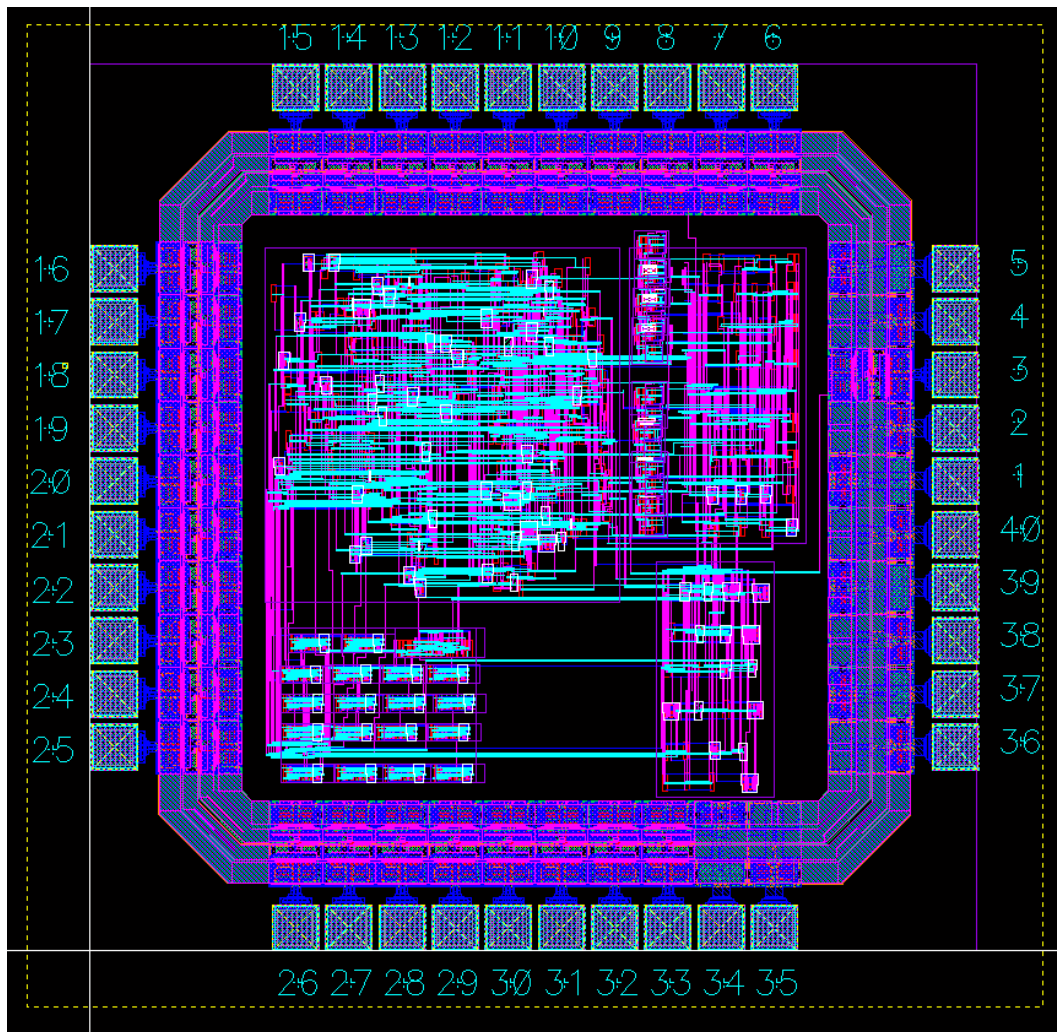


Dept. of Computer Science and Engineering

EECS 4612: Digital VLSI

Final Project: 1-Player Tic-Tac-Toe



Submitted by:

Kajanan Kanathipan 212846853

Ariel Laboriante 212951984

Instructor: Sebastian Magierowski

Date: April 5 2016

1.0 Introduction

The goal of this project is to create a 1-player game of tic-tac-toe which fits on a 1.5 X 1.5 mm 40-pin MOSIS “TinyChip” fabricated in a 0.6- μm process. The “player” will compete against the “computer” and the chip will monitor and record each turn. At the end of the game, if there is a tie between the player and the computer, the game will automatically reset. The person who starts the new game is the one who had the second-last turn. In that way, it will alternate between player and computer. If there is a winner, the game ends and returns the win/lose state of the player. The player will always play an “X” and the computer will always play a “O” on each of the nine spaces on the board.

1.1 Game Architecture

The core of the chip contains four top-level modules: Memory Board, Who Won, Computer, and Controller.

1.1.1 Memory Board

The tic-tac-toe game board is represented by 9 cells. To implement the game, 2-bits are needed to mark each cell array as either empty, a cross or a circle as shown in figure 1. A 00 will represent the empty cell, 10 will represent the “O”, and 11 will represent the “X”. The Memory Board stores the state of the game board through the use of sequential logic.

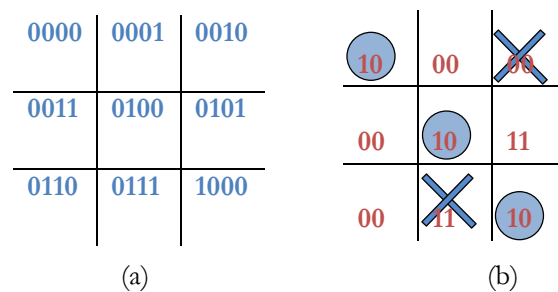


Figure 1: Each position on the board is located using 4 bits (a).
The game board is represented by 18-bits, where 2-bits represent each cell (b).

1.1.2 Who Won

This module uses combinational logic to check if there is a winner, a tie or if the game is done.

1.1.3 Computer

The Computer module monitors the player’s turn and uses combinational logic to make a move according to three principles:

- 1) Win: If a move will complete a three-in-a-row it will be done.
- 2) Don’t Lose: If a move will prevent the player from winning it will be done.
- 3) Pick the first open square: If 1) and 2) fail, it will traverse the board and pick the first open square

1.1.4 Controller

The Controller module is a finite state machine that controls the datapath which consists of the other three modules. It determines when the computer must move and when the player's inputs are stored in memory. It also determines when the game board must reset when there is a tie.

2.0 Specifications

Since the tic-tac-toe game board consists of 9 cells, and 2-bits are needed to mark it as a cross or a circle, hence the game board has a bus width of 18-bits. The LSB indicates the symbol and the MSB indicates whether the cell is empty or not.

Table 1: List of I/O pins (total of 39 pins)

Name (Function)	Description	Type	Bus Width
VDD (power)	Provides power	Input/Output	1 (x3)
GND (ground)	Ground	Input/Output	1 (x6)
reset (reset)	Reset to blank board	Input	1
ph1 (clock)	Single-phase clock	Input	1
isPlayerFirst (first turn)	Determines whether player or computer starts	Input	1
playerInput (input position)	Indicates the position of the player's move	Input	4
pcMove (computer's turn)	Determines what kind of move the computer will make; 11 to complete a 3-in-a row, 10 to prevent the player from winning, and 01 to traverse for next turn	Input	2
winner (winner of game)	Displays 11 if the player wins, 10 if the computer wins, and 01 if there is a draw	Output	2
memory (board)	Displays the current state of the game	Output	18

The operation of the chip is shown in figure 2. The controller module monitors the turns according to `isPlayerFirst`. The player's input is received from `playerInput[3:0]` and if it is the player's turn, it sends the target address `addr[3:0]` along with the player's intended cell state. The wire `cellState[1:0]` will contain 'b11 if it is the player's turn. The `memBoard` module receives the address and stores the cell state using the enable-reset flip-flops and a 4-to-8 bit decoder. Then, `memBoard` outputs the current game board state called `memory[17:0]` and sends this information to `whoWon`. The `whoWon` module then determines if the game is done, if there is a tie or if there is a winner. If it is the computer's turn, the controller sends this information to computer through `gameState[1:0]`. The computer then sends its calculated move to the controller.

