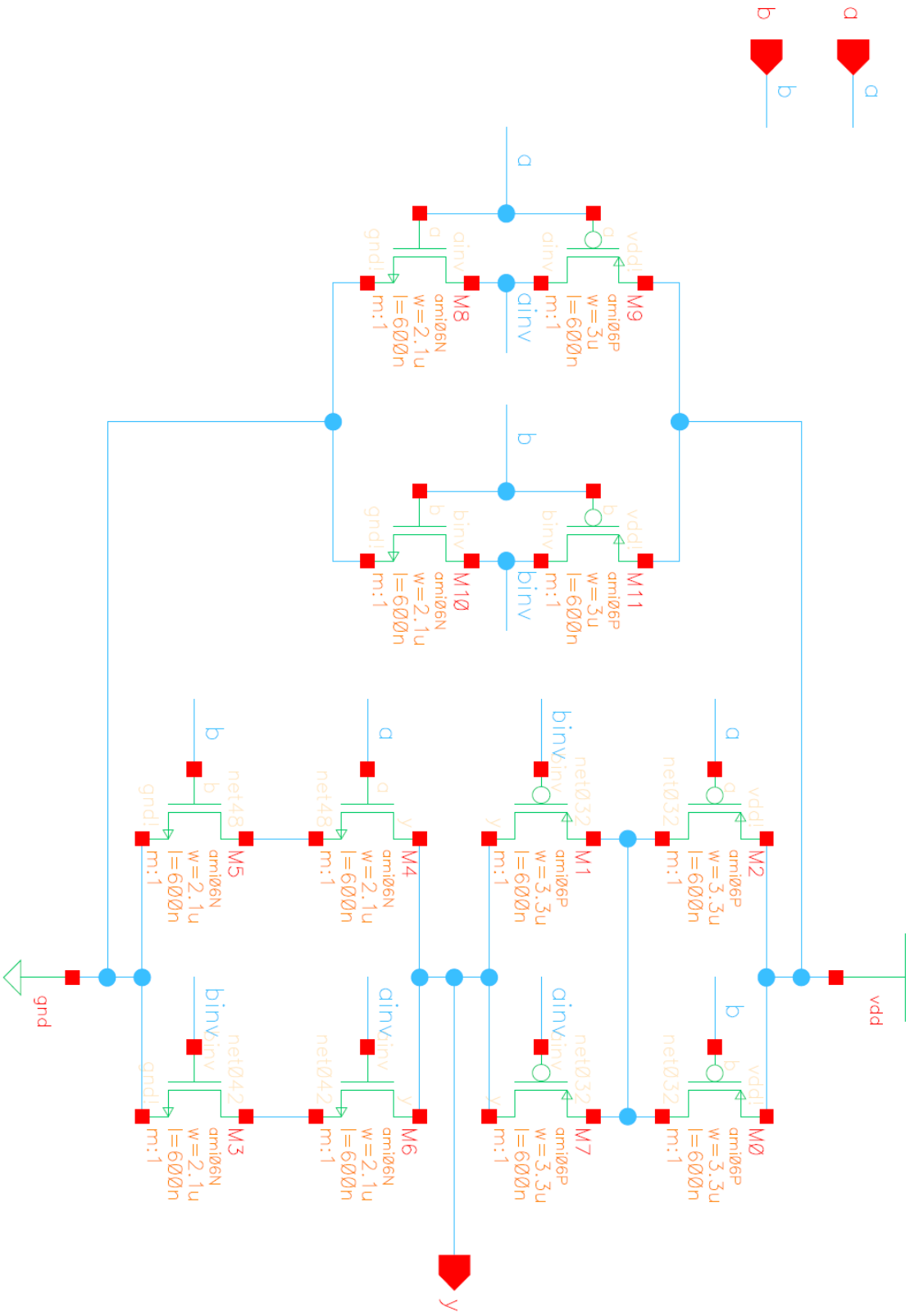


Figure 16: XOR CMOS Schematic (Rotated 90 Degrees)

