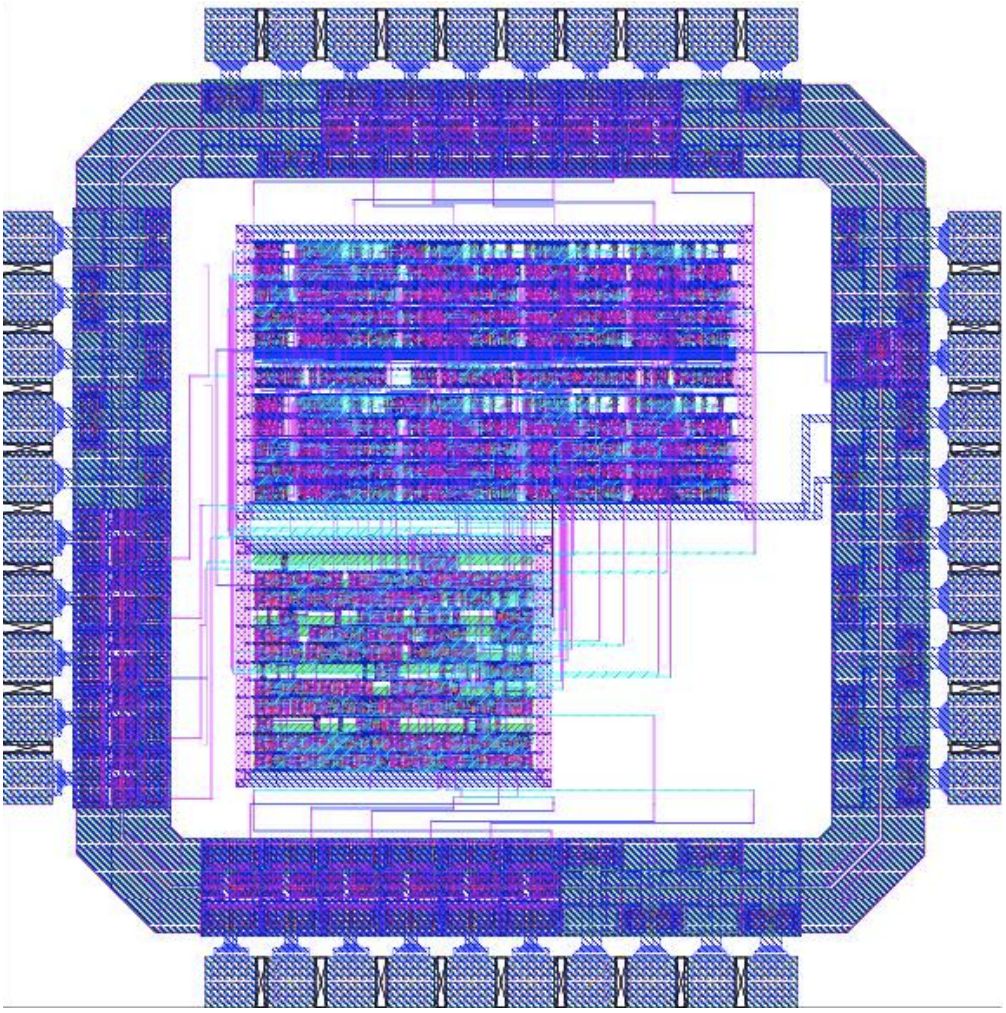

E158: CMOS VLSI Design

Binary Alarm Clock Chip

Designers:
Leo Altmann
Hannah Troisi

Advisor:
David Harris

April 19, 2010

Introduction

The goal of this project was to create a controller for an alarm clock that displays the time in binary form, similar to that available online at the Think Geek Store [1]. To ease readability, the display uses base-10 digit divisions, as on a standard digital clock, but displays the binary representation of each such digit. The time is displayed following the 24-hour convention and includes include hours, minutes and seconds. Time-division multiplexing is used to drive the LED display with a minimal number of output connections.

Current time and alarm time are set by buttons to increment hour and minute values. An additional button changes the display and incrementing functionality from the current time to the alarm time. An alarm will be triggered when the current time matches the desired alarm time if the alarm enable switch is on and will continue until the enable switch is turned off. The resulting alarm signal could be used to switch on a buzzer or equivalent alert device, or alternatively be used as control for another circuit, such as a radio or signal lamp.

The controller chip will be fabricated on the MOSIS 0.6 uM CMOS process and housed in a 40-pin DIP-form package. The total design size, including I/O pads, will not exceed 1.5 x 1.5 mm.

Specifications

The binary alarm clock chip is designed to run the hardware necessary to create a 24-hour format alarm clock that displays time in binary. The current time, displayed by default, is set by toggling the minute and hour buttons, corresponding to chip inputs minBtnPhys and hrBtnPhys respectively. When the alarm button, driving the alarmSet signal, is depressed and held, the display will show the set alarm time, which can be changed by again toggling the same minute and hour buttons. The alarm on/off switch, driving input alarmEn, determines whether or not the alarm is enabled and serves as an off switch for when the alarm goes off. An active alarm will drive the alarm signal high.

The time display will not be in true binary form, instead each digit of the time in decimal will be displayed in binary. Thus, the ones digit of the second, minute and hour values will need at maximum 4 bits to represent all possible values, the tens digit of the second and minute values will need at maximum 3 bits and the tens digit of the hours will need at maximum 2 bits. To simplify the design, a time-division multiplexed bus will be used to drive the LEDs for each digit. The chip output displayState[5:0] is a one-hot enable signal for each column of the display. The displayValue[3:0] bus passes the value for the appropriate digit as it is activated. To keep synthesized logic small, a 1kHz external clock will be used to drive the chip, as a low frequency allows for fewer required bits of state in the clock divider module.

The inputs and outputs of the binary alarm clock are summarized in Table 1.

Direction	Name	Width
input	gnd	1
input	vdd	1
input	ph1	1
input	ph2	1
input	reset	1
input	hrBtnPhys	1
input	minBtnPhys	1
input	alarmSet	1
input	alarmEn	1
output	alarm	1
output	displayState	6
output	displayValue	4

Table 1: Table of Input and Outputs to the Binary Alarm Clock Chip

Floorplan

The datapath, binary_alarm_clock (Figure 21) has an area of roughly $1470\lambda \times 2580\lambda$. The estimated floorplan in our proposal had an area of $630\lambda \times 2200\lambda$. The discrepancies in projected and actual size stem from a design change that simplified other portions of the chip at the expense of increasing the number of counters required in the datapath. Using ten 4-bit counters to count each digit of the current and alarm time individually, versus the original plan of five 7-bit counters for each hour, minute and second value, eliminated the need to convert from base-2 to base-10 in the synthesized logic. Though the implemented counters are smaller than the proposed counters, a second row of them and additional zipper bitlines were required. The proposed design also did not take power rings into account.

The synthesized portion of the design had a total area of $1250\lambda \times 1570\lambda$, which is significantly less than the total estimated area of $730\lambda \times 550\lambda$ for the display logic, $90\lambda \times 650\lambda$ for the input & alarm logic and $1000\lambda \times 1000\lambda$ for the clock divider. The input and alarm logic remained unchanged from the proposal to the verilog implementation, but the logic necessary for the display was vastly simplified by the design changes in the datapath. This eliminated the need for logic to convert a binary number to decimal and back again.

Name	Pin Number
reset	3
ph1	8
ph2	9
hrBtnPhys	10
minBtnPhys	11
alarmSet	12
alarmEn	13
alarm	21
displayState[5:0]	26-31
displayValue[3:0]	22-25

Table 2: Table of Pin Names and Numbers

Verification

The Verilog description (**Appendix A**) of the design simulates properly in the testbench. The testbench is not self-checking, but provides pulses of the relevant inputs, allowing the functionality of the chip can be hand-checked in the waveforms. The exported netlist from the design schematics also passes the testbench. The chip layout passes DRC and LVS against the chip schematic. The exported CIF of the chip layout loads correctly back into Cadence and passes DRC, but fails LVS with one yet unexplained 'dubiousData' error.

Postfabrication Test Plan

After the design is fabricated, it will be tested by building a simple alarm clock implementation. When the device is powered, proper clock signals are given and all button inputs properly tied low, the display output signals should reflect accurate operation. First, an oscilloscope will be connected to the displayState pins to verify that only one bit is high at a time and that the bits go high in sequential order. A basic display will then be constructed from six groups of four LEDs each, sharing the displayValue bus, with external transistors controlling the power to each group based on the displayState signals.

Once the display logic is shown to behave as designed, the rest of the clock's functionality will be tested by attaching buttons to the button inputs and verifying that both the main time and the alarm time increment in the prescribed manner. The alarm function will be tested by setting the alarm time and waiting for the clock to reach the set value. If all the basic functionality is shown to be correct, the clock will be left to run for an extended period of time to check for time drift. If timekeeping is consistently inaccurate, the external oscillator frequency can be tuned to compensate.

Design Time

The majority of the design time was spent on Verilog implementation, which was initially written behaviorally and later re-written structurally. About 25% of the design time was spent on the schematics as well as the layout. The total number of hours spent on this project was 135. Table 1 shows the breakdown of time spent on each component of the project.

Design Component	**Time (hours)**
Project Proposal	2
Preliminary Design	3
Leaf Cell	4
Verilog Implementation	55
Testbench	14
Schematic Implementation	25
Datapath Layout	25
Padframe & Chip Layout	7
Total	**135**

Table 3: Table of Design Time for Each Component of the Project

File Locations

The directory `/home/laltmann/binaryclock/final/` in Leo Altmann's account on chips contains a clean copy of all of the files related to this project. Table 4 summarizes the exact locations of the project files within this directory.

Design Component	**Location**
Verilog code & Self-checking Testbench	`binaryclock.sv`
Synthesis Results	`synthesis/`
All Cadence Libraries	`cadlib/`
CIF	`cif/`
PDF chip plot	`chip.pdf`
PDF of this report	`finalreport.pdf`

Table 4: Table of File Locations

References

[1] Binary Alarm Clock, *Think Geek Store*. Accessed 5 April 2010. `http://www.thinkgeek.com/homeoffice/lights/59e0/`