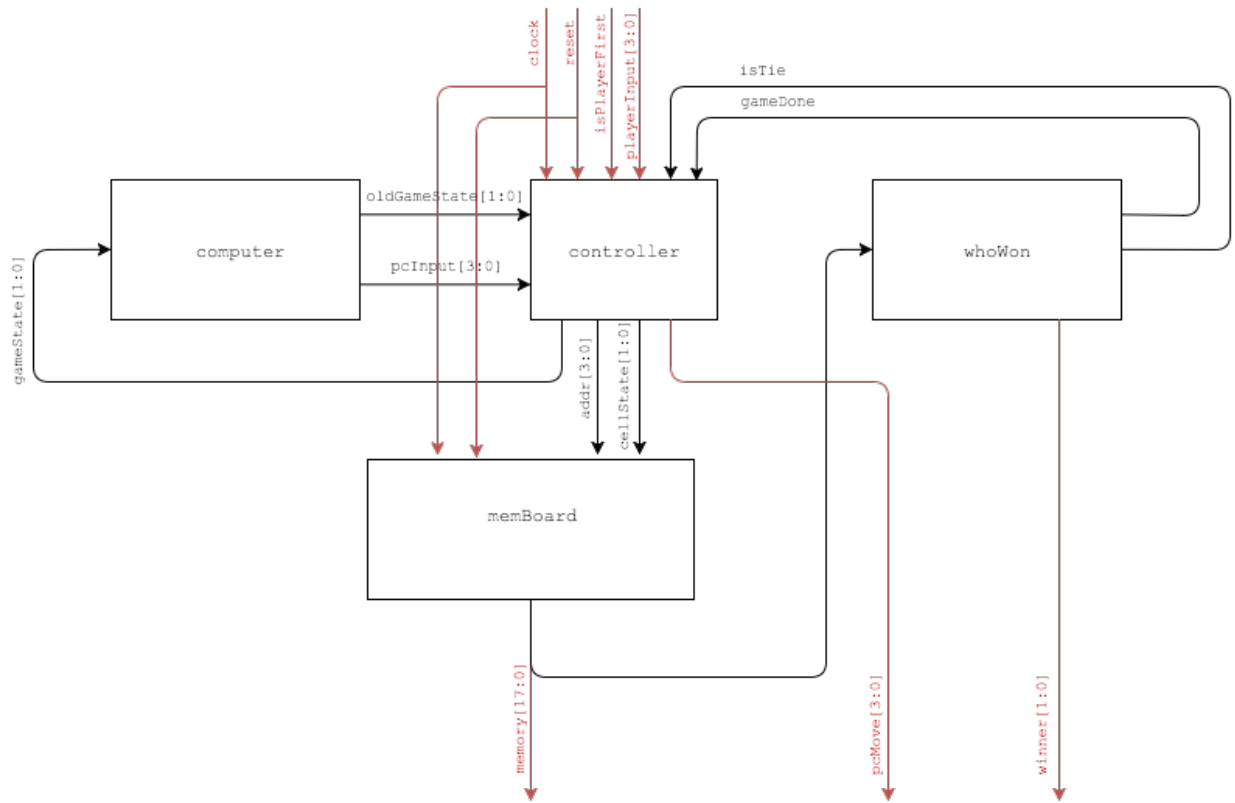


Figure 2: Block diagram of the top-level modules in the chip

3.0 Floorplan

There are four top-level blocks: memBoard, whoWon, computer, and controller. There were a few discrepancies between the actual and the proposed floorplan. The final computer module was larger than the proposed one as it had an area of $3600\lambda^2$ compared to the estimated $2000\lambda^2$. This is due to the fact that a submodule of computer named traverse used several comparisons making it larger than expected. The synthesized controller was found only $400\lambda^2$ which is smaller than what we had predicted. The final memBoard module measured 260λ by 330λ which is extremely small compared to our estimate of 256λ by 2610λ . This is because our estimate of the flipflop module was much larger than what we had expected.

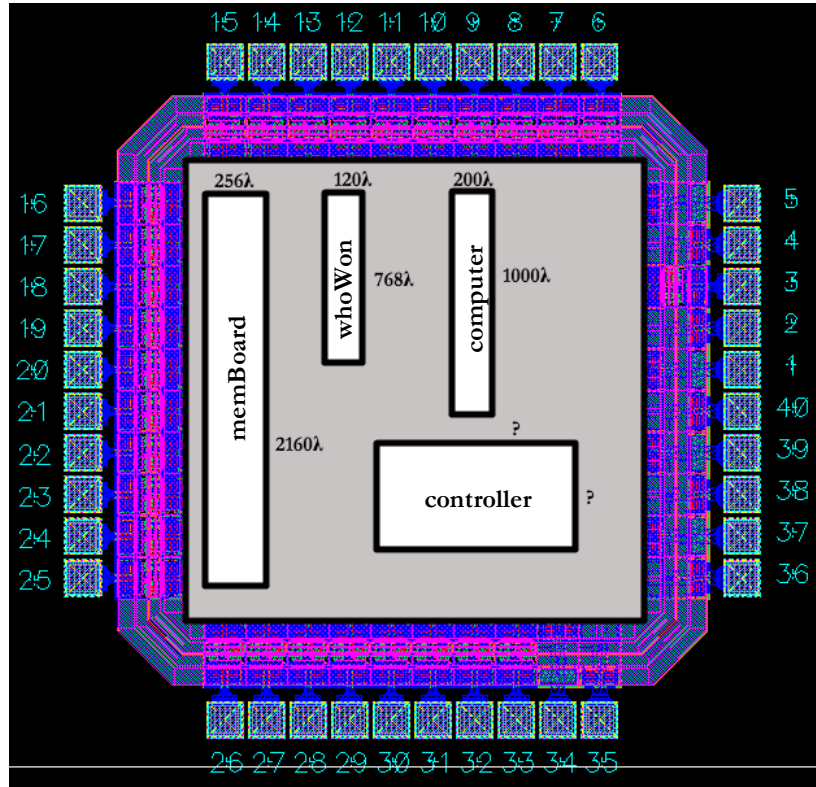


Figure 3: Tic-tac-toe chip proposal floor plan

MemBoard consists of two main parts which are the flip-flops and a logic circuit to access them. Since there are 9 addresses there will be 18 flip-flops, where 2 flip-flops are assigned to each address. The logic circuit will determine which flip-flop is being accessed and will enable them. The value assigned to the flip flop is based on whose turn it is. The flip-flop will be positioned in a 9 by 2 array format and will be taken from the muddlib10 library. Each flip-flop on its own is 144 λ by 96λ and when arranged as an array the total space taken up is 1296λ by 192 λ. The logic circuit contains 9 copies of two NOR gates connected to an AND gate assuming all inputs and their compliments are available. One copy would take up 96λ by 64λ. Assuming the copies are stacked on top of each other the total amount of space taken is 864λ by 64λ. This means the MemBoard block will take up 2160λ by 256λ.

WhoWon will check the rows, columns, and diagonals to see if either the player or computer has won. There are 8 different winning combinations that WhoWon will check for. The logic check consists of two parts. The first one checks any of the row, column, or diagonal is full and the second checks if they are all the same. The logic to check if a row, column, or diagonal is full is

$$y = abc$$

where a, b, and c are the LSB of the value stored in the memory location. Since this is just a 3 input AND gate it will be constructed using the nand3 gate and an inverter from the muddlib10 library. The logic to check if they are the same is

$$Y = \overline{ABC} + \overline{ABC}$$

where A, B and C are the MSB of the value stored in the memory location. The winner will be determined by who moved last. This will be a custom block designed using transistors from the UofU_Analog_Parts library. The dimensions of the first logic block are 96λ by 32λ . The second logic block consists of two 3 input AND gates connected to a 2 input nor gate and as a result the estimated dimensions are 96λ by 88λ . Assuming these are placed in series and that there are 8 copies of them, one for each winning combination, the total estimated dimension for the WhoWon block is 768λ by 120λ .

Computer is a block that will determine what move the computer should perform. Recall that the current plan is for the computer to follow three rules in order which are:

- 1) Win: If a move will complete a three-in-a-row it will be done.
- 2) Don't Lose: If a move will prevent the player from winning it will be done.
- 3) Pick the first open square: If 1) and 2) fail, it will traverse the board and pick the first open square

Rule 1 to 3 involves checking the memory to see what the flip flops are currently storing. As a result, we will be using a modified version of the WhoWon block. This version will check the locations that the player has accessed to determine if the player is one move away from winning. This version will also check if the computer is one move away from winning based on the current state of the flip-flop memory array. This means that the size of the block will be slightly larger than the WhoWon block. From this it can be estimated that the size of the Computer block will be about 1000λ by 200λ .

Controller is a top-level block which controls various aspects of the game which includes: whose turn it is, what the players input is, and accessing the memory array. This block is more complex than the other 4 top-level blocks and as a result it has been decided that the block will be synthesized rather than designed by hand. This is also because the block is similar to that of the controller module from Lab 3. Since it will be synthesized the dimensions are not known but it can be estimated that it will fit within the remaining amount of space.