Appendix C: Layouts

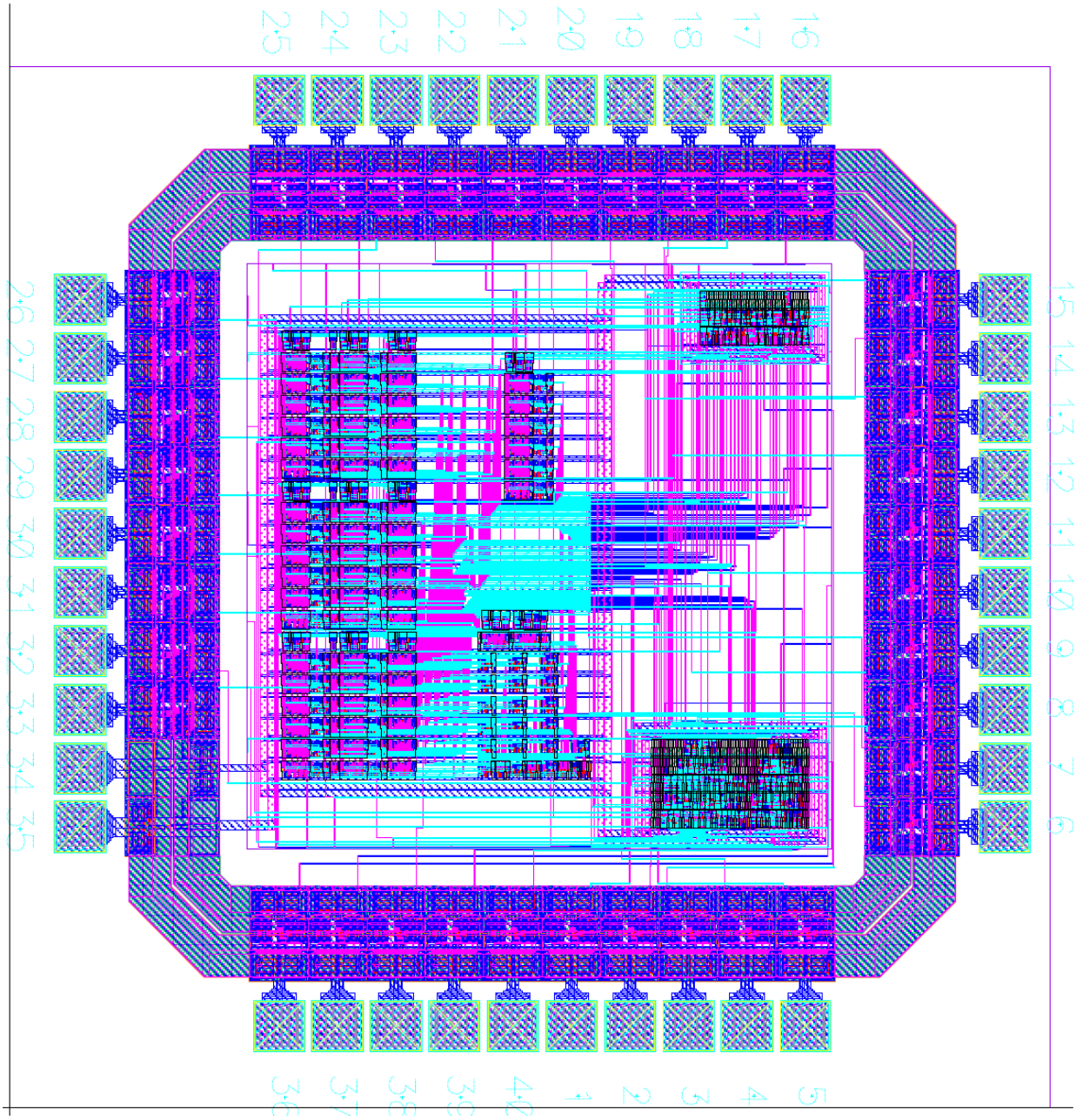


Figure 17: Chip Layout

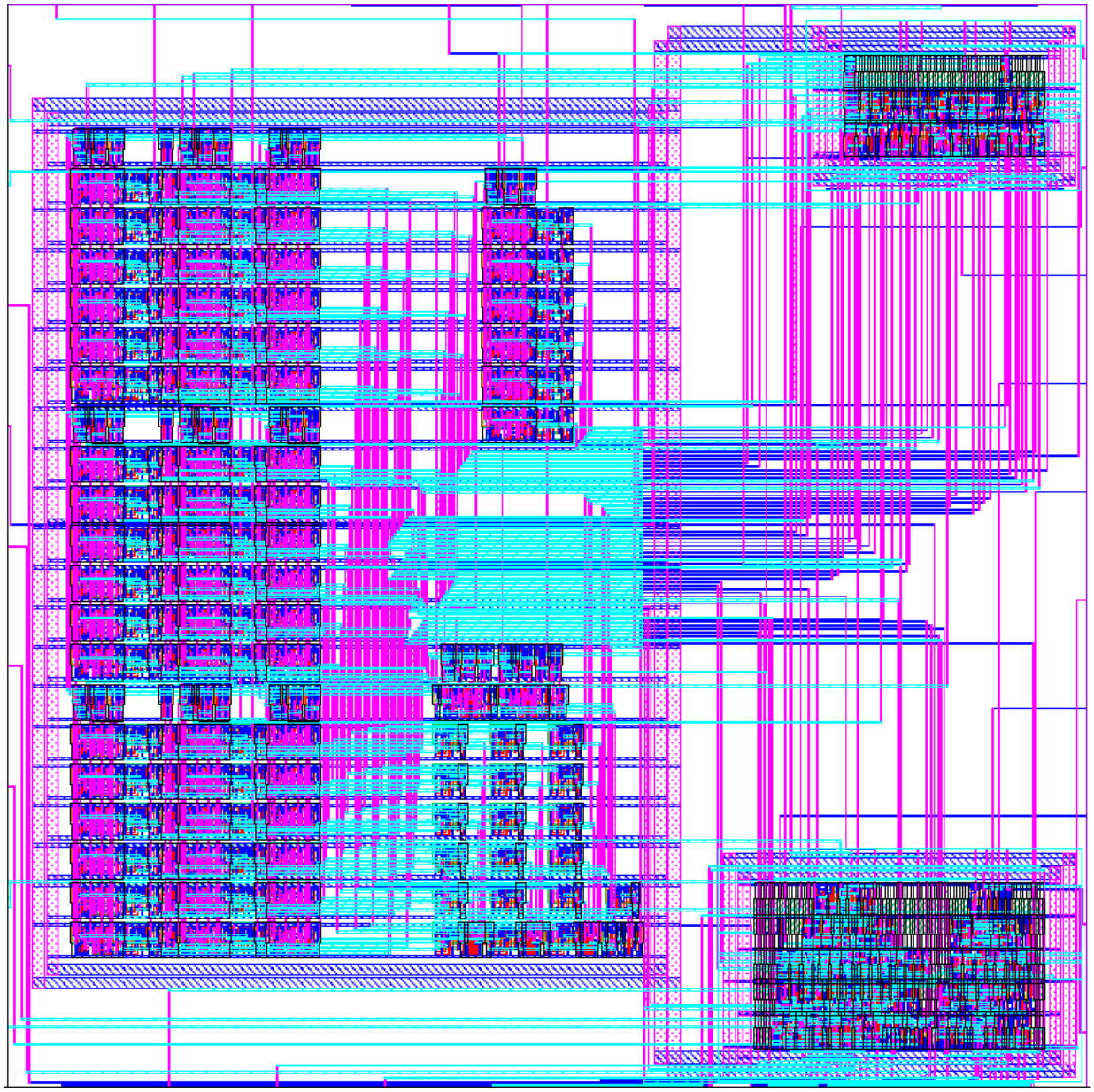