Appendices

Appendix A: Verilog Code

```
1
//    binaryclock.sv
3 //
//    Leo Altmann (laltmann@hmc.edu)
5 //    Hannah Troisi (htroisi@hmc.edu)
//
7 //    Verilog HDL for a binary-display alarm clock chip.
//
9
`timescale 1ns / 100ps
11
// testbench logic
13 module testbench();
      logic           ph1, ph2, reset;
15    logic           hrBtnPhys, minBtnPhys, alarmSet, alarmEn, alarm;
      logic [5:0]     displayState;
17    logic [3:0]     displayValue;

19    logic           hrBtn, minBtn, incSecOnes;
      logic [1:0]     hrTens, alarmHrTens;
21    logic [3:0]     secTens, minTens, alarmMinTens;
      logic [3:0]     secOnes, minOnes, hrOnes, alarmMinOnes, alarmHrOnes;
23
      logic [1000:0] runtime, cycleCounter;
25    logic [3:0]     exphrTens, exphrOnes, expminTens, expminOnes, expsecTens,
      logic           expsecOnes;
27
      // instantiate devices to be tested
29    // .* notation instantiates all ports in the mips module
      // with the correspondingly named signals in this module
31    core dut(.*);

33    // generate 2 phase non-overlapping clock
      always begin
35          ph1 <= 0; ph2 <= 0; #1;
            ph1 <= 1; #4;
37          ph1 <= 0; #1;
            ph2 <= 1; #4;
39    end

41    // reset system
      initial begin
43          reset = 1; #37; reset = 0;
      end
45
      // increment cycleCounter
47    always @(posedge ph1) begin
            cycleCounter <= cycleCounter + 1;
49          if (reset)
```

```
                cycleCounter <= 0;
51      end

53      initial begin
            // initialize expected runtime and times
55          runtime = 12'b110010000100;
            exphrTens = 4'b0000;
57          exphrOnes = 4'b0000;
            expminTens = 4'b0000;
59          expminOnes = 4'b1001;
            expsecTens = 4'b0000;
61          expsecOnes = 4'b0000;
            // set inputs low
63          hrBtnPhys = 0; minBtnPhys = 0; alarmSet = 0; alarmEn = 0; alarm = 0;
            // set starting time at 23:00
65          // hrTens = 2'b10; hrOnes = 4'b0011;
            // assert alarmEn
67          #33; alarmEn = 1; $display("AlarmEn On");
            //Alarm should go high at 00:00 (default)
69          // turn alarm off after 00:00 (1hr = #223260)
            #223333; alarmEn = 0;
71
            // assert hrBtnPhys for 30 cyles sometime after 00:00
73          #223663; hrBtnPhys = 1; #300; hrBtnPhys = 0;
            // assert minBtnPhys for 65 cycles (add 10 min to expmin)
75          #663; minBtnPhys = 1; #650; minBtnPhys = 0;

77          // assert alarmEn
            #53; alarmEn = 1;
79          // set alarm to 9:05
            #33; alarmSet = 1; #33 hrBtnPhys = 1; #93; hrBtnPhys = 0;
81          #33; minBtnPhys = 1; #93; minBtnPhys = 0; alarmSet = 0;
            //turn alarm off after 9:05
83          //#500000; alarmEn = 0;

85      end

87  endmodule

89  module core
        (input  logic          ph1, ph2, reset,
91       input  logic          hrBtnPhys, minBtnPhys, alarmSet, alarmEn,
         output logic          alarm,
93       output logic [5:0]    displayState,
         output logic [3:0]    displayValue);
95
        logic          hrBtn, minBtn, incSecOnes;
97      logic [1:0]    hrTens, alarmHrTens;
        logic [3:0]    secTens, minTens, alarmMinTens;
99      logic [3:0]    secOnes, minOnes, hrOnes, alarmMinOnes, alarmHrOnes;

101     datapath  dp(ph1, ph2, reset, incSecOnes, hrBtn, minBtn, alarmSet, alarmEn,
                  hrTens, alarmHrTens, secTens, minTens, alarmMinTens, secOnes,
103               minOnes,hrOnes, alarmMinOnes, alarmHrOnes, alarm);
```

```
105       core_synth      cs(ph1, ph2, reset, hrBtnPhys, minBtnPhys, alarmSet, hrTens,
                          alarmHrTens, secTens, minTens, alarmMinTens, secOnes, minOnes,
107                       hrOnes, alarmMinOnes, alarmHrOnes, hrBtn, minBtn, incSecOnes,
                          displayState, displayValue);
109 endmodule

111 module core_synth
          (input    logic          ph1, ph2, reset, hrBtnPhys, minBtnPhys, alarmSet,
113        input    logic [1:0]    hrTens, alarmHrTens,
           input    logic [3:0]    secTens, minTens, alarmMinTens,
115        input    logic [3:0]    secOnes, minOnes, hrOnes, alarmMinOnes, alarmHrOnes,
           output   logic          hrBtn, minBtn, incSecOnes,
117        output   logic [5:0]    displayState,
           output   logic [3:0]    displayValue);
119
          buttonInput     minbtninp(ph1, ph2, reset, minBtnPhys, minBtn);
121
          buttonInput     hrbtninp(ph1, ph2, reset, hrBtnPhys, hrBtn);
123
          clkDivider #(1000,10)     clkdiv(ph1, ph2, reset, incSecOnes);  // For 1kHz osc
125
          displayDecoder ddec(ph1, ph2, reset, alarmSet, hrOnes, alarmHrOnes, minOnes,
127                       alarmMinOnes, secOnes, minTens, alarmMinTens, secTens, hrTens,
                          alarmHrTens, displayState, displayValue);
129 endmodule

131 module clkDivider #(parameter SEC = 20, SECBITS = 10)
          (input  logic   ph1, ph2, reset,
133        output logic   incSecOnes);

135       logic [SECBITS-1:0] tickCount, newTickCount;
          logic      newIncSecOnes;
137
          flop #(SECBITS)      tickFlop(ph1, ph2, newTickCount, tickCount);
139       flop #(1)       incSecFlop(ph1, ph2, newIncSecOnes, incSecOnes);

141       always_comb
               if (reset | incSecOnes) begin
143                 newTickCount   <= 0;
                    newIncSecOnes  <= 0;
145            end
               else begin
147                 newTickCount <= tickCount + 1;
                    if (tickCount == SEC)
149                     newIncSecOnes <= 1;
               end
151 endmodule

153 module buttonInput  // converts a continuous input to a one cycle long output
          (input    logic      ph1, ph2, reset, btnPhys,
155        output   logic      btn);

157       logic [1:0]     state, nextstate;
```

```
159       always_comb
              case (state)
161               2'b00:    if (btnPhys)   nextstate <= 2'b01;
                            else       nextstate <= 2'b00;
163               2'b01:    if (btnPhys)   nextstate <= 2'b10;
                            else       nextstate <= 2'b00;
165               2'b10:    if (btnPhys)   nextstate <= 2'b10;
                            else       nextstate <= 2'b00;
167               default:       nextstate <= 2'b00;
              endcase
169
          flopenr   #(2) floppy(ph1, ph2, reset, 1'b1, nextstate, state);
171
          assign btn = (state == 2'b01);
173
      endmodule
175
      module datapath
177         (input  logic          ph1, ph2, reset, incSecOnes,
           input  logic          hrBtn, minBtn, alarmSet, alarmEn,
179        output logic [1:0]    hrTens, alarmHrTens,
           output logic [3:0]    secTens, minTens, alarmMinTens,
181        output logic [3:0]    secOnes, minOnes, hrOnes, alarmMinOnes,alarmHrOnes,
           output logic          alarm);
183
          // Increment trigger signals
185       logic     incSecTens, incMinOnes, incMinTens, incHrOnes, incHrTens;
          logic     incAlarmMinTens, incAlarmMinOnes, incAlarmHrTens, incAlarmHrOnes;
187       logic     onesTrig, alarmOnesTrig, rollover, alarmRollover;
          logic     incMinOnesOv, incHrOnesOv;
189
          // Increment ones digit of min/hr for time and alarm
191       // for either rollover or button
          assign    incMinOnesOv = ((minBtn & ~alarmSet) | incMinOnes);
193       assign    incHrOnesOv = ((hrBtn & ~alarmSet) | incHrOnes);
          assign    incAlarmMinOnes = (minBtn & alarmSet);
195       assign    incAlarmHrOnesOv  = (hrBtn & alarmSet);

197       // Counter array for main timekeeping
          counter #(4, 10)    secOnesCounter(ph1, ph2, reset, incSecOnes, secOnes,
199                           incSecTens);
          counter #(4, 6)     secTensCounter(ph1, ph2, reset, incSecTens, secTens,
201                           incMinOnes);
          counter #(4, 10)    minOnesCounter(ph1, ph2, reset, incMinOnesOv, minOnes,
203                           incMinTens);
          counter #(4, 6)     minTensCounter(ph1, ph2, reset, incMinTens, minTens,
205                           incHrOnes);
          counterHrOnes #(4, 10, 4)    hrOnesCounter(ph1, ph2, reset, incHrOnesOv,
207                           rollover, hrOnes, incHrTens, onesTrig);
          counterHrTens #(2, 2)  hrTensCounter(ph1, ph2, reset, incHrTens, onesTrig,
209                           hrTens, rollover);

211       // Alarm counter array
```

```
        counter #(4, 10)     alarmMinOnesCounter(ph1, ph2, reset, incAlarmMinOnes,
213                          alarmMinOnes, incAlarmMinTens);
        counter #(4, 6)      alarmMinTensCounter(ph1, ph2, reset, incAlarmMinTens,
215                          alarmMinTens, incAlarmHrOnes);
                             // incAlarmHrOnes not needed
217     counterHrOnes #(4, 10, 4)      alarmHrOnesCounter(ph1, ph2, reset,
                                       incAlarmHrOnesOv, alarmRollover, alarmHrOnes,
219                                    incAlarmHrTens, alarmOnesTrig);
        counterHrTens #(2, 2)     alarmHrTensCounter(ph1, ph2, reset, incAlarmHrTens,
221                               alarmOnesTrig, alarmHrTens, alarmRollover);

223     // alarm combinational logic
        logic      secOnesGo, secTensGo, minOnesGo, minTensGo, hrOnesGo, hrTensGo;
225     logic      combinedGo;

227     comparator #(4)      minOnesComp(minOnes, alarmMinOnes, minOnesGo);
        comparator #(4)      minTensComp(minTens, alarmMinTens, minTensGo);
229     comparator #(4)      hrOnesComp(hrOnes, alarmHrOnes, hrOnesGo);
        comparator #(2)      hrTensComp(hrTens, alarmHrTens, hrTensGo);
231     comparator #(4)      secOnesComp(secOnes, 4'b0, secOnesGo);
        comparator #(4)      secTensComp(secTens, 4'b0, secTensGo);
233
        assign  combinedGo = (minOnesGo & minTensGo & hrOnesGo & hrTensGo & secOnesGo &
235                           secTensGo);
        assign  alarm = (alarmEn & (alarm | combinedGo));
237
    endmodule
239

241 module counterOnes #(parameter WIDTH = 4, ROLLOVER = 10)
        (input  logic   ph1, ph2, reset, inc,
243      output logic   [WIDTH-1:0] compOut,
         output logic   incNextOneCycle);
245
        logic [WIDTH-1:0]   adderOut;
247     logic [WIDTH-1:0]   flopIn;
        logic               incNext;
249     logic               comparatorOut;

251     flopenr         #(WIDTH)  timeFlop(ph1, ph2, reset, inc, flopIn, compOut);

253     adder           #(WIDTH)  addar(4'b0001, compOut, 1'b0, adderOut);

255     comparator      #(WIDTH)  comp(adderOut, ROLLOVER, comparatorOut);

257     assign                    flopIn = adderOut & {~comparatorOut, ~comparatorOut,
                                  ~comparatorOut, ~comparatorOut};
259
        assign                    incNext = inc & comparatorOut;
261
        incNextForOneCycle        increment(ph1, ph2, reset, incNext, incNextOneCycle);
263
    endmodule
265
```

```
module counterTens #(parameter WIDTH = 4, ROLLOVER = 6)
    (input  logic    ph1, ph2, reset, inc,
     output logic    [WIDTH-1:0] compOut,
     output logic    incNextOneCycle);

    logic [WIDTH-1:0]    adderOut;
    logic [WIDTH-1:0]    flopIn;
    logic                incNext;
    logic                comparatorOut;

    flopenr         #(WIDTH)  timeFlop(ph1, ph2, reset, inc, flopIn, compOut);

    adder           #(WIDTH)  addar(4'b0001, compOut, 1'b0, adderOut);

    comparator      #(WIDTH)  comp(adderOut, ROLLOVER, comparatorOut);

    assign                    flopIn = adderOut & {~comparatorOut, ~comparatorOut,
                              ~comparatorOut, ~comparatorOut};

    assign                    incNext = inc & comparatorOut;

    incNextForOneCycle        increment(ph1, ph2, reset, incNext, incNextOneCycle);

endmodule


module counterHrOnes #(parameter WIDTH = 4, ROLLOVER = 10, TRIGVAL = 4)
    (input logic    ph1, ph2, reset, inc, rollover,
    output logic    [WIDTH-1:0] compOut,
    output logic    incNextOneCycle, onesTrig);

    logic [WIDTH-1:0]    adderOut;
    logic [WIDTH-1:0]    flopIn;
    logic                incNext;
    logic                comparatorNineOut;
    logic                comparatorFourOut;

    flopenr #(WIDTH)     timeFlop(ph1, ph2, (reset | rollover), inc, flopIn,
                         compOut);

    adder #(WIDTH)       addar(4'b0001, compOut, 1'b0, adderOut);

    comparator #(WIDTH)  compnine(adderOut, ROLLOVER, comparatorNineOut);

    assign               flopIn = adderOut & {~comparatorNineOut,
                         ~comparatorNineOut, ~comparatorNineOut,
                         ~comparatorNineOut};

    assign               incNext = inc & comparatorNineOut;

    incNextForOneCycle   increment(ph1, ph2, reset, incNext, incNextOneCycle);

    comparator #(WIDTH)  compfour(flopIn, TRIGVAL, comparatorFourOut);
```

```
        flopenr #(1)          timeFlop2(ph1, ph2, reset, inc, comparatorFourOut,
321                           onesTrig);

323 endmodule

325 module counterHrTens #(parameter WIDTH = 4)
        (input logic    ph1, ph2, reset, inc, onesTrig,
327     output logic    [WIDTH-1:0] compOut,
        output logic    rollover);
329
        logic [WIDTH-1:0]    nextcompOut;
331
        assign     nextcompOut[0] = ( ( (~reset & inc) & (~|{compOut[1],compOut[0]}) ) |
333                               ( (~compOut[1] & compOut[0]) & (~|{reset, inc}) ) );

335     assign     nextcompOut[1] = ( ( (compOut[1] & ~compOut[0]) & (~|{reset, onesTrig
            }) ) |
                                  ( (~reset & inc) & (~compOut[1] & compOut[0]) ) );
337
        flopenr #(WIDTH)          timeFlop(ph1, ph2, reset, (inc | rollover),
339                               nextcompOut, compOut);

341     assign                    rollover = ( (~compOut[0] & compOut[1]) & onesTrig );
                                  // rollover = (rolloverTens & onesTrig) , onesTrig = (
                                      hrOnes == 3)
343
    endmodule
345
    module incNextForOneCycle     // converts continuous input to one cycle long output
347     (input logic    ph1, ph2, reset, incNext,
         output logic   incNextOneCycle);
349
        logic [1:0]     state, nextstate;
351
        always_comb
353         case (state)
                2'b00:    if (incNext)    nextstate <= 2'b01;
355                       else              nextstate <= 2'b00;
                2'b01:    if (incNext)    nextstate <= 2'b10;
357                       else              nextstate <= 2'b00;
                2'b10:    if (incNext)    nextstate <= 2'b10;
359                       else              nextstate <= 2'b00;
                default:                    nextstate <= 2'b00;
361         endcase

363     flopenr    #(2) floppy(ph1, ph2, reset, 1'b1, nextstate, state);

365     assign incNextOneCycle = (state == 2'b01);

367 endmodule

369 // made with an xnor (our custom leaf cell)
    module  comparator #(parameter WIDTH = 8)
371             (input  logic [WIDTH-1:0] a, b,
```

```
                output logic         y);
373
            logic [WIDTH-1:0]    temp;
375         assign temp = (a ~^ b);
            assign y = &temp;
377 endmodule

379 // Various modules (adder, flopenr, mux3, flop, latch) copied from mips.sv

381 module adder #(parameter WIDTH = 8)
                (input  logic [WIDTH-1:0] a, b,
383              input  logic         cin,
                 output logic [WIDTH-1:0] y);
385
            assign y = a + b + cin;
387
    endmodule
389
    module flopenr #(parameter WIDTH = 8)
391              (input  logic             ph1, ph2, reset, en,
                  input  logic [WIDTH-1:0] d,
393               output logic [WIDTH-1:0] q);

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

397         assign resetval = 0;

399         mux3 #(WIDTH) enrmux(q, d, resetval, {reset, en}, d2);
            flop #(WIDTH) f(ph1, ph2, d2, q);
401
    endmodule
403
    module mux3 #(parameter WIDTH = 8)
405             (input  logic [WIDTH-1:0] d0, d1, d2,
                 input  logic [1:0]       s,
407              output logic [WIDTH-1:0] y);

409         always_comb
            casez (s)
411              2'b00: y = d0;
                 2'b01: y = d1;
413              2'b1?: y = d2;
            endcase
415 endmodule

417 module flop #(parameter WIDTH = 8)
            (input  logic             ph1, ph2,
419          input  logic [WIDTH-1:0] d,
             output logic [WIDTH-1:0] q);
421
            logic [WIDTH-1:0] mid;
423
            latch #(WIDTH) master(ph2, d, mid);
425         latch #(WIDTH) slave(ph1, mid, q);
```

```
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

module displayDecoder
     (input  logic           ph1, ph2, reset, alarmSet,
      input  logic   [3:0]   hrOnes, alarmHrOnes, minOnes, alarmMinOnes, secOnes,
      input  logic   [3:0]   minTens, alarmMinTens, secTens,
      input  logic   [1:0]   hrTens, alarmHrTens,
      output logic   [5:0]   displayState,
      output logic   [3:0]   displayValue);

     logic [10:0]   count, countOut;
     logic [5:0]    nextDisplayState;
     logic [3:0]    nextDisplayValue;

     flopenr #(11)  countflop(ph1,ph2, reset, 1'b1, count, countOut);
     flopenr #(6)   dispStFlp(ph1,ph2, reset, 1'b1, nextDisplayState, displayState);
     flopenr #(4)   dispVlFlp(ph1,ph2, reset, 1'b1, nextDisplayValue, displayValue);

     always_comb begin
          if (reset) begin
               nextDisplayState <= 6'b000000;
               nextDisplayValue <= 0;
               count <= 0;
               end
          else begin
               if (countOut == 3) begin // Max counter value
                    count <= 0;
                    // Update 1-hot lines
                    case (displayState)
                         6'b000000:     nextDisplayState <= 6'b000001;
                         6'b000001:      nextDisplayState <= 6'b000010;
                         6'b000010:      nextDisplayState <= 6'b000100;
                         6'b000100:      nextDisplayState <= 6'b001000;
                         6'b001000:      nextDisplayState <= 6'b010000;
                         6'b010000:      nextDisplayState <= 6'b100000;
                         6'b100000:     nextDisplayState <= 6'b000001;
                         default:        nextDisplayState <= 6'b000001;
                    endcase
               end
               else begin
                    count <= countOut + 1;
                    nextDisplayState <= displayState;
                    end
               end
          if (alarmSet)
```

```
                case (displayState)
481                 6'b000001:      nextDisplayValue <= 0;
                    6'b000010:      nextDisplayValue <= 0;
483                 6'b000100:      nextDisplayValue <= alarmMinOnes;
                    6'b001000:      nextDisplayValue <= alarmMinTens;
485                 6'b010000:      nextDisplayValue <= alarmHrOnes;
                    6'b100000:      nextDisplayValue <= {2'b0, alarmHrTens};
487                 default:         nextDisplayValue <= 0;
                endcase
489         else
                case (displayState)
491                 6'b000001:      nextDisplayValue <= secOnes;
                    6'b000010:      nextDisplayValue <= secTens;
493                 6'b000100:      nextDisplayValue <= minOnes;
                    6'b001000:      nextDisplayValue <= minTens;
495                 6'b010000:      nextDisplayValue <= hrOnes;
                    6'b100000:      nextDisplayValue <= {2'b0, hrTens};
497                 default:        nextDisplayValue <= 0;
                endcase
499     end
endmodule
```