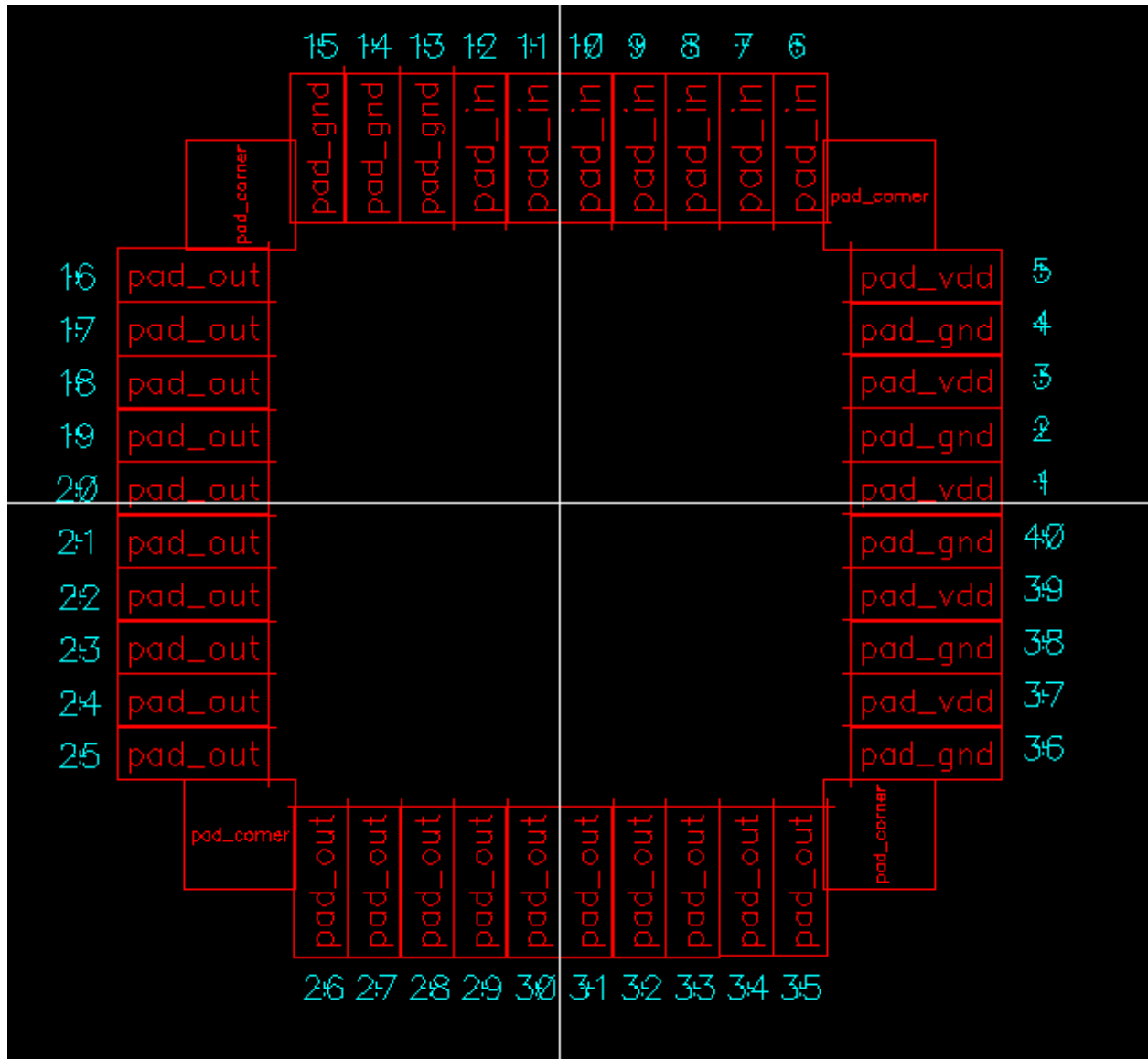


Figure 4: Padframe pinout diagram

Table 2: List of I/O pins and pin numbers

Pin number	Pin	Pin number	Pin
1	VDD	21	memory[3]
2	GND	22	memory[4]
3	VDD	23	memory[5]
4	GND	24	memory[6]
5	VDD	25	memory[7]
6	Reset	26	memory[8]
7	Ph1	27	memory[9]
8	isPlayerFirst	28	memory[10]
9	playerInput[0]	29	memory[11]
10	playerInput[1]	30	memory[12]

11	playerInput[2]	31	memory[13]
12	playerInput[3]	32	memory[14]
13	GND	33	memory[15]
14	GND	34	memory[16]
15	GND	35	memory[17]
16	winner[0]	36	GND
17	winner[1]	37	VDD
18	memory[0]	38	GND
19	memory[1]	39	VDD
20	memory[2]	40	GND

4.0 Verification

5.0 Post-fabrication Test Plan

If the chip is fabricated, it can be tested by the following set of tests:

- 1) Inspect the function of the chip using an FPGA board.
- 2) Test the minimum VDD by reducing the voltage until the chip fails the test.
- 3) Test the chip's minimum frequency by reducing the clock cycle until the chip fails the test.

6.0 Design Time

The summary of design time spent on each component of the project is shown in table 2:

Table 3: Design time spent on each component

Component	Hours Spent
Proposal	10
Verilog	30
Schematic	10
Layout	30
Simulations	30
Final Report	20
Total	130

7.0 File Locations

The files for the project can be located using table 3:

Table 4: Location of project files

Files	Location
Verilog code	
Test vectors	
Synthesis results	
Cadence libraries	

HSPICE simulations (if any)	
CIF	
PDF chip plot	
PDF of final report	

8.0 References

Weste, Neil H. E., and David Money. Harris. CMOS VLSI Design: A Circuits and Systems Perspective. Boston: Addison Wesley, 2011. Web.

9.0 Appendices

9.1 Verilog Code

9.1.1 isFull Verilog Code

```
module testbench();
    logic a, b, c, y;
    isFull dut(a, b, c, y);
    `include "testfixture.verilog"

endmodule

module isFull (input  logic a,
               input  logic b,
               input  logic c,
               output logic y)

    assign #1 y = (a & b & c);
endmodule
```

9.1.2 isSame Verilog Code

```
module testbench();
    logic a, b, c, y;
    isSame dut(a, b, c, y);
    `include "testfixture.verilog"

endmodule

module isSame (input  logic a,
               input  logic b,
               input  logic c,
```

```

        output logic y);

    assign #1 y = (a & b & c) + (~a & ~b & ~c);

endmodule

```

9.1.3 isTwoSame Verilog Code

```

module testbench();
    logic [5:0] in;
    logic [2:0] y;
    isTwoSame dut(in, y);
    `include "testfixture.verilog"

endmodule

module isTwoSame (input logic [5:0] in,
                  output logic [2:0] y);
always @ (in)
begin
    case (in)
        6'b001010: y = 3'b011;
        6'b001111: y = 3'b011;
        6'b100010: y = 3'b101;
        6'b110011: y = 3'b101;
        6'b101000: y = 3'b111;
        6'b111100: y = 3'b111;

        default: y = 3'b000;

    endcase
end

endmodule

```

9.1.4 address Verilog Code

```

module testbench();
    logic [3:0] addr;
    logic [8:0] writePos;
    address dut(addr, writePos);

    `include "testfixture.verilog"

endmodule

```

```

module address (input  logic [3:0] addr,
                output  logic [8:0] writePos);

    assign #1 writePos[0] = ~(addr[0] | addr[1]) & ~(addr[2] | addr[3]);
    assign #1 writePos[1] = (~(~addr[0] | addr[1]) & ~(addr[2] | addr[3]));
    assign #1 writePos[2] = ~(addr[0] | ~addr[1]) & ~(addr[2] | addr[3]);
    assign #1 writePos[3] = (~(~addr[0] | ~addr[1]) & ~(addr[2] | addr[3]));
    assign #1 writePos[4] = ~(addr[0] | addr[1]) & ~(~addr[2] | addr[3]);
    assign #1 writePos[5] = (~(~addr[0] | addr[1]) & ~(~addr[2] | addr[3]));
    assign #1 writePos[6] = ~(addr[0] | ~addr[1]) & ~(~addr[2] | addr[3]);
    assign #1 writePos[7] = (~(~addr[0] | ~addr[1]) & ~(~addr[2] | addr[3]));
    assign #1 writePos[8] = ~(addr[0] | addr[1]) & ~(addr[2] | ~addr[3]);

endmodule