Figure 18: Core Schematic

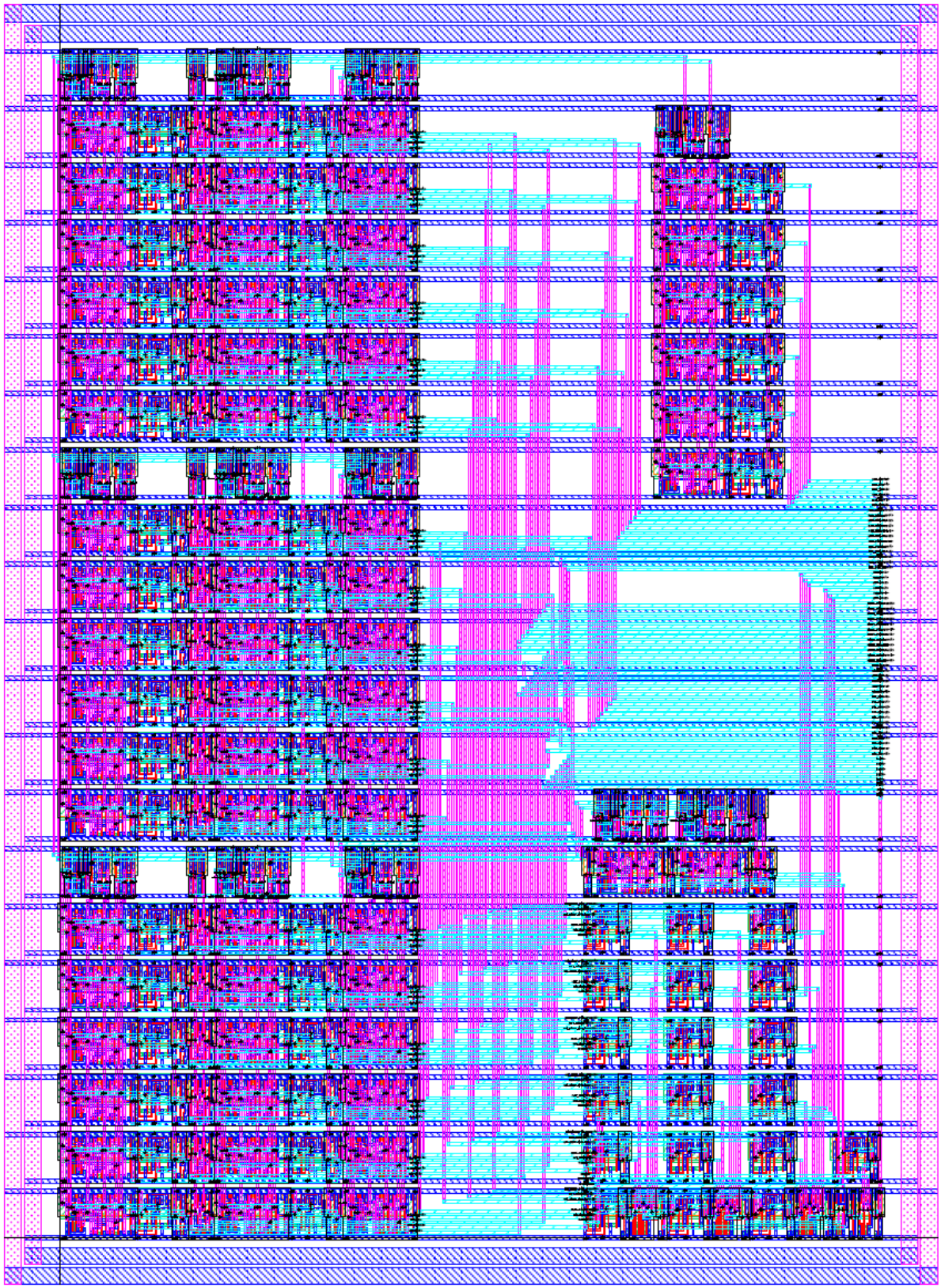


Figure 19: Datapath Schematic

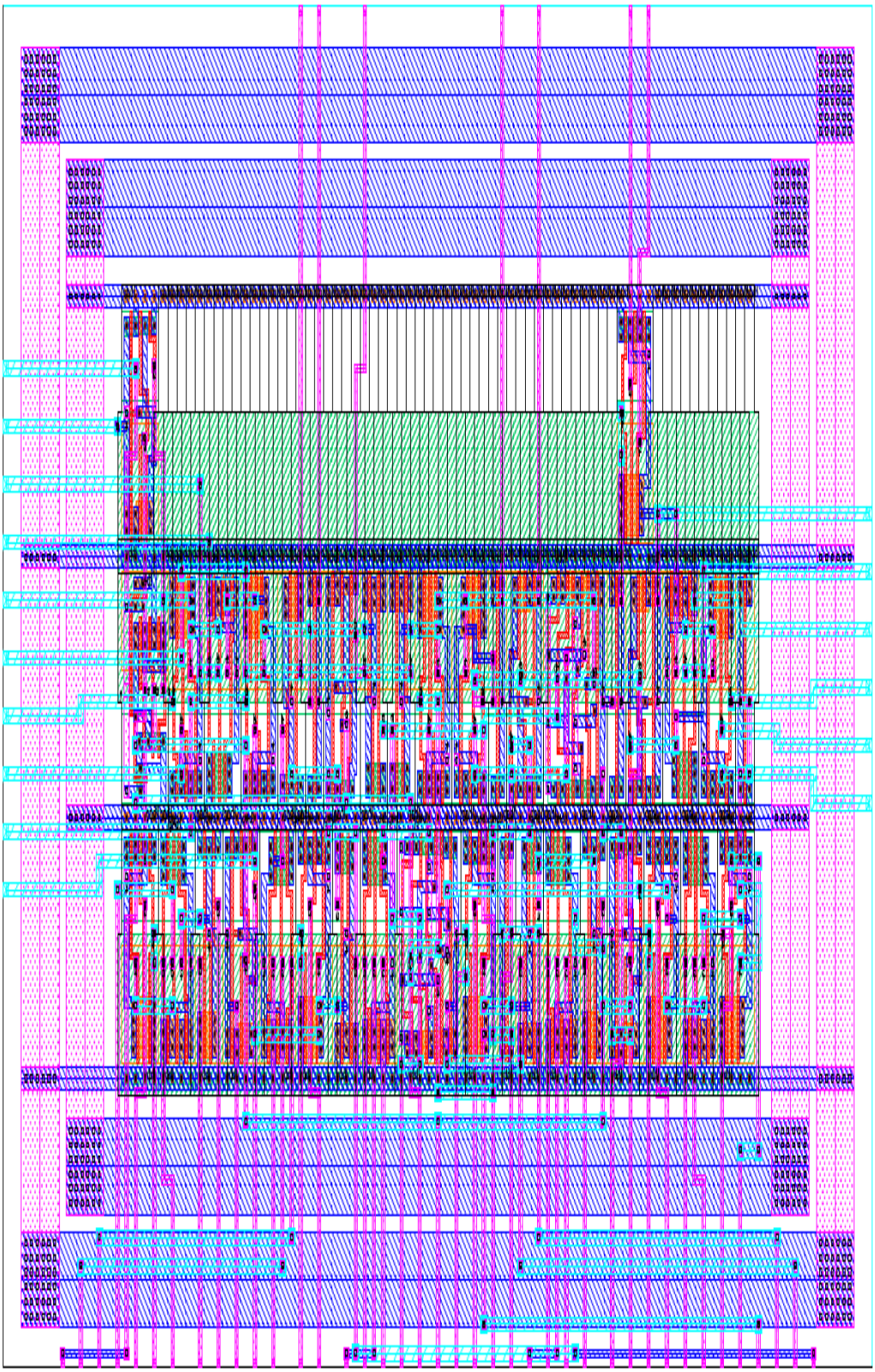