Appendix B: Schematics

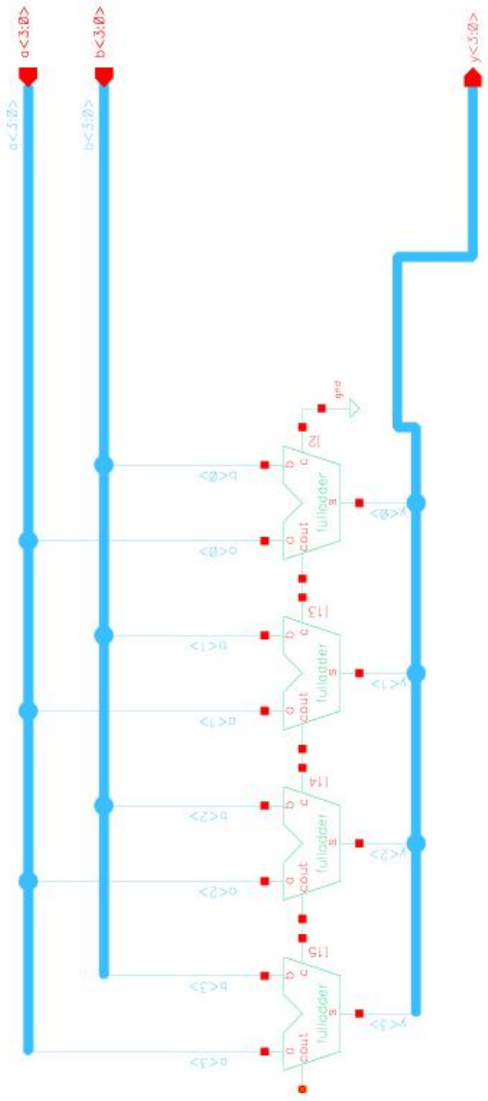

Figure 1: adder4 Schematic

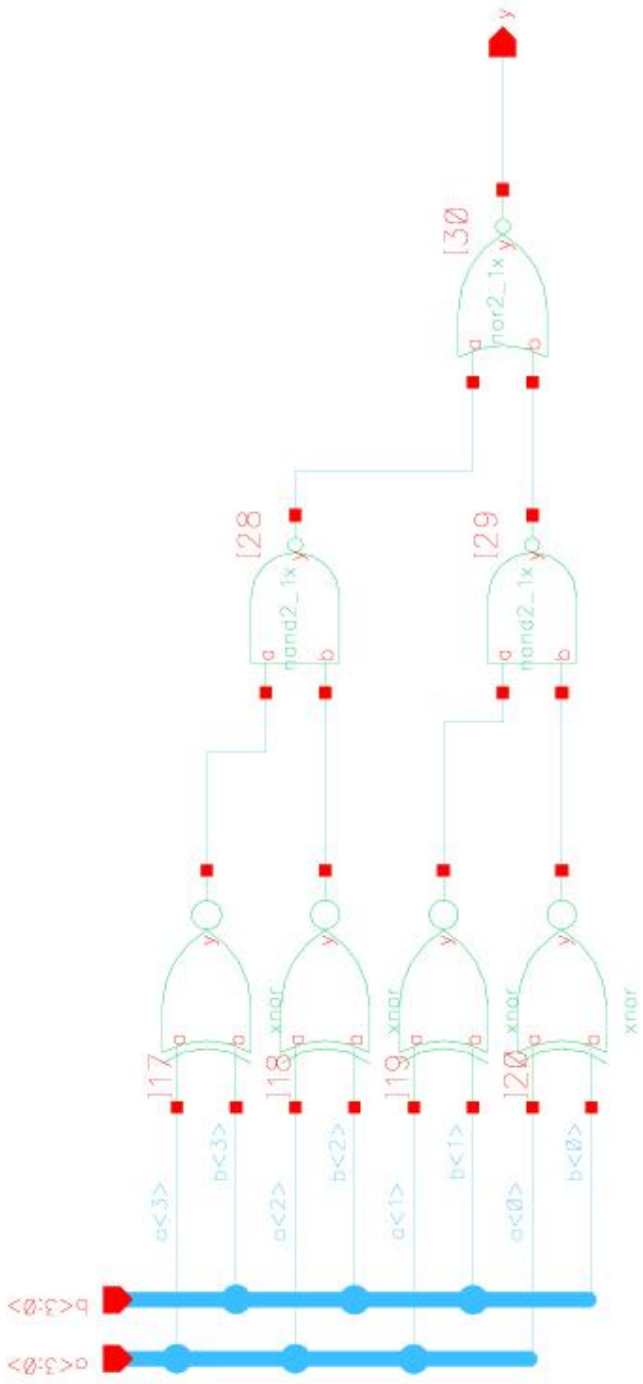

Figure 2: comparator4 Schematic

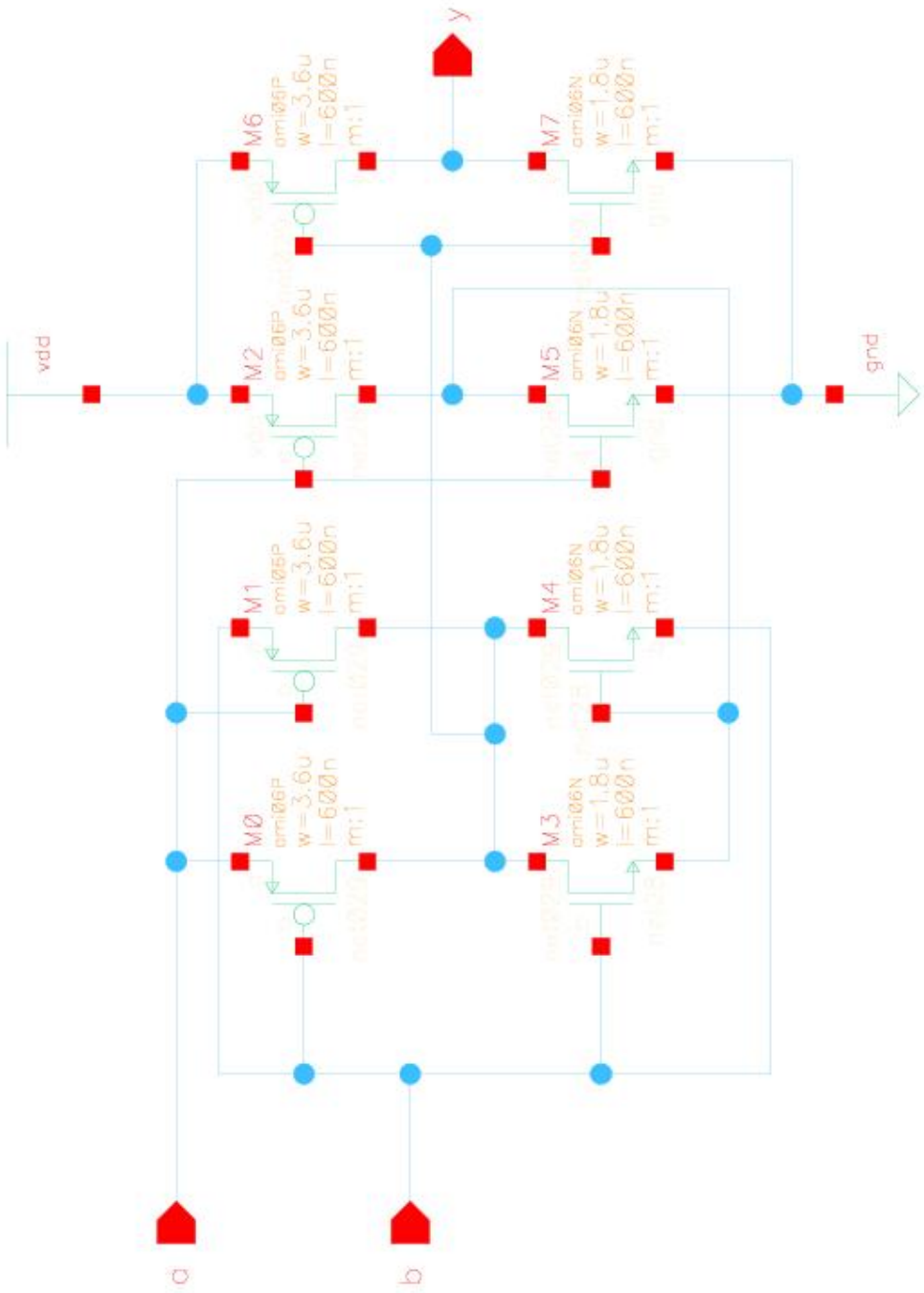

Figure 3: xnor Schematic

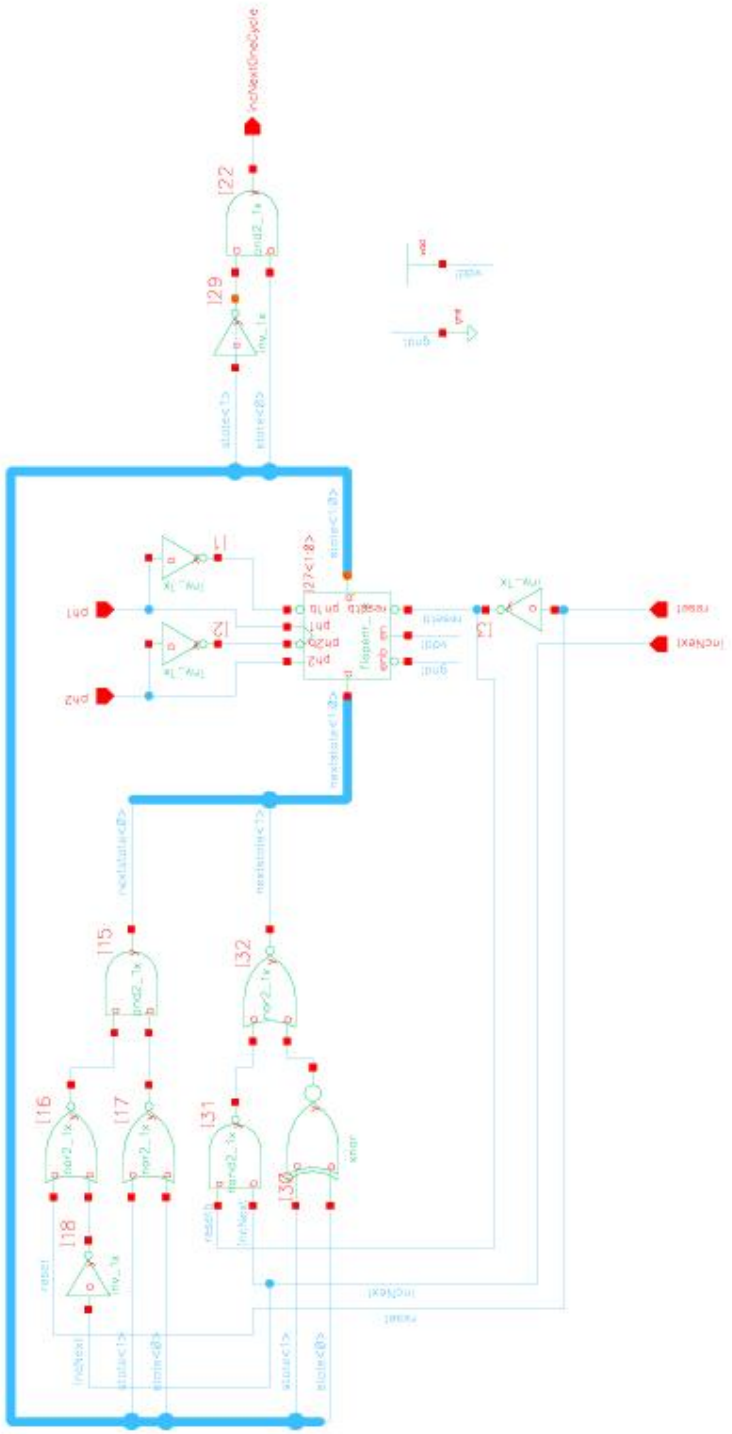

Figure 4: incNextForOneCycle Schematic