```

9.1.5 traverse Verilog Code

```

module testbench();
    logic [17:0] memory;
    logic [3:0] y;
    traverse dut(memory,y);
    `include "testfixture.verilog"

endmodule

module traverse (input  logic [17:0] memory,
                output logic [3:0] y);

    reg [3:0] x;

    always @ (memory[17:0])
    begin

        if (memory[0] == 0)
        begin
            x[0] = 0;
            x[1] = 0;
            x[2] = 0;
            x[3] = 0;
        end

        else if(memory[2] == 0)
        begin

            x[0] = 1;
            x[1] = 0;
            x[2] = 0;
            x[3] = 0;

```

```

end

    else if(memory[4] == 0)
begin

    x[0] = 0;
    x[1] = 1;
    x[2] = 0;
    x[3] = 0;
end

    else if(memory[6] == 0)
begin

    x[0] = 1;
    x[1] = 1;
    x[2] = 0;
    x[3] = 0;
end

    else if(memory[8] == 0)
begin

    x[0] = 0;
    x[1] = 0;
    x[2] = 1;
    x[3] = 0;
end

    else if(memory[10] == 0)
begin

    x[0] = 1;
    x[1] = 0;
    x[2] = 1;
    x[3] = 0;
end

    else if(memory[12] == 0)
begin

    x[0] = 0;
    x[1] = 1;
    x[2] = 1;
    x[3] = 0;
end

    else if(memory[14] == 0)
begin

```

```

        x[0] = 1;
        x[1] = 1;
        x[2] = 1;
        x[3] = 0;
    end

    else if(memory[16] == 0)
    begin

        x[0] = 0;
        x[1] = 0;
        x[2] = 0;
        x[3] = 1;
    end

    else
    begin

        x[0] = 1;
        x[1] = 1;
        x[2] = 1;
        x[3] = 1;
    end

end

end

assign #10 y[0] = x[0];
assign #10 y[1] = x[1];
assign #10 y[2] = x[2];
assign #10 y[3] = x[3];

endmodule

```

9.1.6 computer Verilog Code

```

module testbench();
    logic [17:0] memory;
    logic [3:0] pcInput;
    logic [1:0] gamestate;
    logic [1:0] oldGameState;
    computer dut(memory, gamestate, pcInput, oldGameState);
    `include "testfixture.verilog"

endmodule

```

```

module computer (input  logic [17:0] memory,
                 input  logic [1:0] gamestate,
                 output logic [3:0] pcInput,
                 output logic [1:0] oldGameState);

logic [2:0] row1, row2, row3;
logic [2:0] col1, col2, col3;
logic [2:0] dia1, dia2;
logic [3:0] x;

    isTwoSame row1(memory[0], memory[1], memory[2], memory[3], memory[4],
memory[5], row1);
    isTwoSame row2(memory[6], memory[7], memory[8], memory[9], memory[10],
memory[11], row2);
    isTwoSame row3(memory[12], memory[13], memory[14], memory[15],
memory[16], memory[17], row3);
    isTwoSame ccol1(memory[0], memory[1], memory[6], memory[7], memory[12],
memory[13], col1);
    isTwoSame ccol2(memory[2], memory[3], memory[8], memory[9], memory[14],
memory[15], col2);
    isTwoSame ccol3(memory[4], memory[5], memory[10], memory[11],
memory[16], memory[17], col3);
    isTwoSame ddia1(memory[0], memory[1], memory[8], memory[9], memory[16],
memory[17], dia1);
    isTwoSame ddia2(memory[4], memory[5], memory[8], memory[9], memory[12],
memory[13], dia2);

    traverse traversetest (memory, x);

always_comb
begin

    if ( ~gamestate[0] & gamestate[1] )
    begin

        if (row1[0] == 1 )
        begin
            case (row1[2:1])
                2'b01: pcInput = 4'b0000;
                2'b10: pcInput = 4'b0001;
                2'b11: pcInput = 4'b0010;
                default: pcInput = 4'b1111;

            endcase
        end

        else if (row2[0] == 1 )
        begin

```

```

        case (row2[2:1])
            3'b01: pcInput = 4'b0011;
            3'b10: pcInput = 4'b0100;
            3'b11: pcInput = 4'b0101;
            default: pcInput = 4'b1111;

        endcase
    end

    else if (row3[0] == 1 )
    begin
        case (row3[2:1])
            2'b01: pcInput = 4'b0110;
            2'b10: pcInput = 4'b0111;
            2'b11: pcInput = 4'b1000;
            default: pcInput = 4'b1111;

        endcase
    end

    else if (col1[0] == 1 )
    begin
        case (col1[2:1])
            2'b01: pcInput = 4'b0000;
            2'b10: pcInput = 4'b0011;
            2'b11: pcInput = 4'b0110;
            default: pcInput = 4'b1111;

        endcase
    end

    else if (col2[0] == 1 )
    begin
        case (col2[2:1])
            2'b01: pcInput = 4'b0001;
            2'b10: pcInput = 4'b0100;
            2'b11: pcInput = 4'b0111;
            default: pcInput = 4'b1111;

        endcase
    end

    else if (col3[0] == 1 )
    begin
        case (col3[2:1])
            2'b01: pcInput = 4'b0010;
            2'b10: pcInput = 4'b0101;
            2'b11: pcInput = 4'b1000;
            default: pcInput = 4'b1111;

```

```

        endcase
    end

    else if (dia1[0] == 1 )
    begin
        case (dia1[2:1])
            2'b01: pcInput = 4'b0000;
            2'b10: pcInput = 4'b0100;
            2'b11: pcInput = 4'b1000;
            default: pcInput = 4'b1111;

        endcase
    end

    else if (dia2[0] == 1 )
    begin
        case (dia2[2:1])
            2'b01: pcInput = 4'b0010;
            2'b10: pcInput = 4'b0100;
            2'b11: pcInput = 4'b0110;
            default: pcInput = 4'b1111;

        endcase
    end

    else
    begin

        pcInput[0] = x[0];
        pcInput[1] = x[1];
        pcInput[2] = x[2];
        pcInput[3] = x[3];

    end

end

else
begin
    pcInput[0] = 1;
    pcInput[1] = 1;
    pcInput[2] = 1;
    pcInput[3] = 1;
end

oldGameState = gamestate;

```



```

end
assign #1 pcTurnDone = 1;

endmodule

```

9.1.7 controller Verilog Code

```

module controller (input logic en,
                   input  logic reset,
                   input  logic isPlayerFirst,
                   input  logic isTie,
                   input  logic [3:0] playerInput,
                   input  logic [3:0] pcInput,
                   input  logic gameDone,
                   input  logic [1:0] oldGameState,
                   output logic [3:0] addr,
                   output logic [1:0] cellState,
                   output logic [1:0] gameState);

always_comb
begin

    if(en)
begin

        if (reset == 1'b1 | isTie == 1'b1)
begin

            addr = 15;
            cellState = 0;
            gameState= 0;

        end

        else if (gameDone == 1 )
begin

            addr = 15;
            cellState = 0;
            gameState= 0;

        end

        else if (oldGameState== 2)
begin

```

```

        addr = pcInput;
        cellState = 1;
        gameState= 1;

    end

    else if(isPlayerFirst == 1 | oldGameState== 1) //pcTurnDone == 1)
    begin

        addr = playerInput;
        cellState = 2;
        gameState= 2;

    end

    else
    begin

        addr = 15;
        cellState = 0;
        gameState= 0;

    end

end
end
end
endmodule

```

9.1.8 memBoard Verilog Code

```

module testbench();
    logic [3:0] addr;
    logic [8:0] addrflip;
    logic ph1, en, reset;
    logic d, q;
    logic [1:0] cellState;
    logic [17:0] memory;

    memBoard dut(reset, addr, cellState, ph1, memory);

    `include "testfixture.verilog"

endmodule