Figure 20: clockController Layout

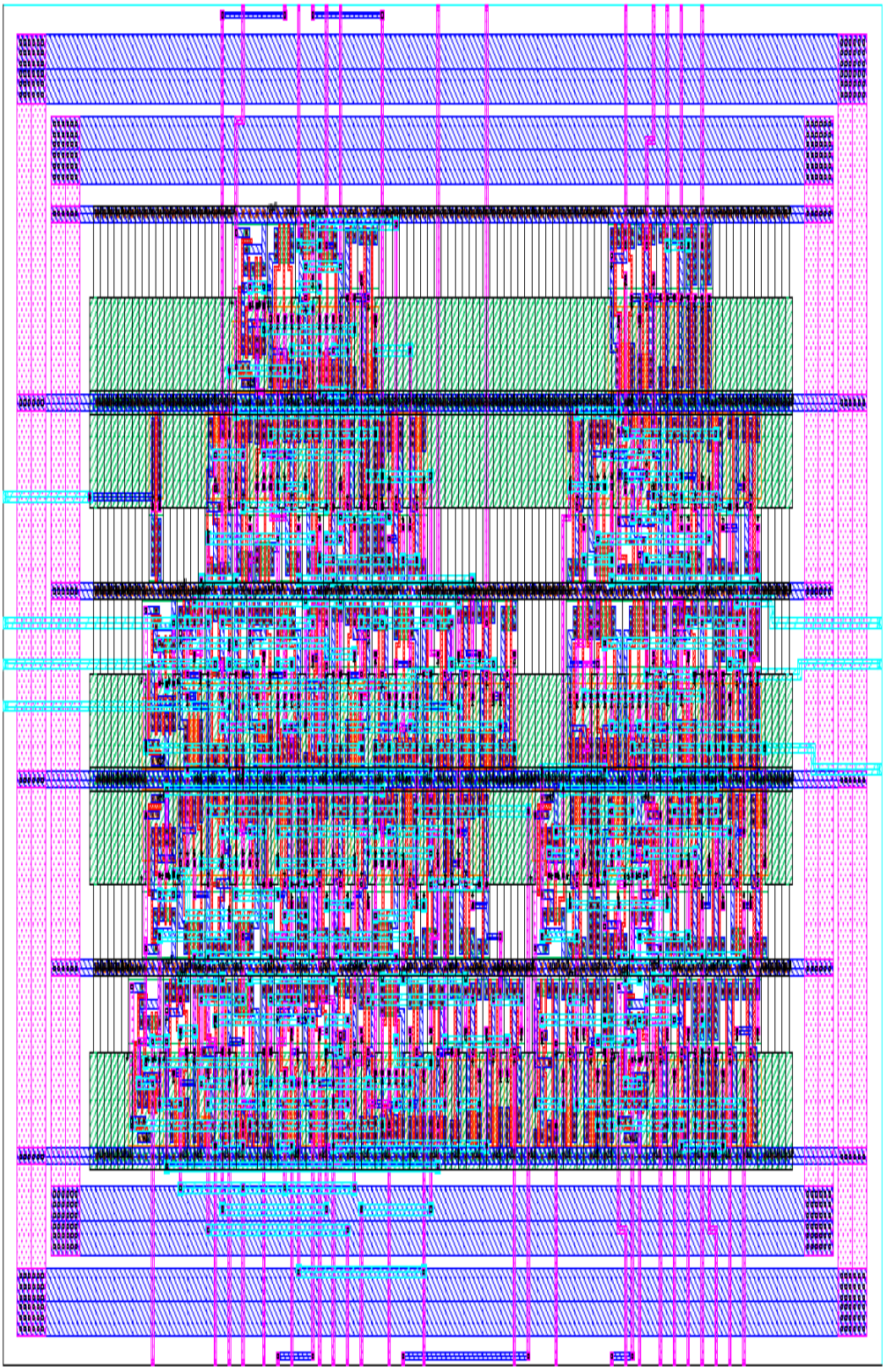


Figure 21: LEDdecoder Layout