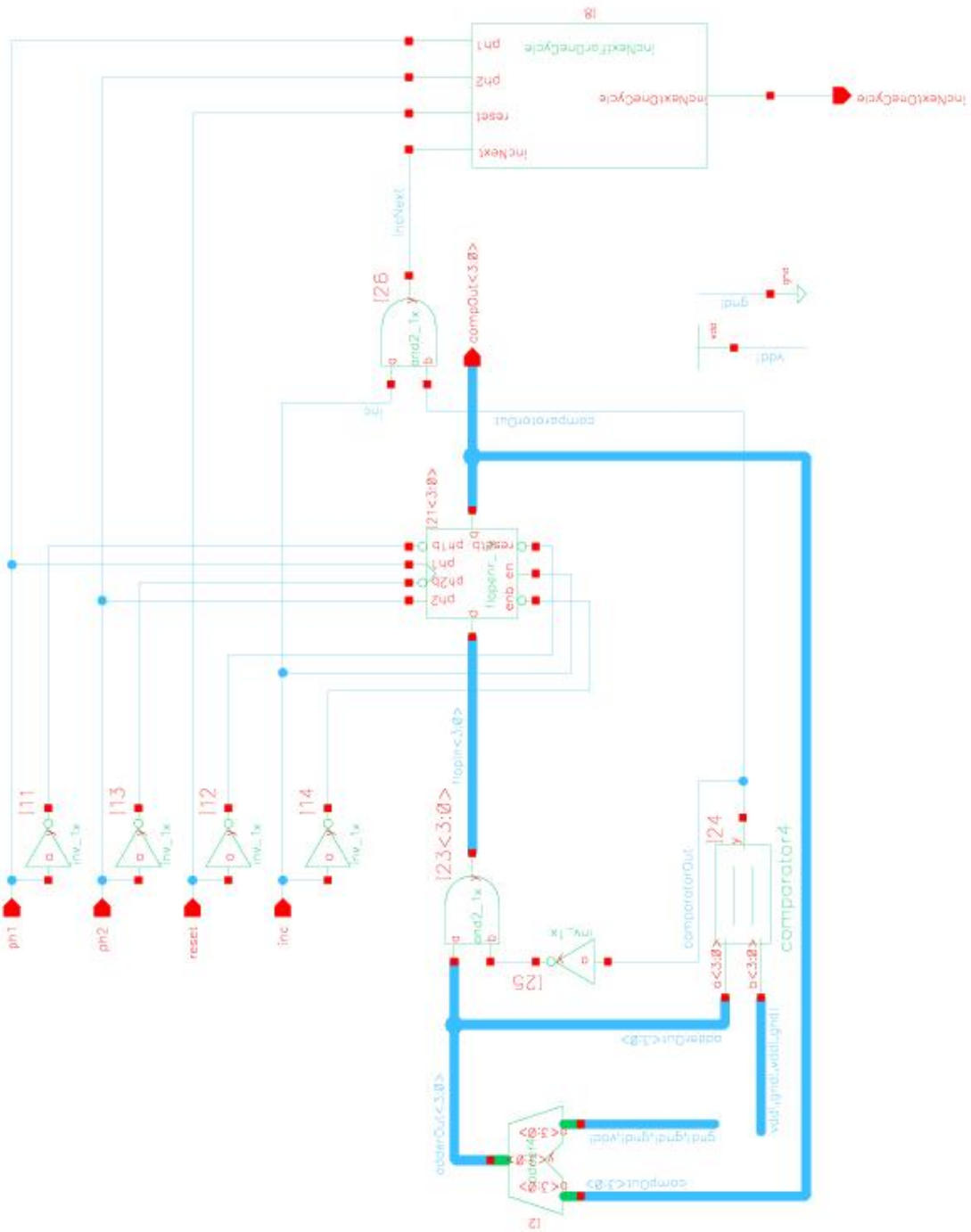

Figure 5: counterOnes Schematic

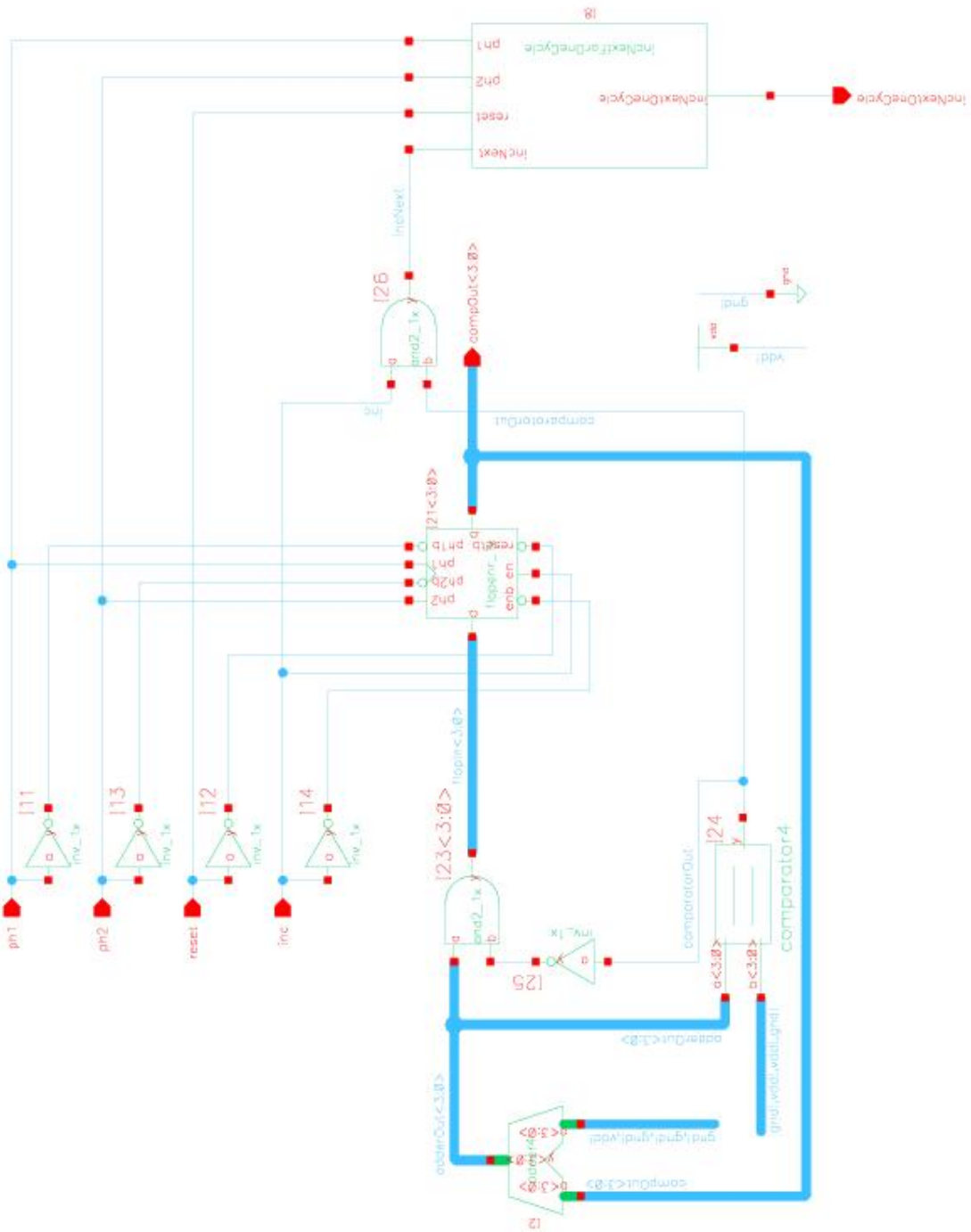

Figure 6: counterTens Schematic

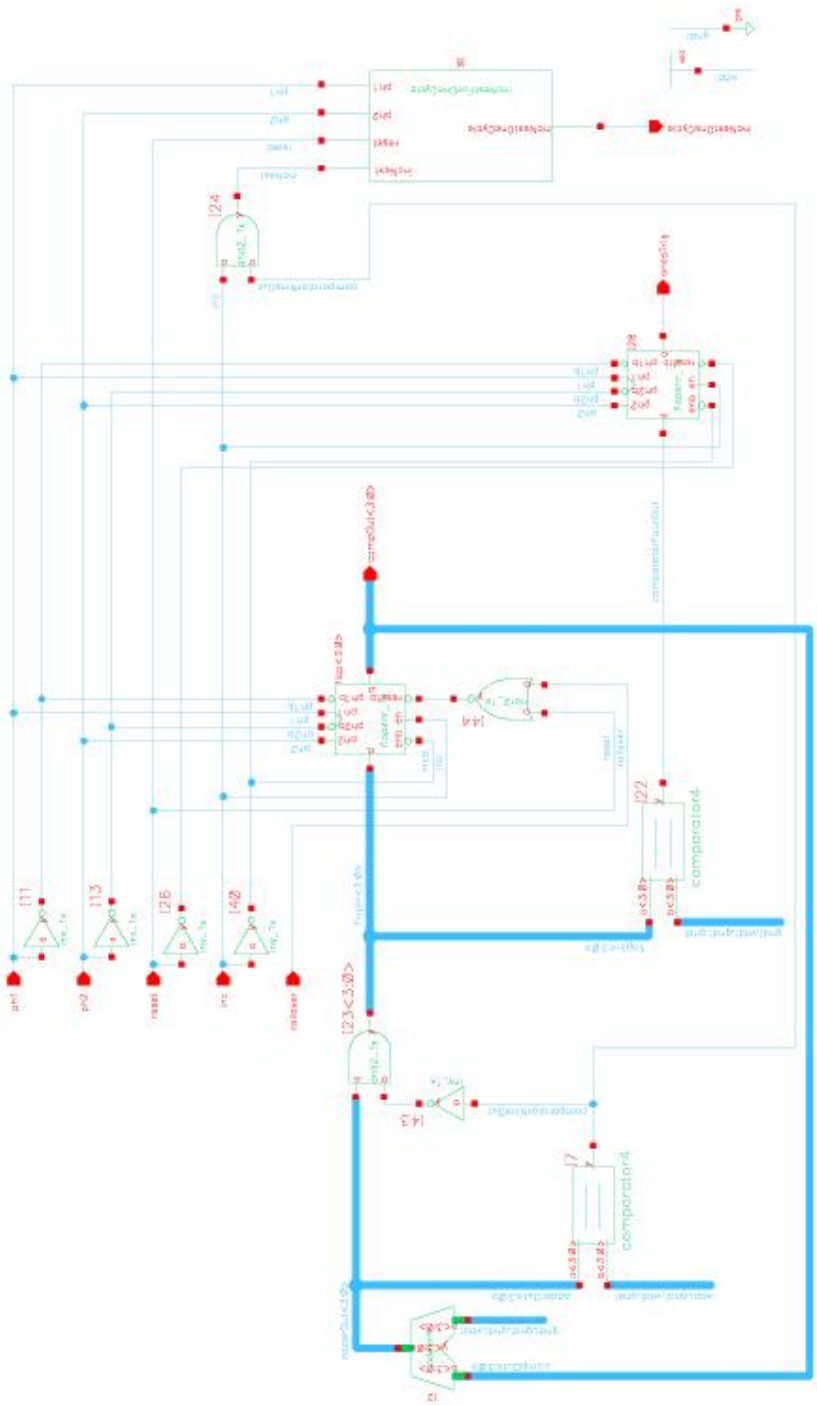

Figure 7: counterHrOnes Schematic

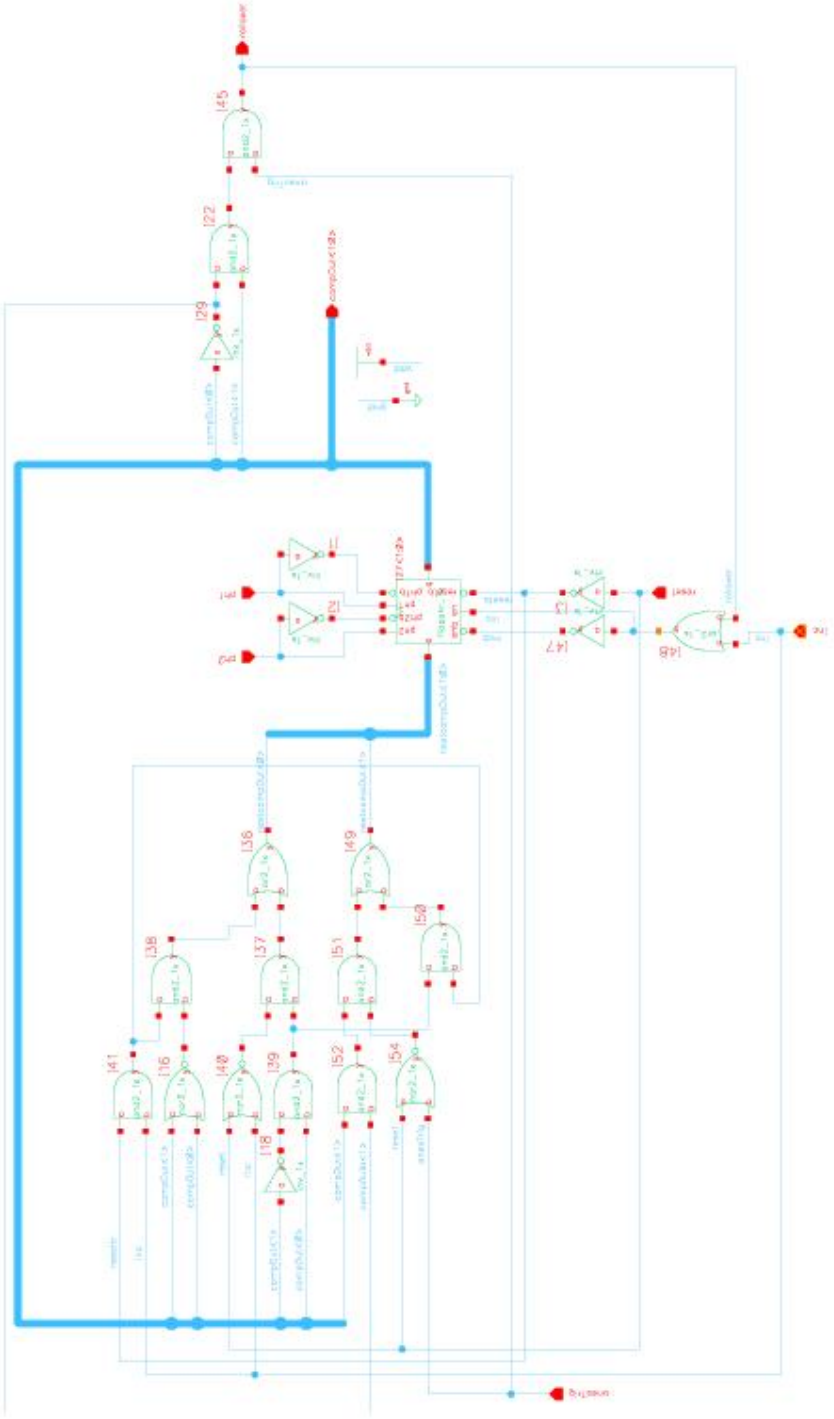

Figure 8: counterHrTens Schematic