```

```

module memBoard (
    input  logic reset,
    input  logic [3:0] addr,
    input  logic [1:0] cellState,
    input          logic ph1,
    output logic [17:0] memory);

    logic [8:0] addrflip;
    logic [17:0] holder;

    address add(addr,addrflip);

    flopenr cell00(ph1, reset, addrflip[0], cellState[0], holder[0]);
    flopenr cell01(ph1, reset, addrflip[0], cellState[1], holder[1]);
    flopenr cell10(ph1, reset, addrflip[1], cellState[0], holder[2]);
    flopenr cell11(ph1, reset, addrflip[1], cellState[1], holder[3]);
    flopenr cell20(ph1, reset, addrflip[2], cellState[0], holder[4]);
    flopenr cell21(ph1, reset, addrflip[2], cellState[1], holder[5]);
    flopenr cell30(ph1, reset, addrflip[3], cellState[0], holder[6]);
    flopenr cell31(ph1, reset, addrflip[3], cellState[1], holder[7]);
    flopenr cell40(ph1, reset, addrflip[4], cellState[0], holder[8]);
    flopenr cell41(ph1, reset, addrflip[4], cellState[1], holder[9]);
    flopenr cell50(ph1, reset, addrflip[5], cellState[0], holder[10]);
    flopenr cell51(ph1, reset, addrflip[5], cellState[1], holder[11]);
    flopenr cell60(ph1, reset, addrflip[6], cellState[0], holder[12]);
    flopenr cell61(ph1, reset, addrflip[6], cellState[1], holder[13]);
    flopenr cell70(ph1, reset, addrflip[7], cellState[0], holder[14]);
    flopenr cell71(ph1, reset, addrflip[7], cellState[1], holder[15]);
    flopenr cell80(ph1, reset, addrflip[8], cellState[0], holder[16]);
    flopenr cell81(ph1, reset, addrflip[8], cellState[1], holder[17]);

    assign #1 memory = holder;

endmodule

```

9.1.9 whoWon Verilog Code

```

module testbench();
    logic [17:0] memory;
    logic gameDone;
    logic [1:0] winner;
    logic col0F, col1F, col2F;
    logic col0, col1, col2;
    logic row0F, row1F, row2F;
    logic row0, row1, row2;

```

```

    logic diag0F, diag1F;
    logic diag0, diag1;
    logic isTie;

    whoWon dut(memory, gameDone, winner, isTie);

    `include "testfixture.verilog"
endmodule

module whoWon (input  logic [17:0] memory,
               output logic gameDone,
               output logic [1:0] winner,
               output logic isTie);

    logic row1F, row2F, row3F;
    logic row1, row2, row3;

    logic col1F, col2F, col3F;
    logic col1, col2, col3;

    logic dia1F, dia2F;
    logic dia1, dia2;

    isSame  rrow1(memory[1], memory[3], memory[5], row1);
    isSame  rrow2(memory[7], memory[9], memory[11], row2);
    isSame  rrow3(memory[13], memory[15], memory[17], row3);

    isSame  ccol1(memory[1], memory[7], memory[13], col1);
    isSame  ccol2(memory[3], memory[9], memory[15], col2);
    isSame  ccol3(memory[5], memory[11], memory[17], col3);

    isSame  ddiag1(memory[1], memory[9], memory[17], dia1);
    isSame  ddiag2(memory[5], memory[9], memory[13], dia2);

    isFull  rrow1F(memory[0], memory[2], memory[4], row1F);
    isFull  rrow2F(memory[6], memory[8], memory[10], row2F);
    isFull  rrow3F(memory[12], memory[14], memory[16], row3F);

    isFull  ccol1F(memory[0], memory[6], memory[12], col1F);
    isFull  ccol2F(memory[2], memory[8], memory[14], col2F);
    isFull  ccol3F(memory[4], memory[10], memory[16], col3F);

    isFull  ddia1F(memory[0], memory[8], memory[16], dia1F);
    isFull  ddia2F(memory[4], memory[8], memory[12], dia2F);

always_comb

```

```

begin

    if (row1 & row1F)
    begin
        gameDone = 1;
        winner = {memory[0],memory[1]};
        isTie = 0;

    end

    else if (row2 & row2F)
    begin
        gameDone = 1;
        winner = {memory[6],memory[7]};
        isTie = 0;

    end

    else if (row3 & row3F)
    begin
        gameDone = 1;
        winner = {memory[12],memory[13]};
        isTie = 0;

    end

    else if (col1 & col1F)
    begin
        gameDone = 1;
        winner = {memory[0],memory[1]};
        isTie = 0;

    end

    else if (col2 & col2F)
    begin
        gameDone = 1;
        winner = {memory[2],memory[3]};

    end

    else if (col3 & col3F)
    begin
        gameDone = 1;
        winner = {memory[4],memory[5]};
        isTie = 0;

    end

```

```

        else if (dia1 & dia1F)
begin
    gameDone = 1;
    winner = {memory[0],memory[1]};
    isTie = 0;
end
    else if (dia2 & dia2F)
begin
    gameDone = 1;
    winner = {memory[4],memory[5]};
    isTie = 0;

end

    else if (row1F & row2F & row3F & col1F & col2F & col3F & dia1F &
dia2F)
begin
    gameDone = 1;
    winner = 1;
    isTie = 1;
end

    else
begin
    gameDone = 0;
    winner = 0;
    isTie = 0;
end

end

endmodule

```

9.2 Test Vectors

9.2.1 isFull.tv

```

0000
0010
0100
0110
1000
1010
1100
1111

```

9.2.2 isSame.tv

```
0001
0010
0100
0110
1000
1010
1100
1111
xxxx
```

9.2.3 isTwoSame.tv

```
000101111
010100011
010001101
010111000
000000000
111111000
010001101
001111111
110000000
110011101
```

9.2.4 address.tv

```
0000__000_000_001
0001__000_000_010
0010__000_000_100
0011__000_001_000
0100__000_010_000
0101__000_100_000
0110__001_000_000
0111__010_000_000
1000__100_000_000
1001__000_000_000
1010__000_000_000
1011__000_000_000
1100__000_000_000
1101__000_000_000
1110__000_000_000
1111__000_000_000
```

9.2.5 traverse.tv

```
0000_11111111111111110
0001_111111111111111011
0010_111111111111101111
0011_111111111110111111
0100_111111111011111111
```

```
0100_000011111011111111
0100_000000001011111111
0100_100000111011111111
0101_100000001111111111
```

9.2.6 whoWon.tv

```
0_11_1_11111111111010101
0_11_1_000000000000111111
0_10_1_000000000000010101
0_00_0_000000000000000000
1_01_1_110111011101011101
0_10_1_000000010101000000
0_11_1_111111000000000000
0_10_1_000001000001000001
0_10_1_000001001101111101
0_11_1_001100001100001100
0_11_1_011101001101001100
0_10_1_010000010000010000
0_11_1_110001110001110100
0_10_1_000001000100010000
0_11_1_110000001100000011
0_10_1_010101000000111111
```

9.2.7 memBoard.tv

```
00000000000000000000_0000_11_0_1
00000000000000000000_0000_11_1_1
00000000000000000000_0000_11_0_0
00000000000000000011_0000_11_1_0
00000000000000000011_0001_11_0_0
00000000000000001111_0001_11_1_0
00000000000000001111_0011_11_0_0
0000000000011001111_0011_11_1_0
0000000000011001111_0001_11_0_0
00000000000000000000_0001_11_1_0
```

9.2.8 flopenr.tv

```
0_0_0_0_1
0_0_0_0_0
0_1_1_1_1
0_1_1_1_0
0_0_1_1_1
0_0_1_1_0
0_0_0_0_0
```

9.2.9 computer.tv


```

10_0000_10_111111111111010100
10_0001_10_111111111111010001
10_0010_10_111111111111000101
10_0011_10_111111010100111111
10_0011_10_111111111100111111
10_0100_10_111111010001111111
10_0101_10_111111000101111111
10_0000_10_000000000000000000
10_0001_10_000000000000000011
10_0000_10_111101111101111100
10_0001_10_111111111111110011
10_0011_10_111111111100111111
10_0000_10_111111111111110000
10_0001_10_001100001100000000
10_0100_10_000011000000110000
00_1111_00_000011000000110000

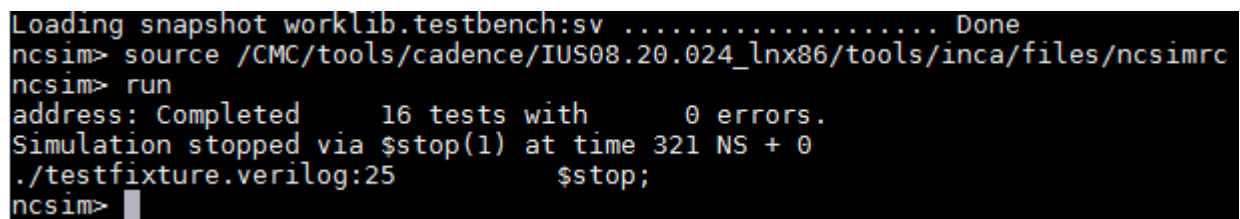
```