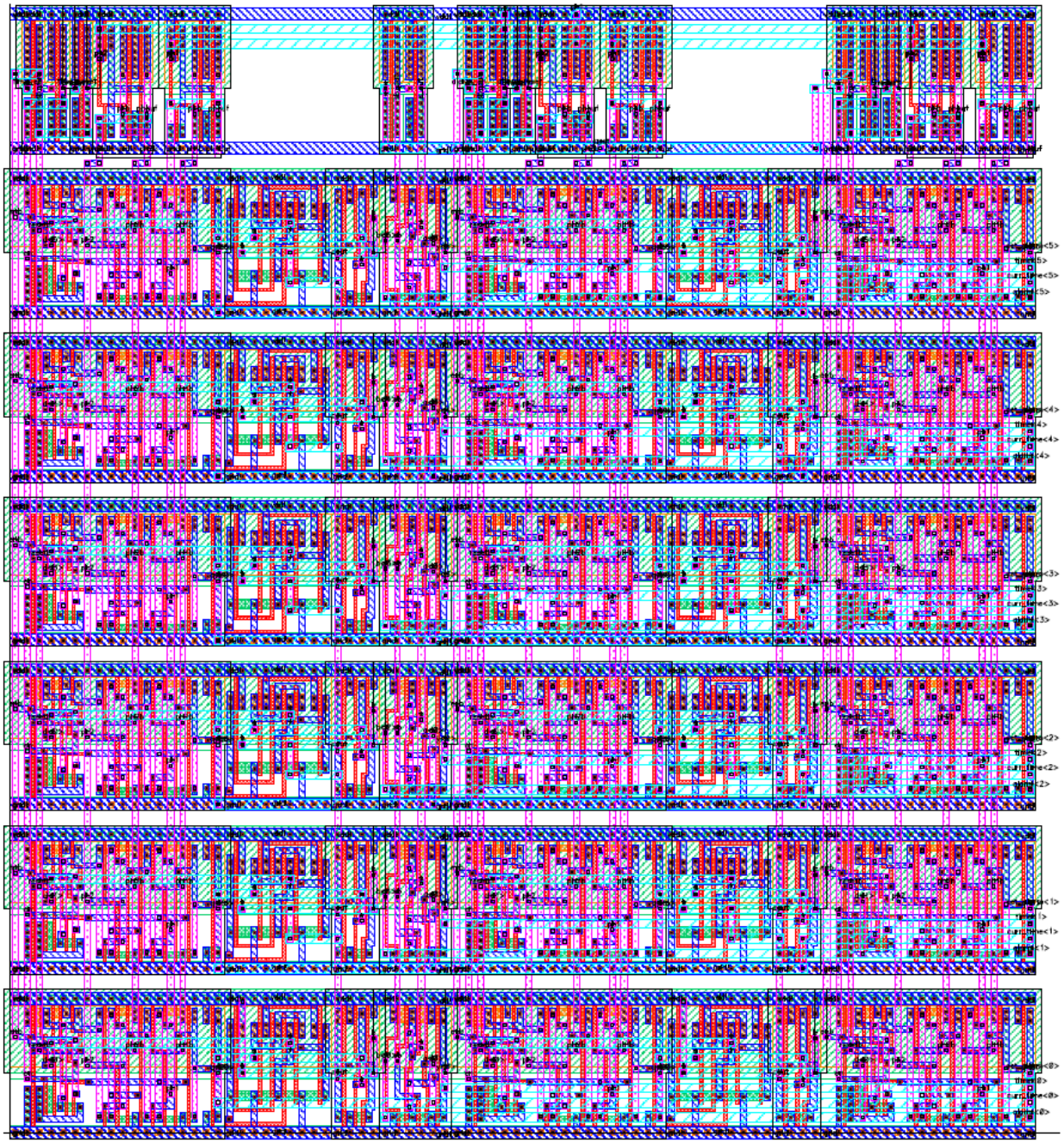


Figure 22: Alarm Layout

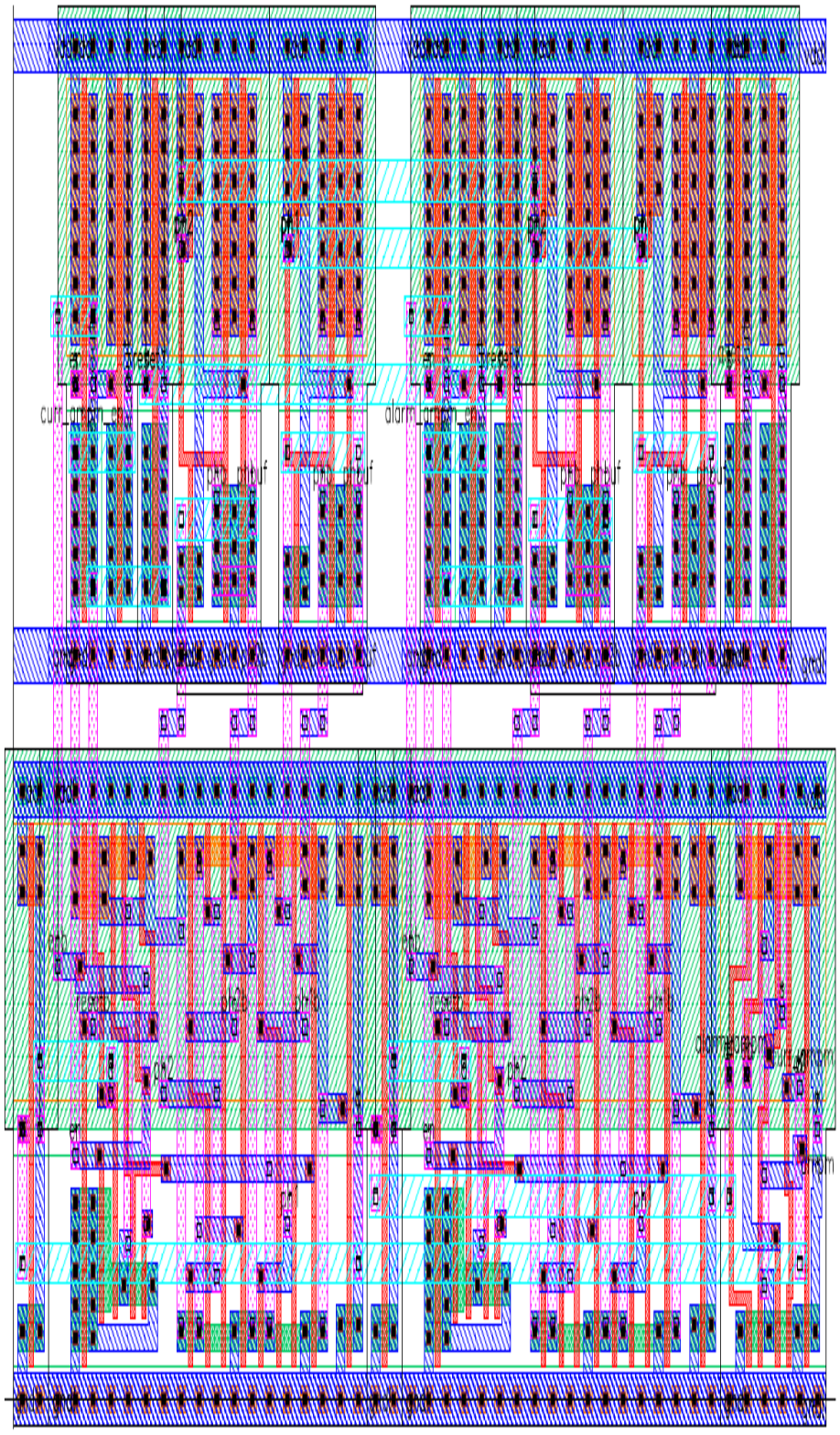


Figure 23: Apmp Layout