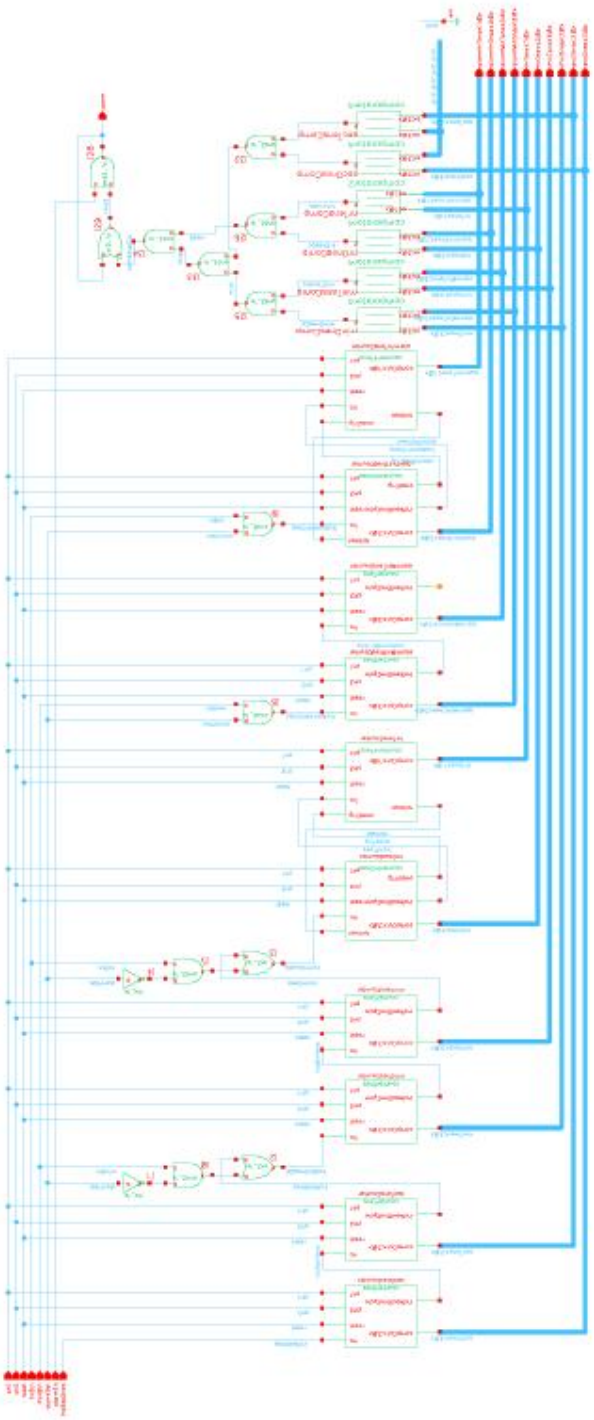

Figure 9: binary_alarm_clock datapath Schematic

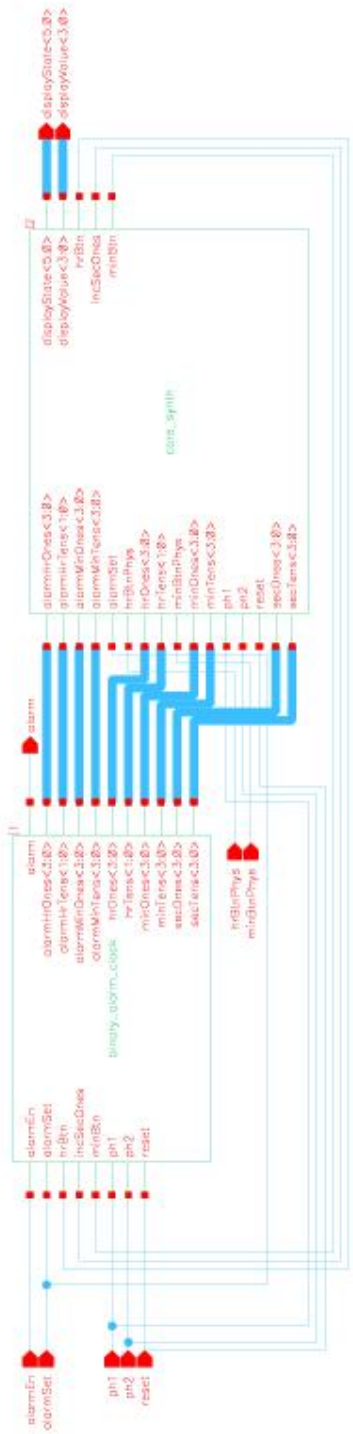

Figure 10: core Schematic

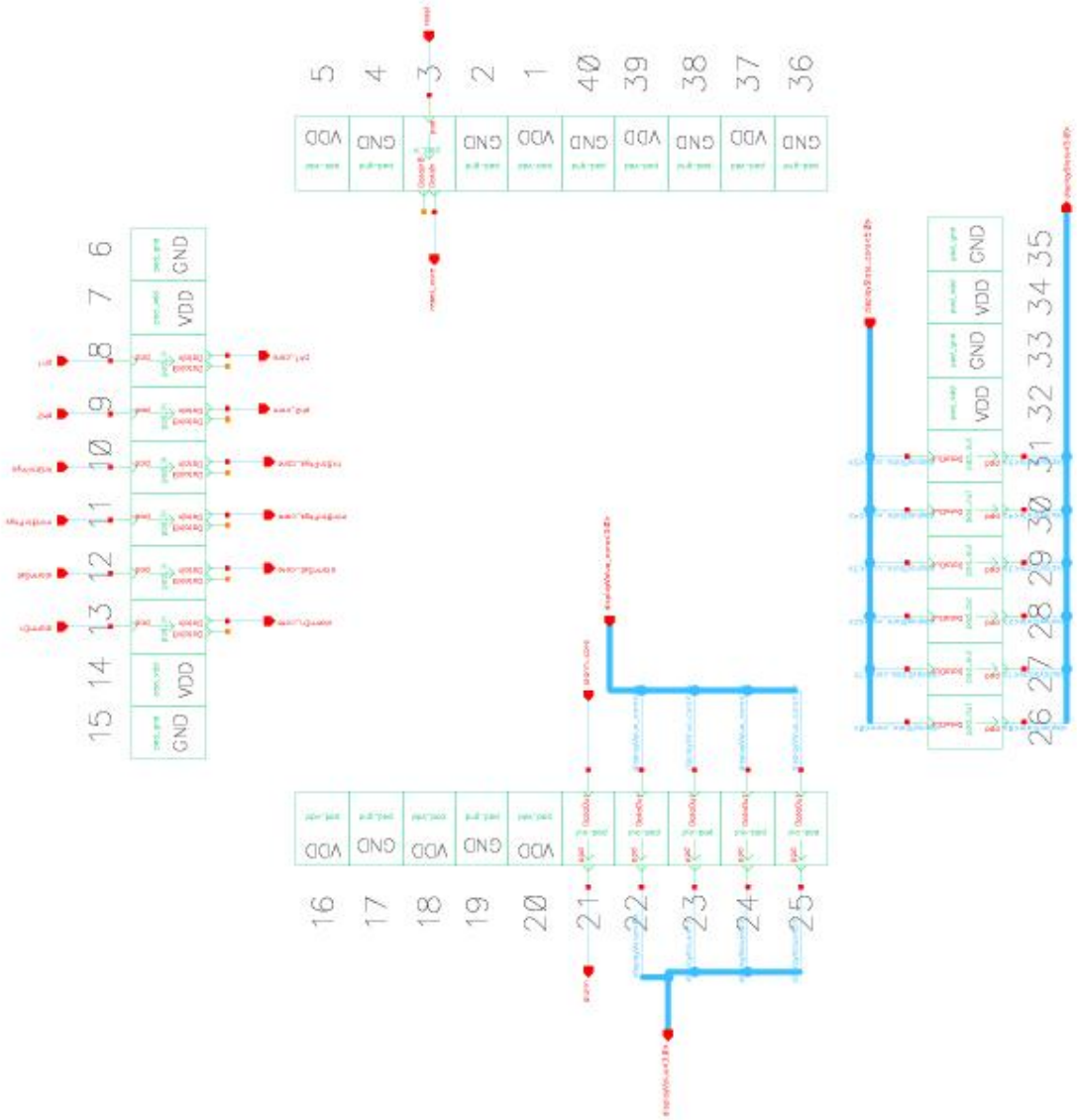

Figure 11: padframe Schematic

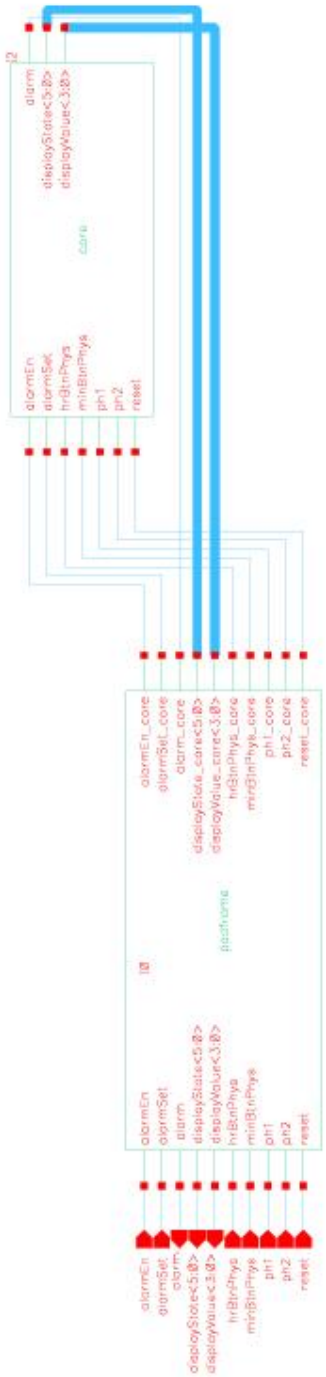