9.2.10 controller.tv

```

1_1_0_0_0000_0000_0_00_1111_00_00
0_0_0_0_0000_0000_0_00_1111_00_00
1_0_1_0_0001_0000_0_01_0001_10_10
0_0_0_0_0001_0000_0_01_0001_10_10
1_0_0_0_0001_0010_0_10_0010_01_01
1_0_0_0_0000_0010_0_01_0000_10_10
1_0_0_0_1111_0101_0_10_0101_01_01
1_0_0_0_0100_0111_0_01_0100_10_10
1_0_0_0_1111_1000_0_10_1000_01_01

```

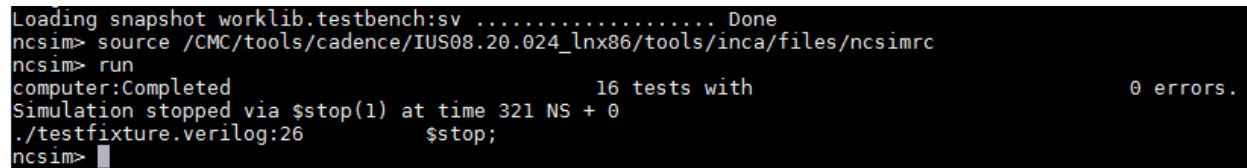


```

Loading snapshot worklib.testbench:sv ..... Done
ncsim> source /CMC/tools/cadence/IUS08.20.024_lnx86/tools/inca/files/ncsimrc
ncsim> run
address: Completed      16 tests with      0 errors.
Simulation stopped via $stop(1) at time 321 NS + 0
./testfixture.verilog:25      $stop;
ncsim> █

```

Figure 5: address passed



```

Loading snapshot worklib.testbench:sv ..... Done
ncsim> source /CMC/tools/cadence/IUS08.20.024_lnx86/tools/inca/files/ncsimrc
ncsim> run
computer:Completed      16 tests with      0 errors.
Simulation stopped via $stop(1) at time 321 NS + 0
./testfixture.verilog:26      $stop;
ncsim> █

```

Figure 6: computer passed

```

Loading snapshot worklib.testbench:sv ..... Done
ncsim> source /CMC/tools/cadence/IUS08.20.024_lnx86/tools/inca/files/ncsimrc
ncsim> run
isFull: Completed 8 tests with 0 errors.
Simulation stopped via $stop(1) at time 161 NS + 0
./testfixture.verilog:23 $stop;
ncsim> █

```

Figure 7: isFull passed

```

Loading snapshot worklib.testbench:sv ..... Done
ncsim> source /CMC/tools/cadence/IUS08.20.024_lnx86/tools/inca/files/ncsimrc
ncsim> run
isSame: Completed 8 tests with 0 errors.
Simulation stopped via $stop(1) at time 161 NS + 0
./testfixture.verilog:23 $stop;
ncsim> █

```

Figure 8: isSame passed

```

Loading snapshot worklib.testbench:sv ..... Done
ncsim> source /CMC/tools/cadence/IUS08.20.024_lnx86/tools/inca/files/ncsimrc
ncsim> run
isTwoSame: Completed 10 tests with 0 errors.
Simulation stopped via $stop(1) at time 201 NS + 0
./testfixture.verilog:27 $stop;
ncsim> █

```

Figure 9: isTwoSame passed

```

Loading snapshot worklib.testbench:sv ..... Done
ncsim> source /CMC/tools/cadence/IUS08.20.024_lnx86/tools/inca/files/ncsimrc
ncsim> run
traverse: Completed 11 tests with 0 errors.
Simulation stopped via $stop(1) at time 221 NS + 0
./testfixture.verilog:25 $stop;
ncsim> █

```

Figure 10: traverse passed

```

Loading snapshot worklib.testbench:sv ..... Done
ncsim> source /CMC/tools/cadence/IUS08.20.024_lnx86/tools/inca/files/ncsimrc
ncsim> run
whoWon: Completed 16 tests with 0 errors.
Simulation stopped via $stop(1) at time 321 NS + 0
./testfixture.verilog:25 $stop;
ncsim> quit

```

Figure 11: whoWon passed

```
ncsim> run
Completed          10 tests with          0 errors.
Simulation stopped via $stop(1) at time 201 NS + 0
./testfixture.verilog:25      $stop;
```

Figure 12: memBoard passed

```
ncsim> run
Controller: Completed          7 tests with          0 errors
Simulation complete via $finish(1) at time 142 NS + 0
./testfixture.verilog:27      $finish;
```