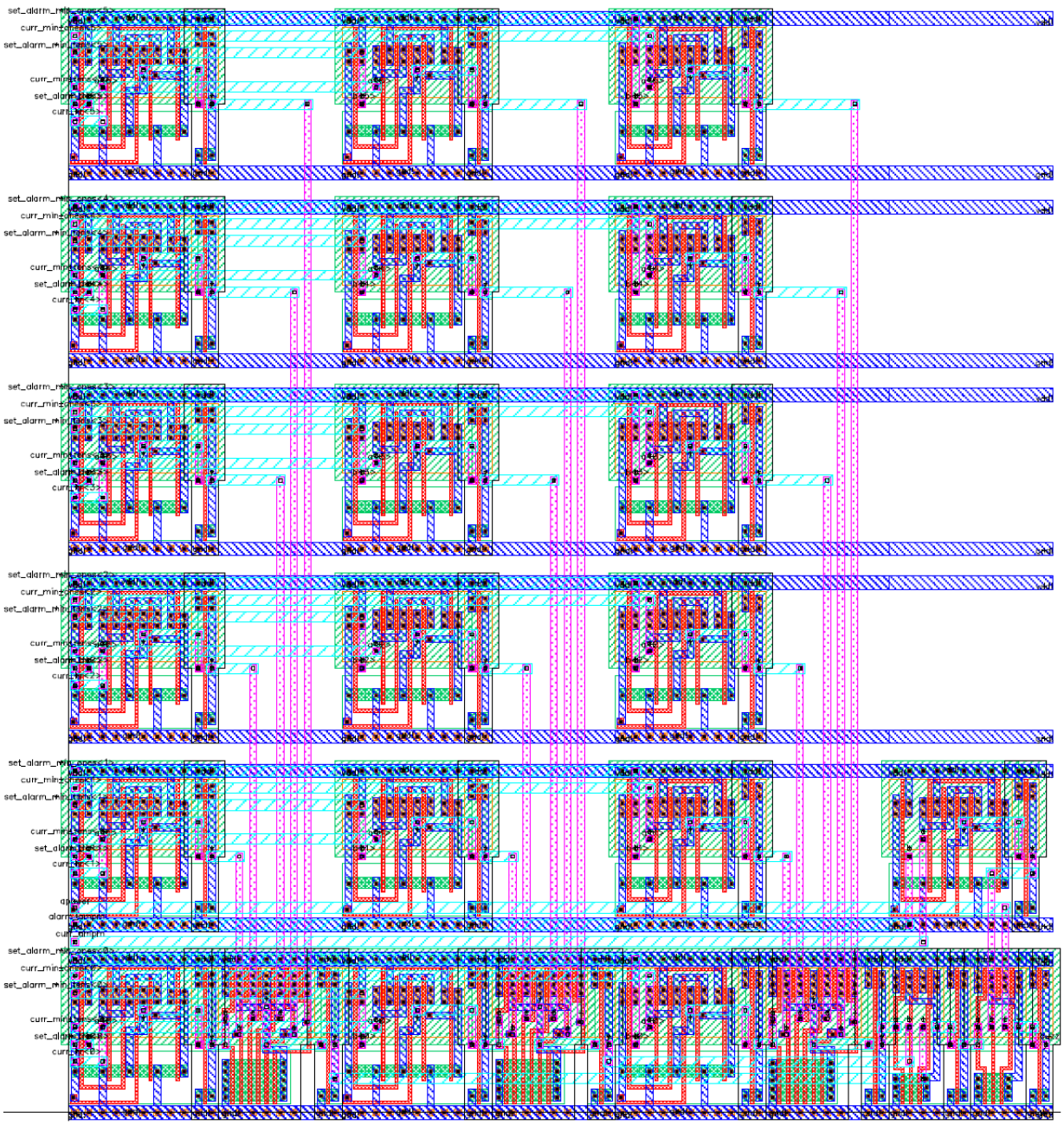


Figure 24: Buzzer Layout

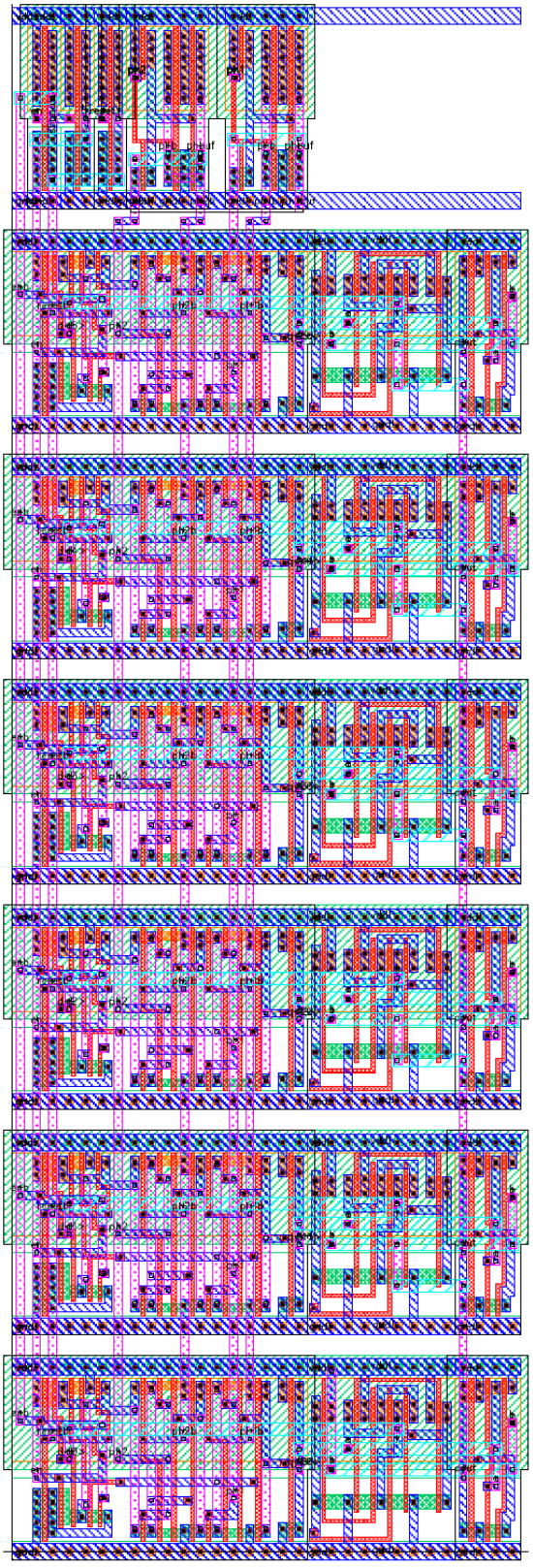