Figure 12: chip Schematic

Appendix C: Layouts

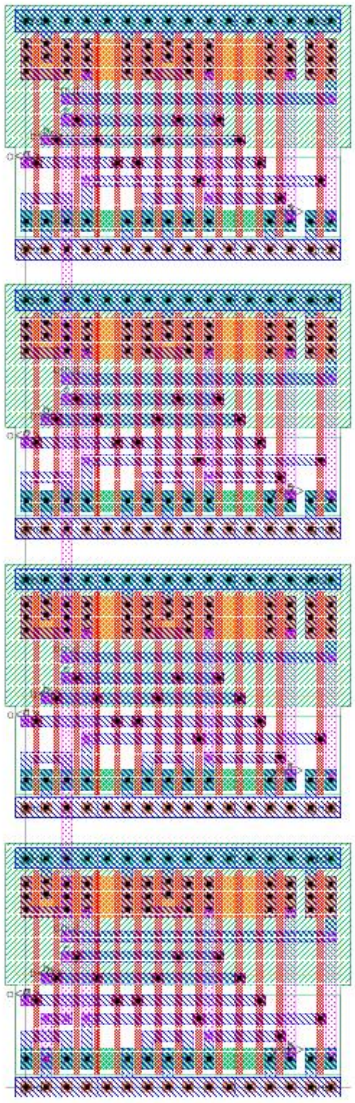

Figure 13: adder4 Layout

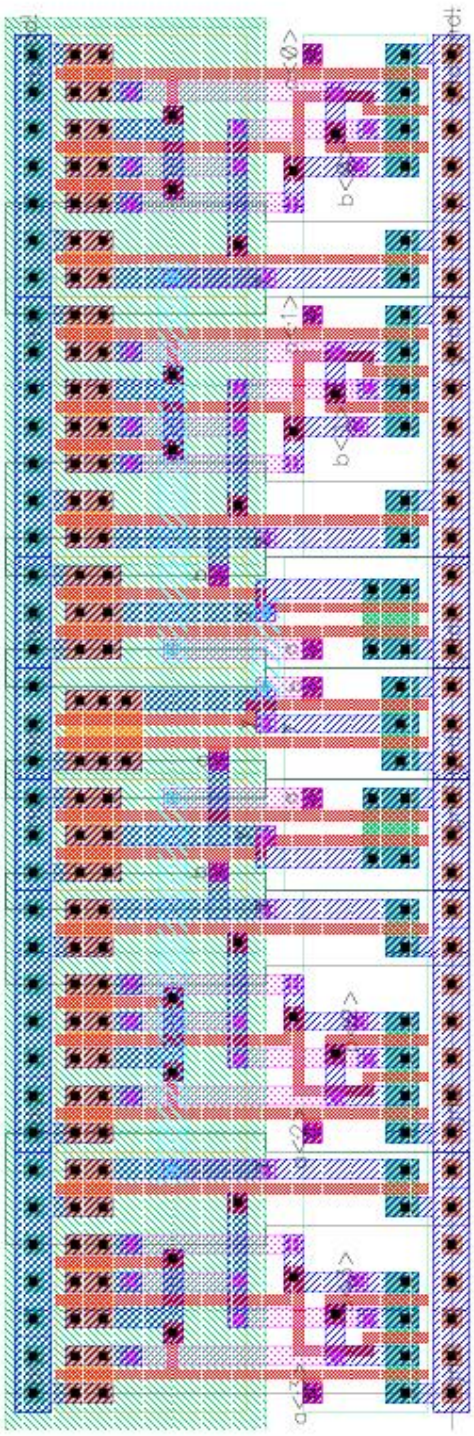

Figure 14: comparator4 Layout

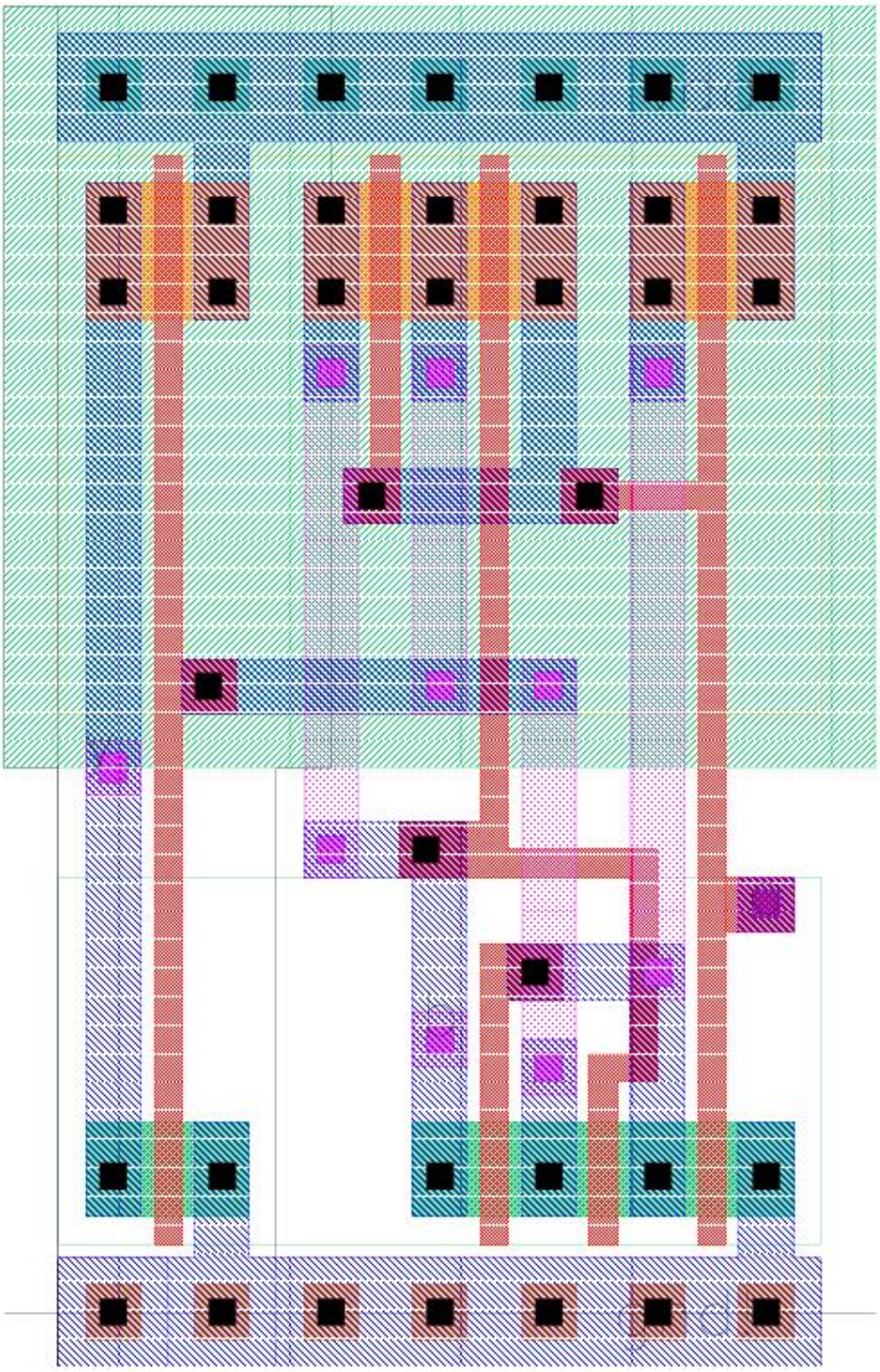

Figure 15: xnor Layout

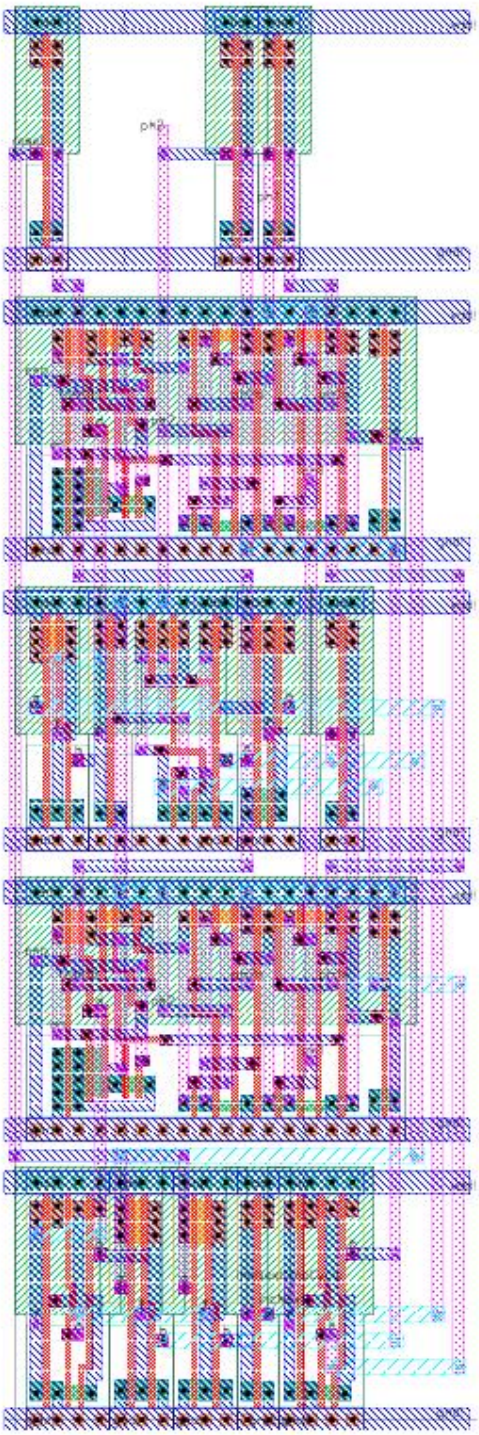