Figure 13: Controller passed

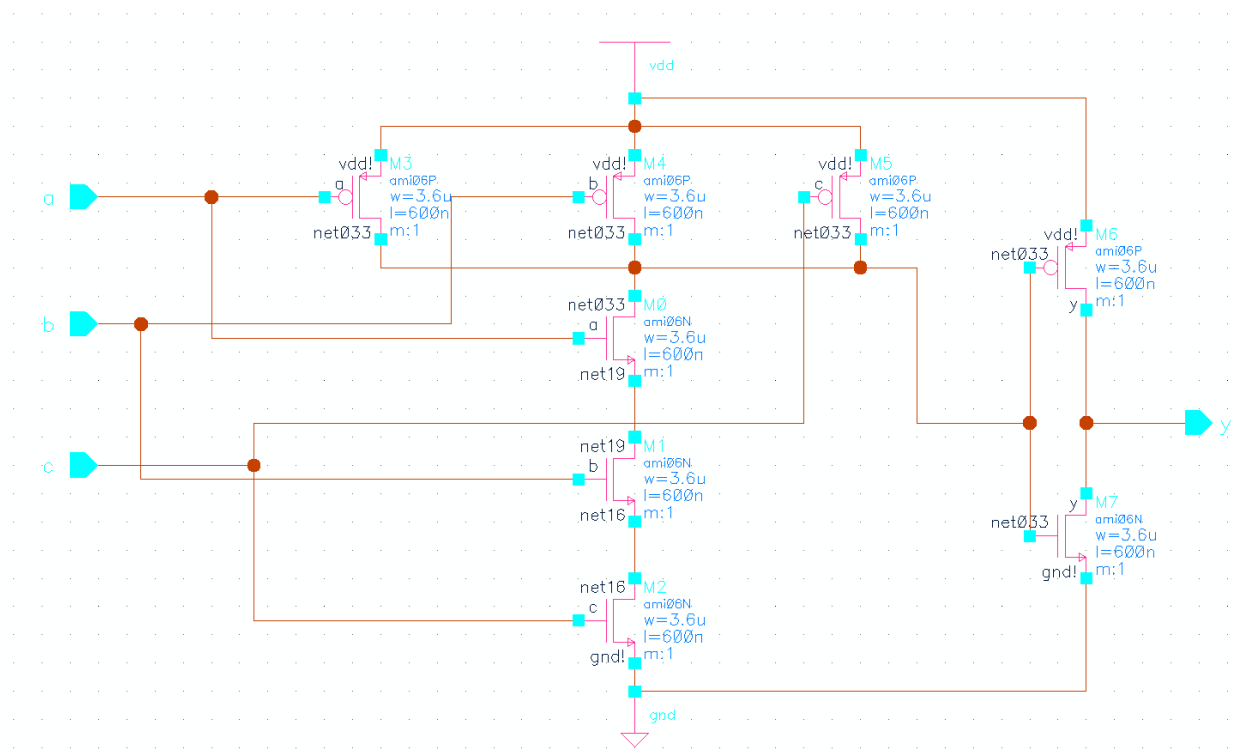


Figure 14: isFull schematic

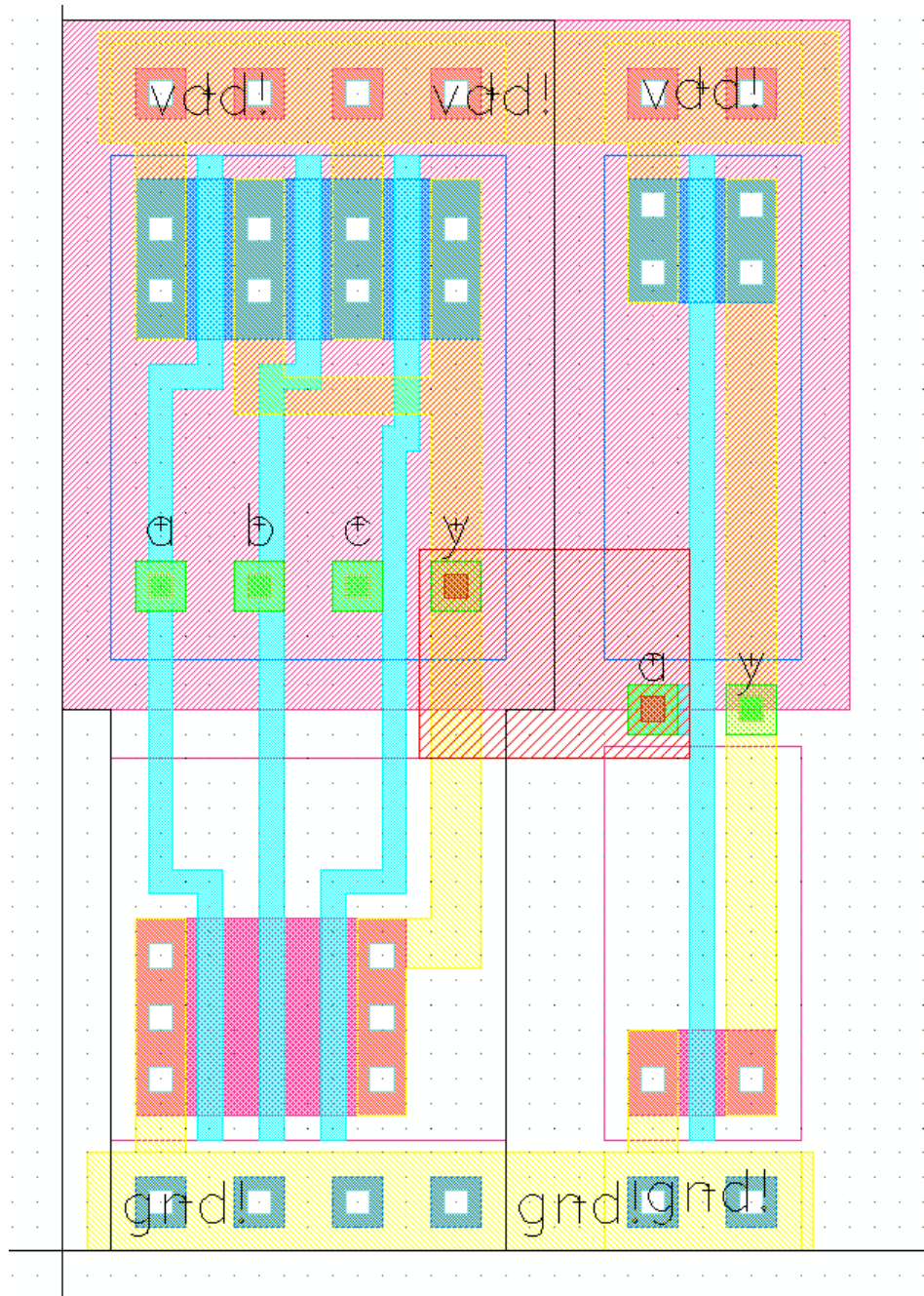


Figure 15: isFull layout

```
DRC started.....Tue Mar 29 14:51:17 2016
completed ....Tue Mar 29 14:51:18 2016
CPU TIME = 00:00:00  TOTAL TIME = 00:00:01
***** Summary of rule violations for cell "isFull layout" *****
Total errors found: 0
```