Figure 25: Counter6 Layout

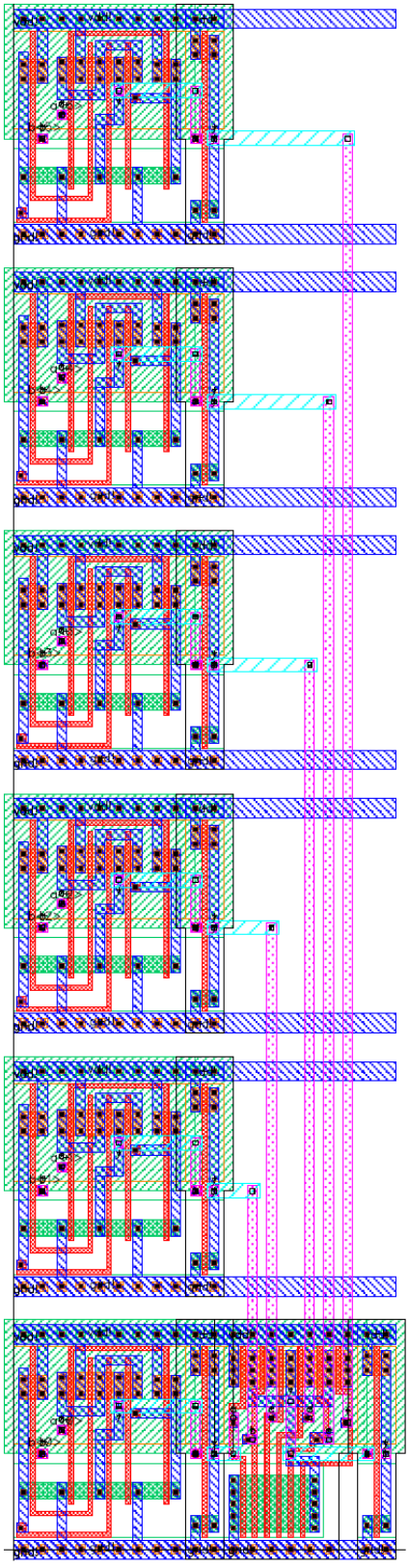


Figure 26: Comparator6 Layout

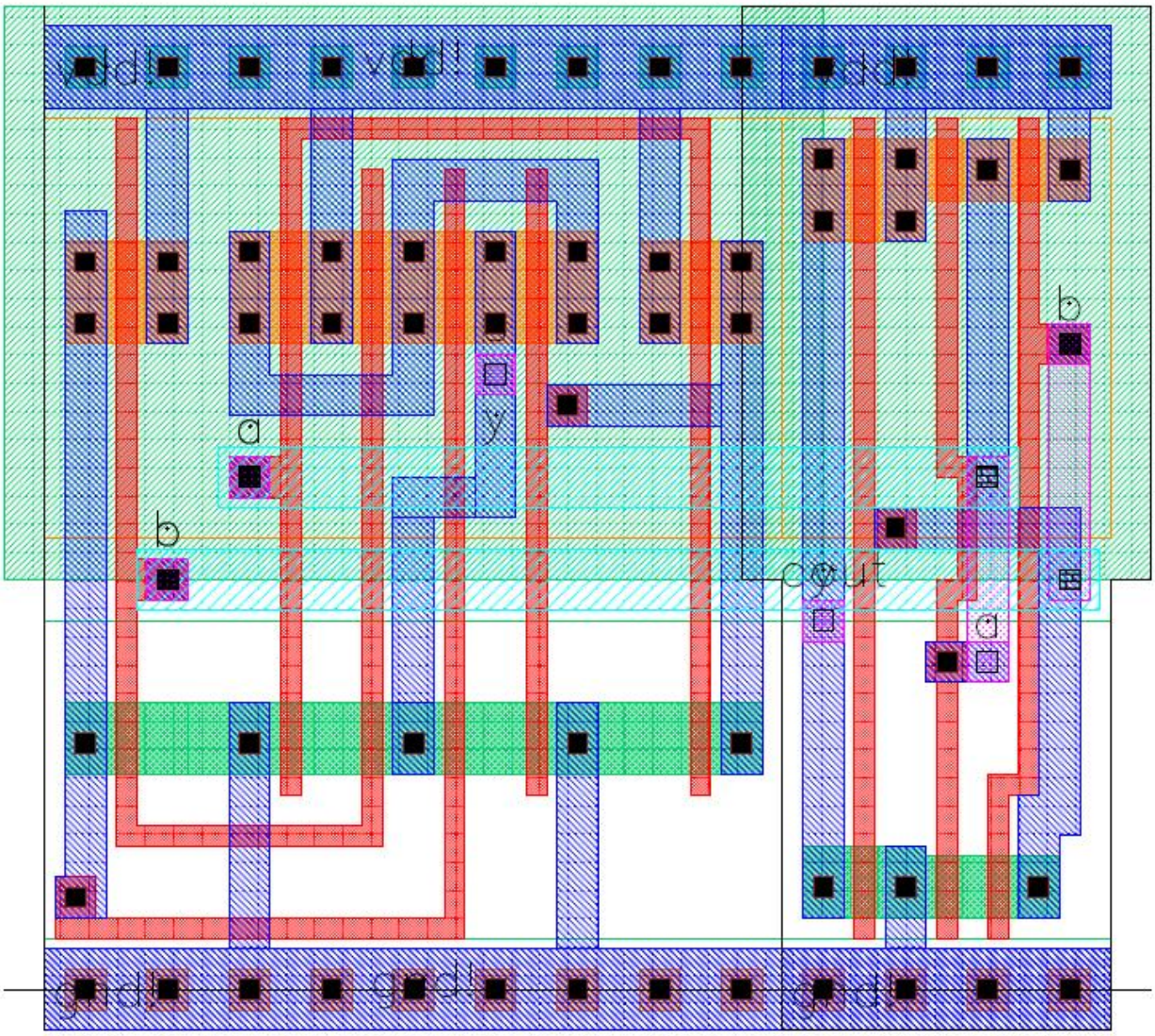