Figure 16: incNextForOneCycle Layout

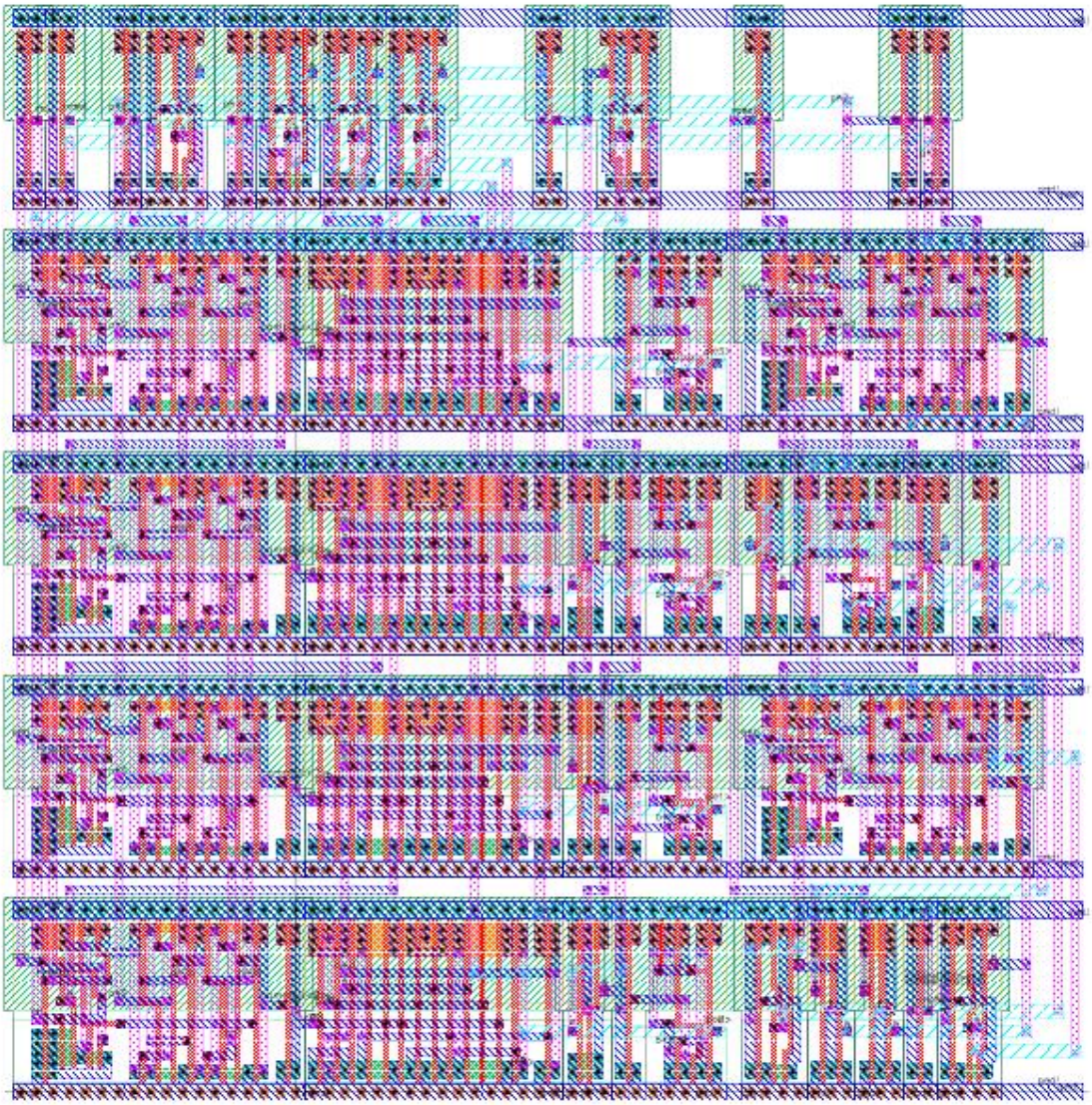

Figure 17: counterOnes Layout

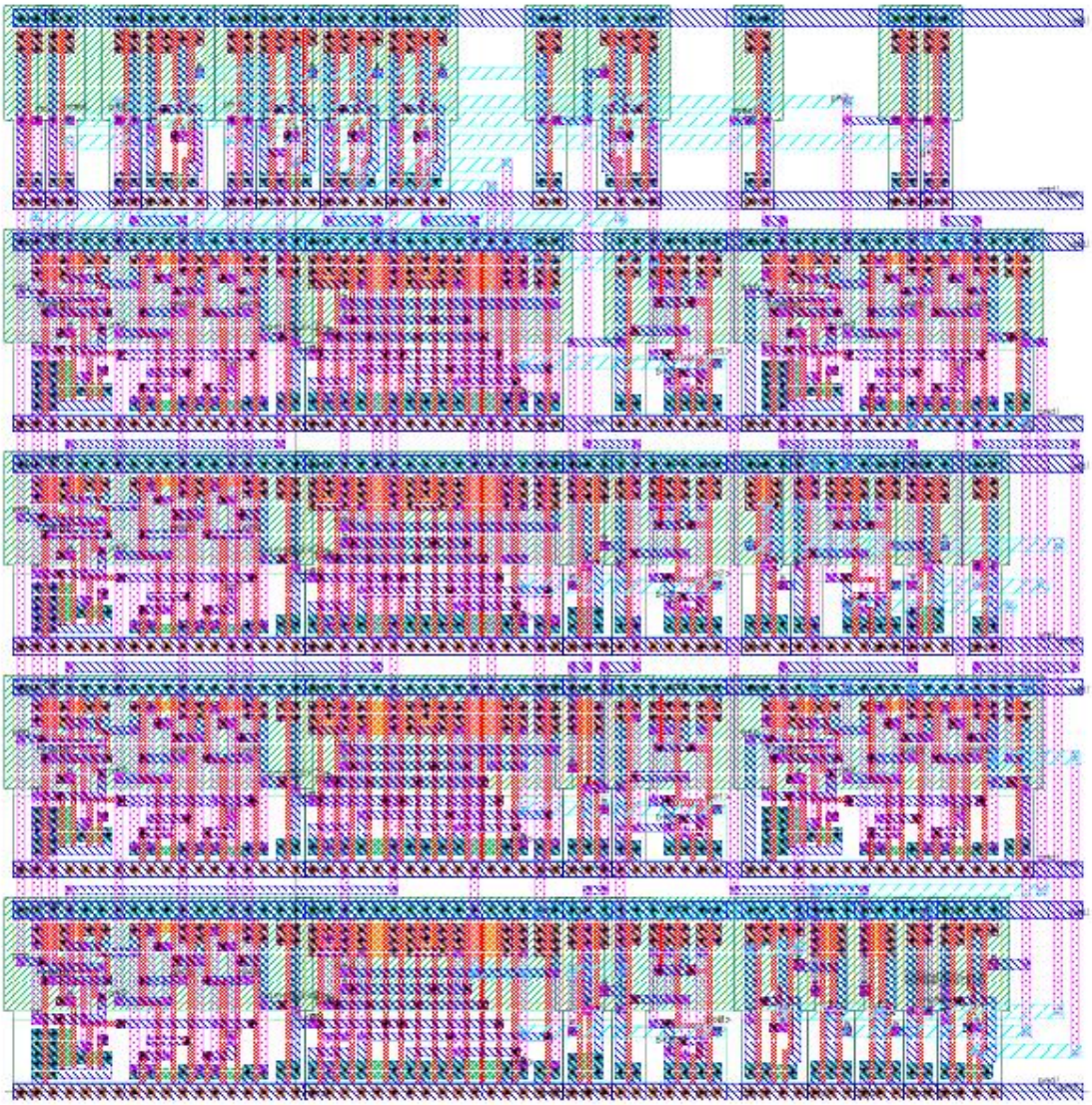

Figure 18: counterTens Layout

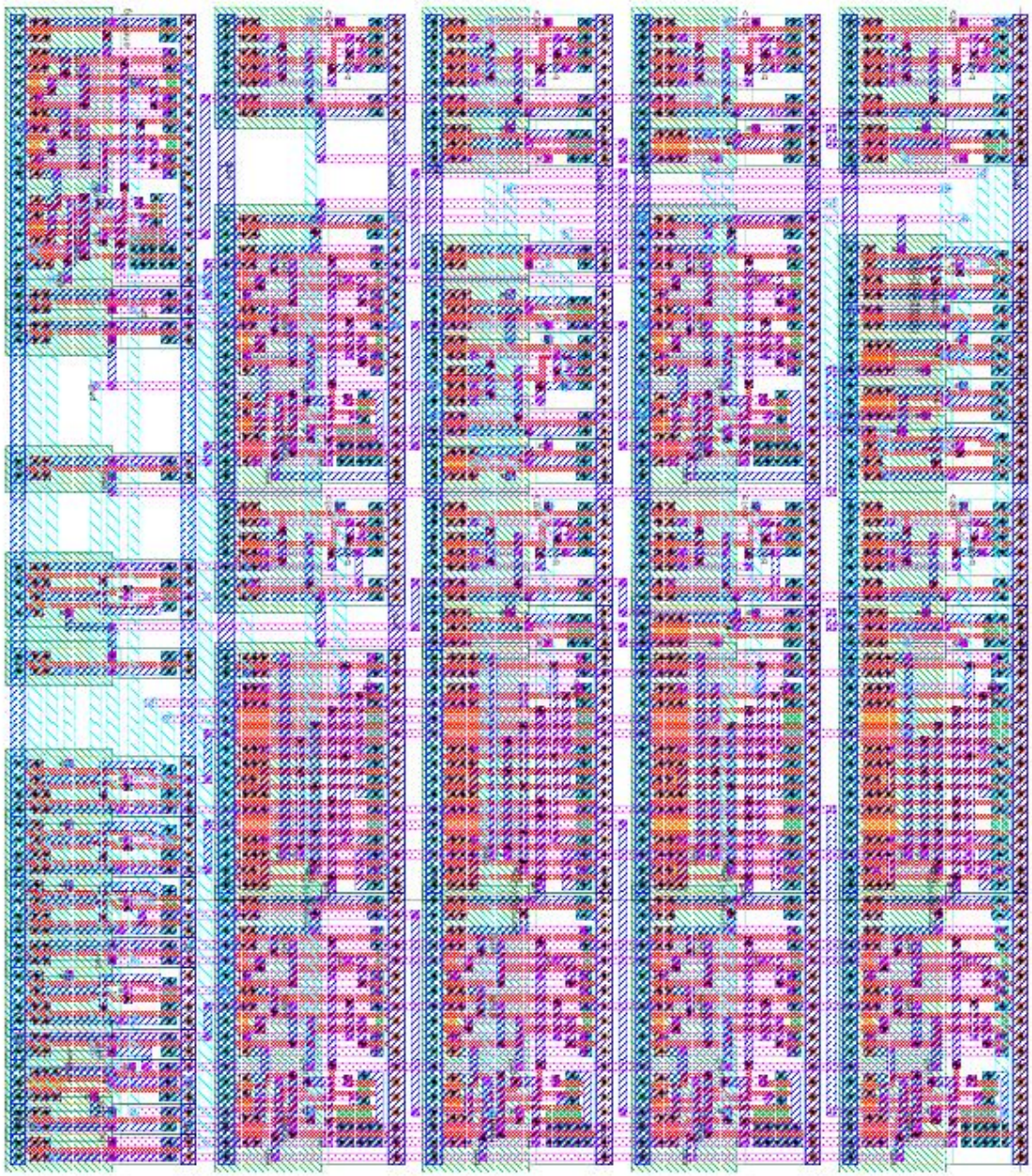

Figure 19: counterHrOnes Layout

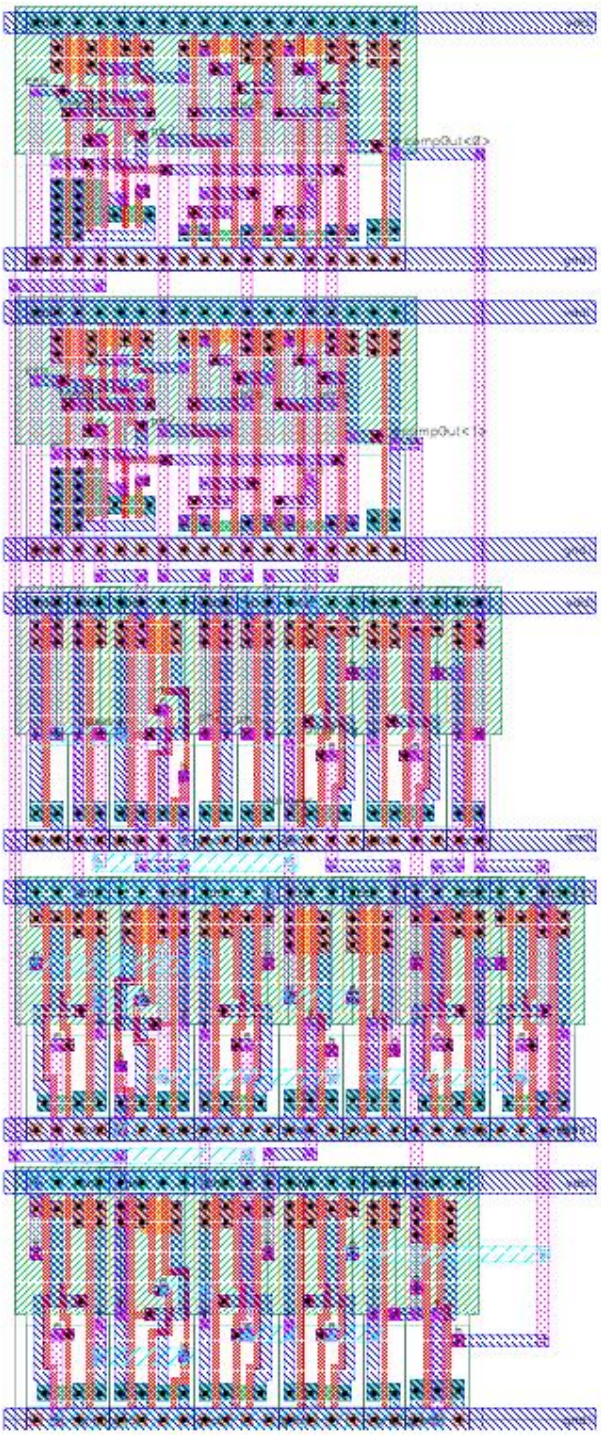

Figure 20: counterHrTens Layout

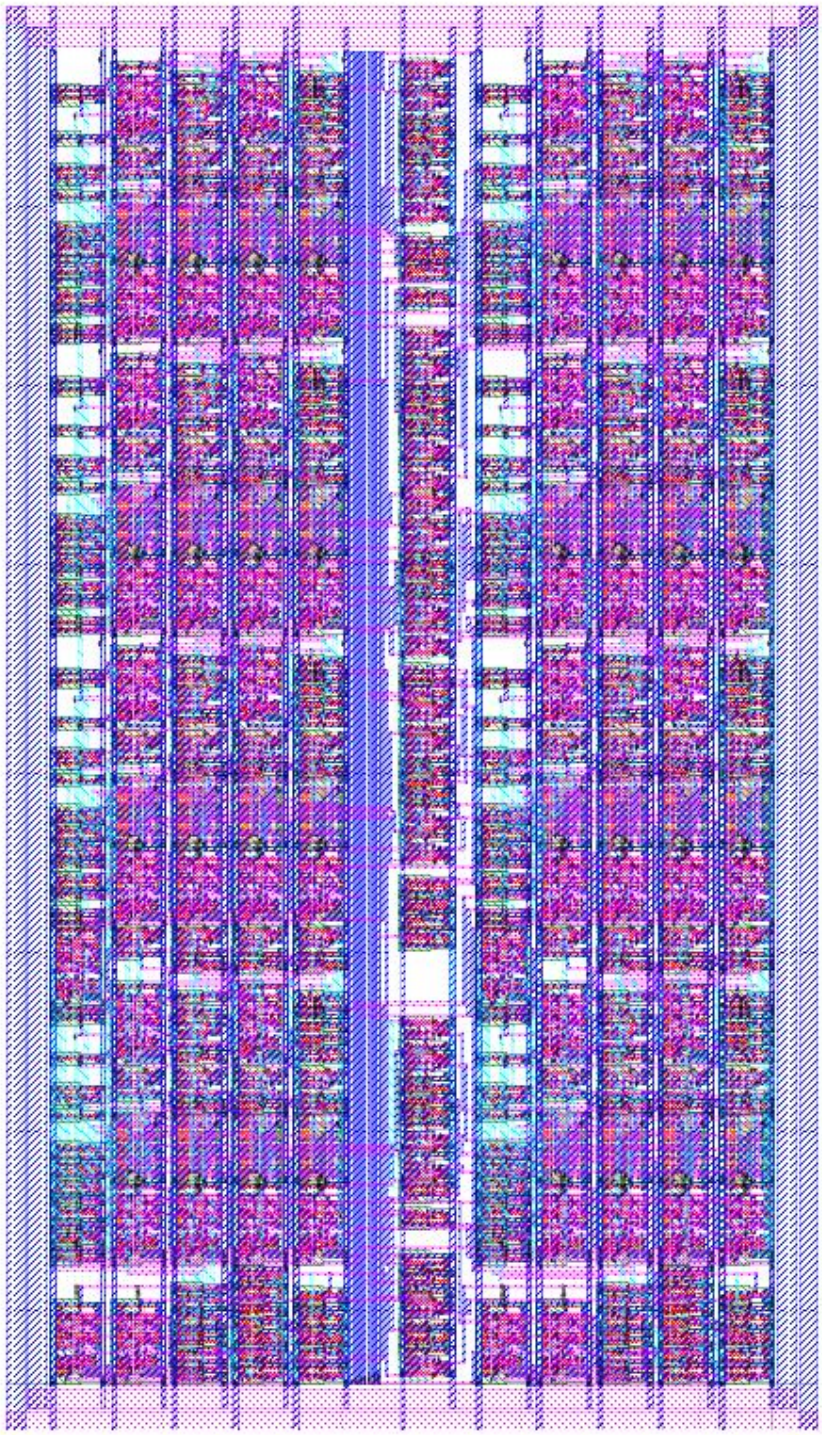

Figure 21: binary_alarm_clock datapath Layout

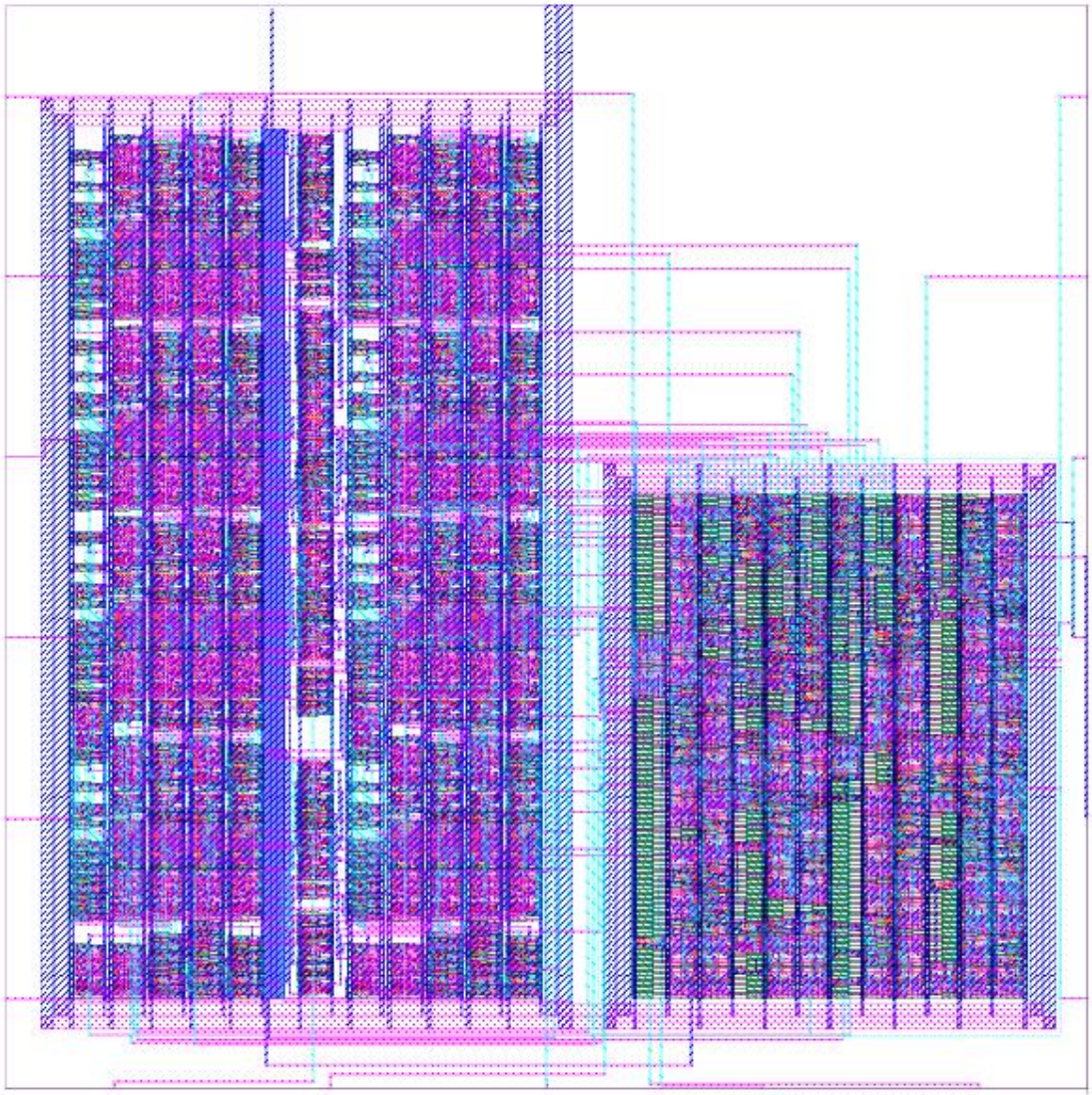

Figure 22: core Layout

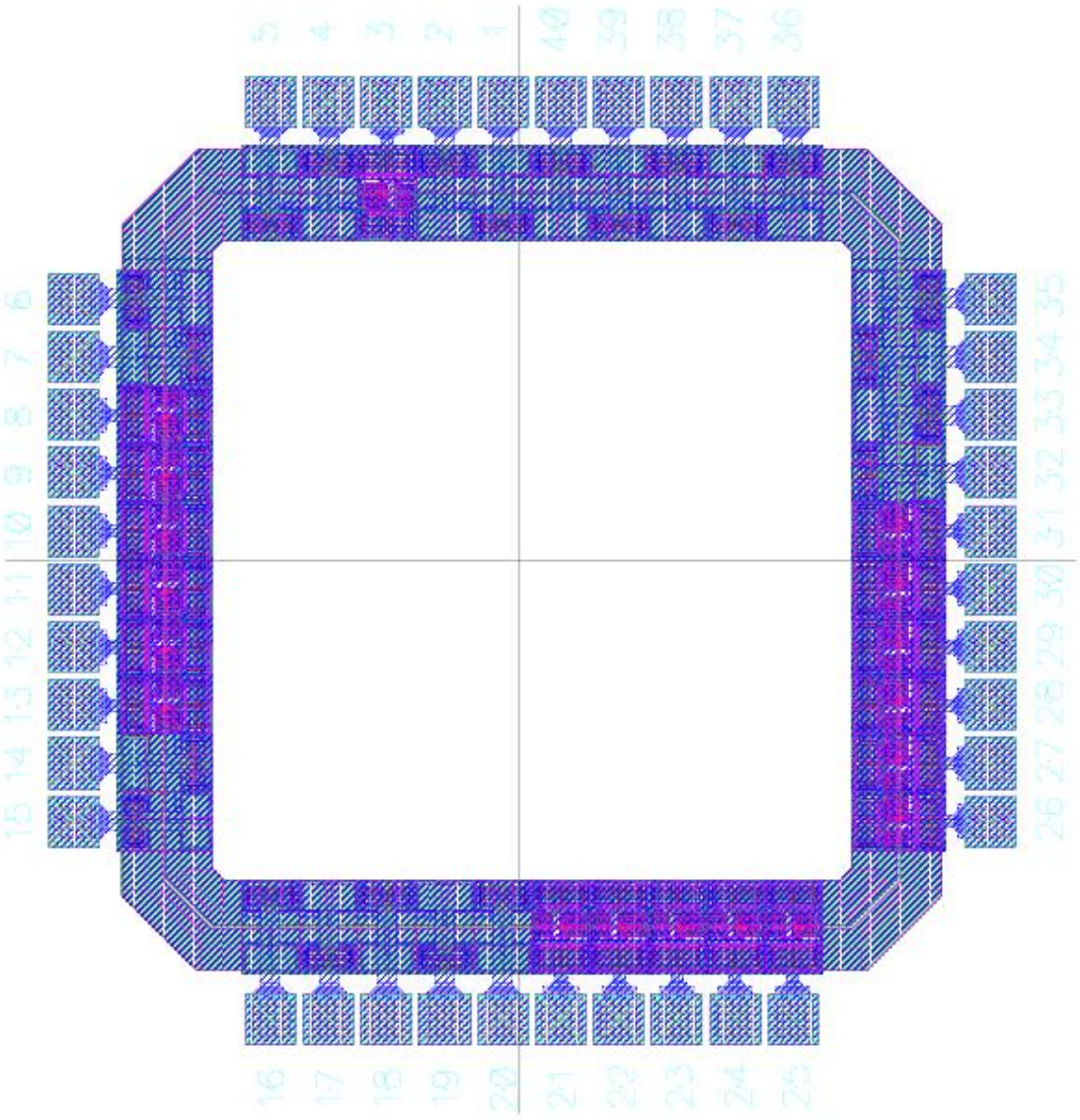

Figure 23: padframe Layout

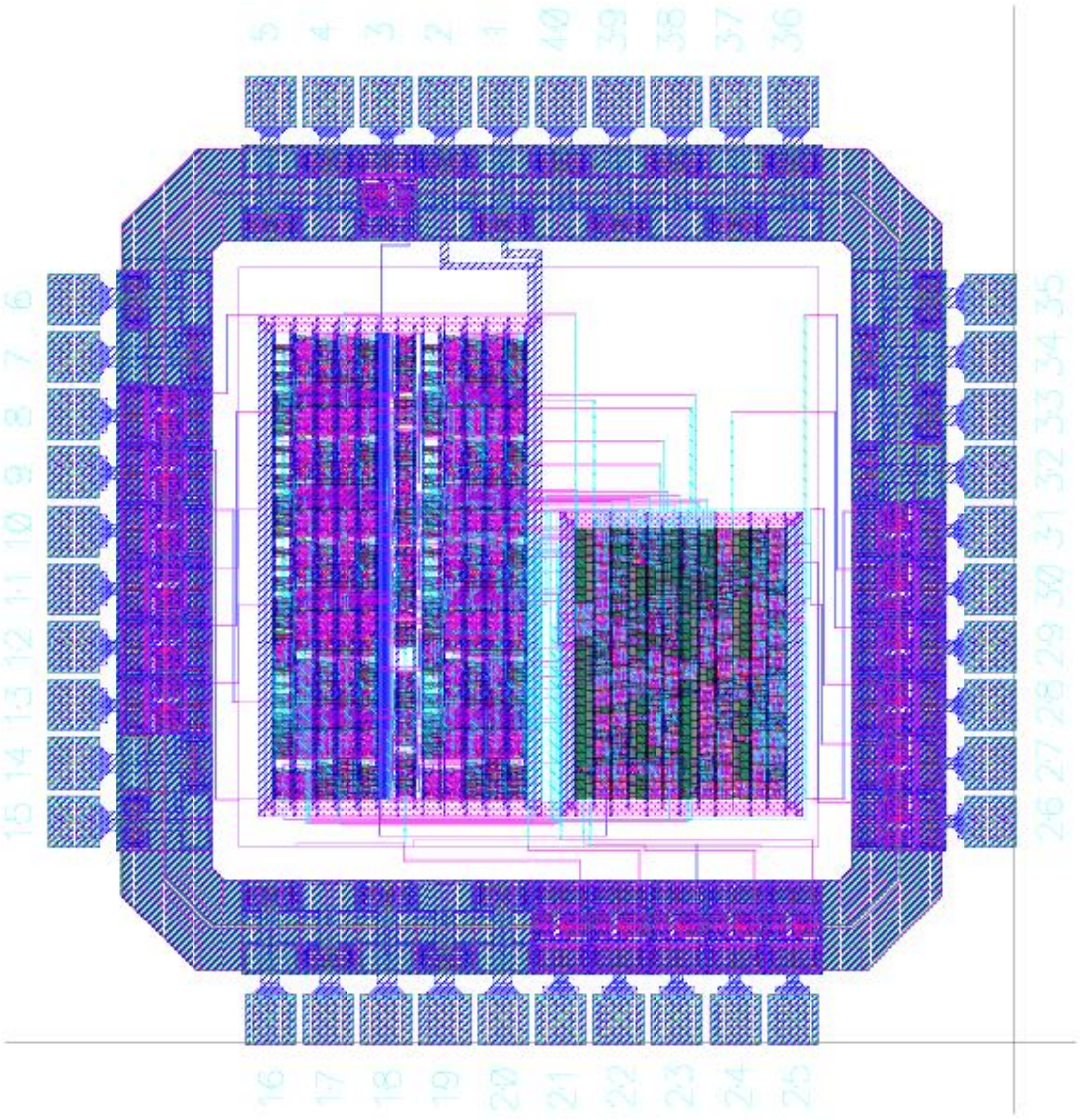

Figure 24: chip Layout