Figure 27: Half-Adder Layout

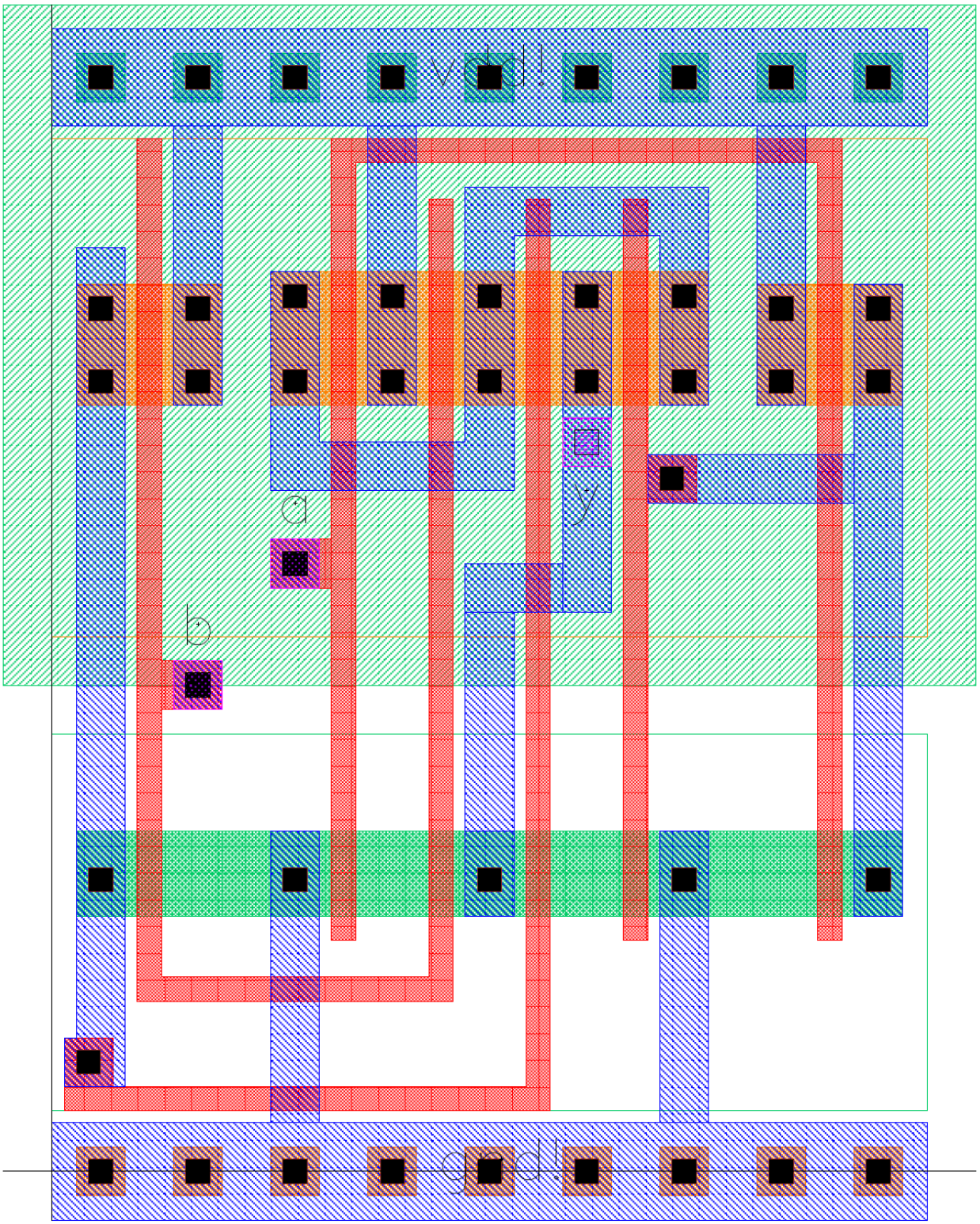


Figure 28: XOR Layout

Appendix D: Proposal Floorplan