Figure 16: isFull passes DRC

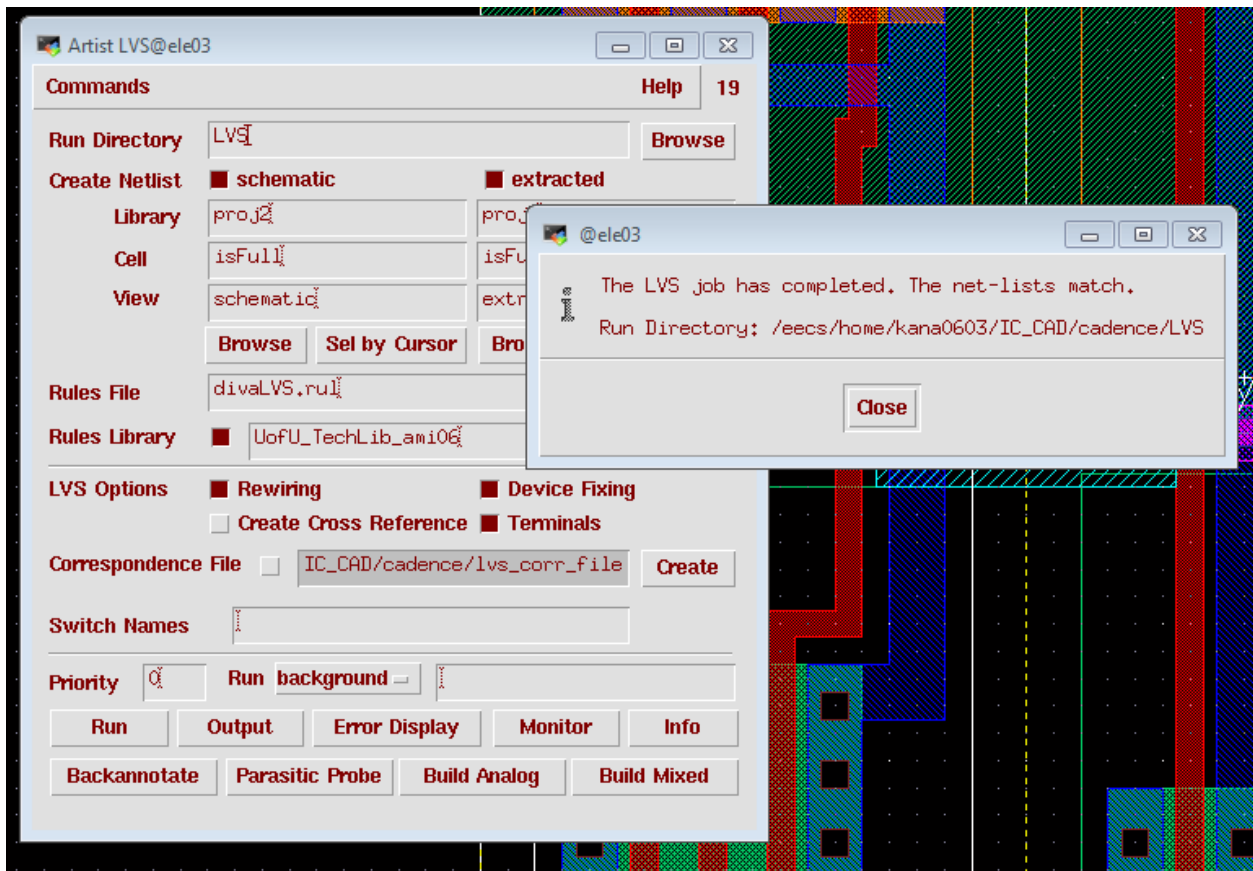


Figure 17: isFull passes LVS

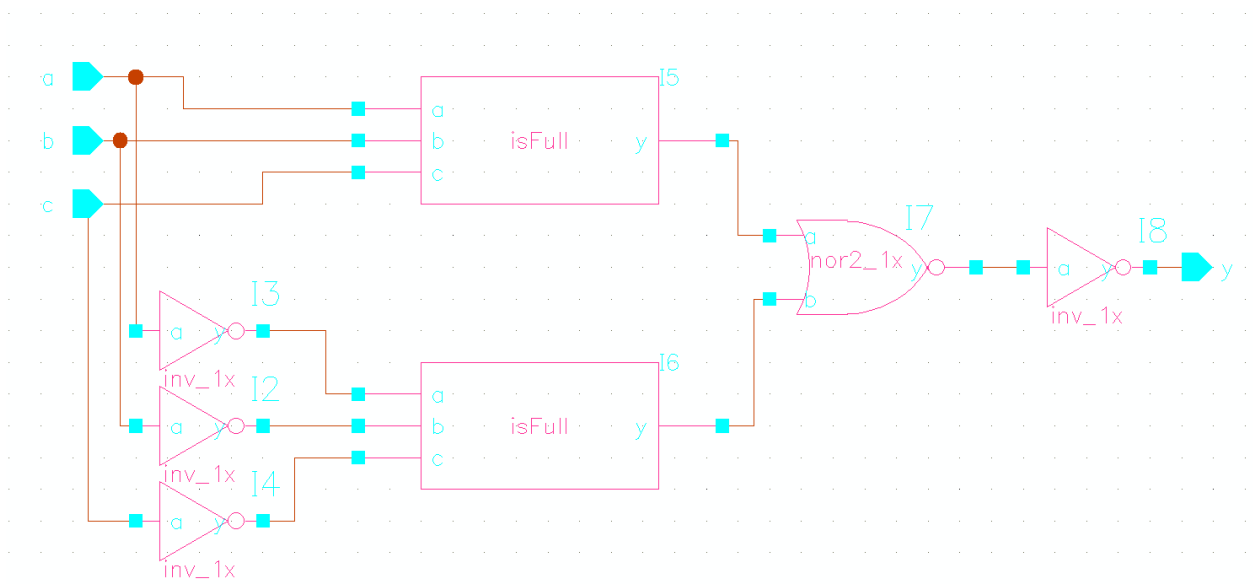


Figure 18: isSame schematic

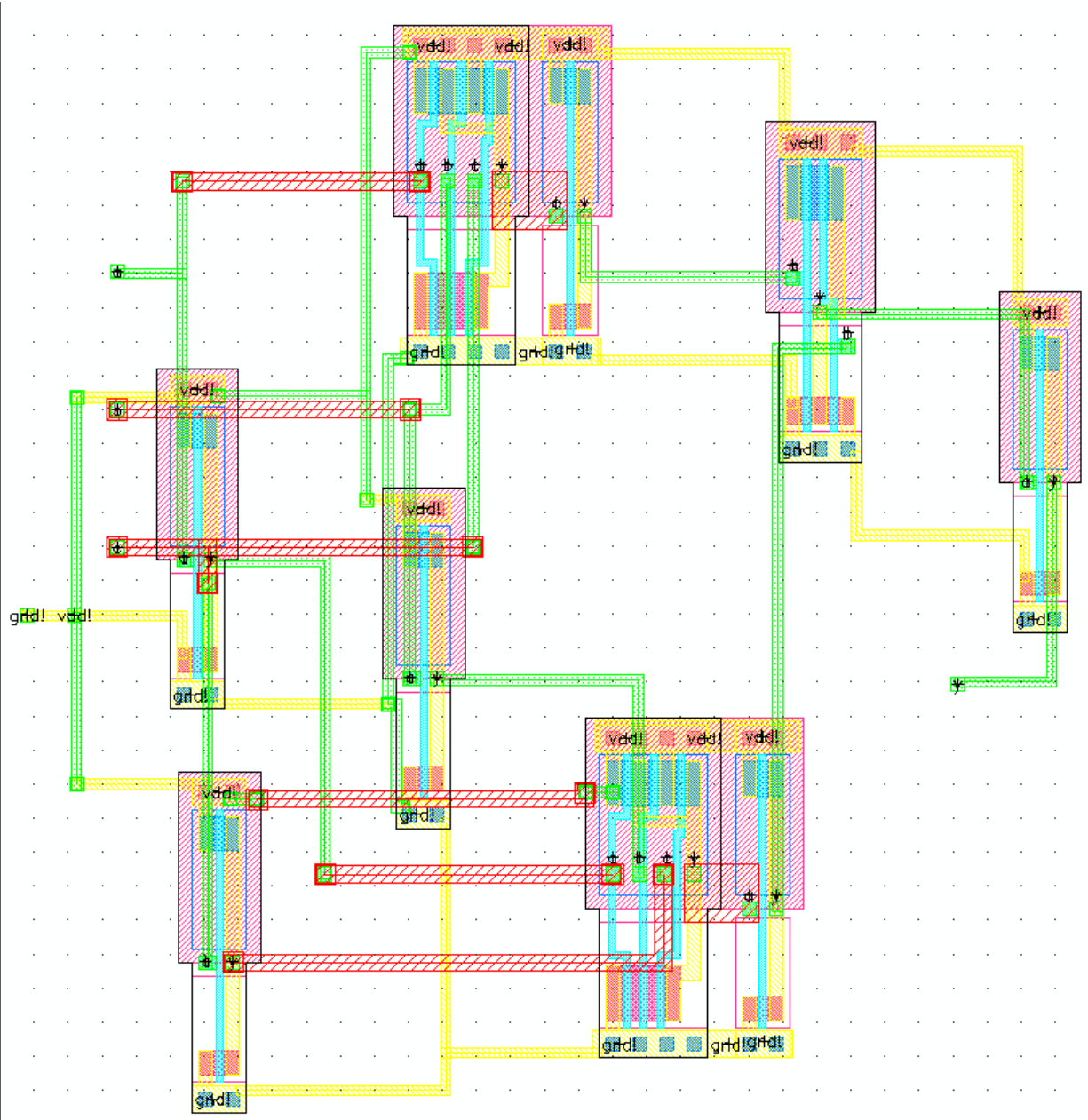


Figure 19: isSame layout

```
DRC started.....Tue Mar 29 14:55:30 2016
completed .....Tue Mar 29 14:55:31 2016
CPU TIME = 00:00:00  TOTAL TIME = 00:00:01
***** Summary of rule violations for cell "isSame layout" *****
Total errors found: 0
```

Figure 20: isSame passes DRC

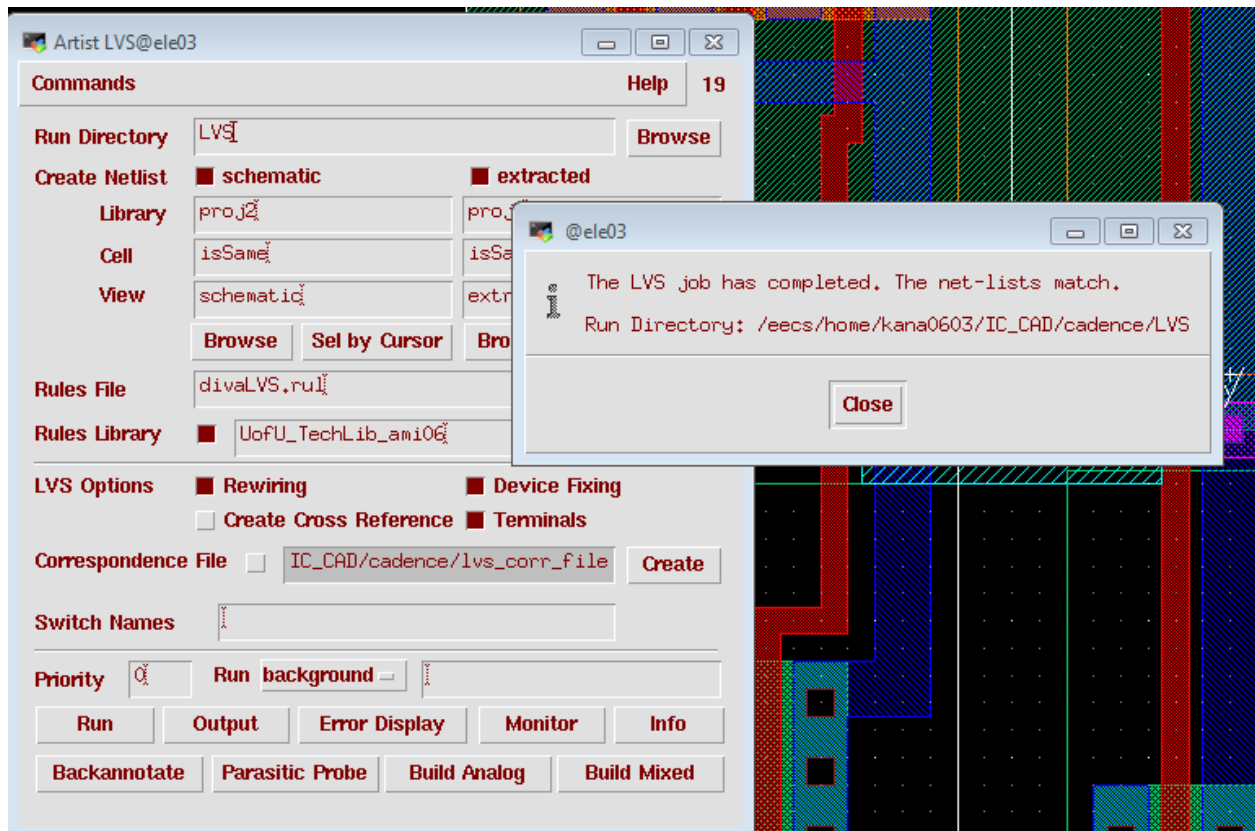


Figure 21: isSame passes LVS

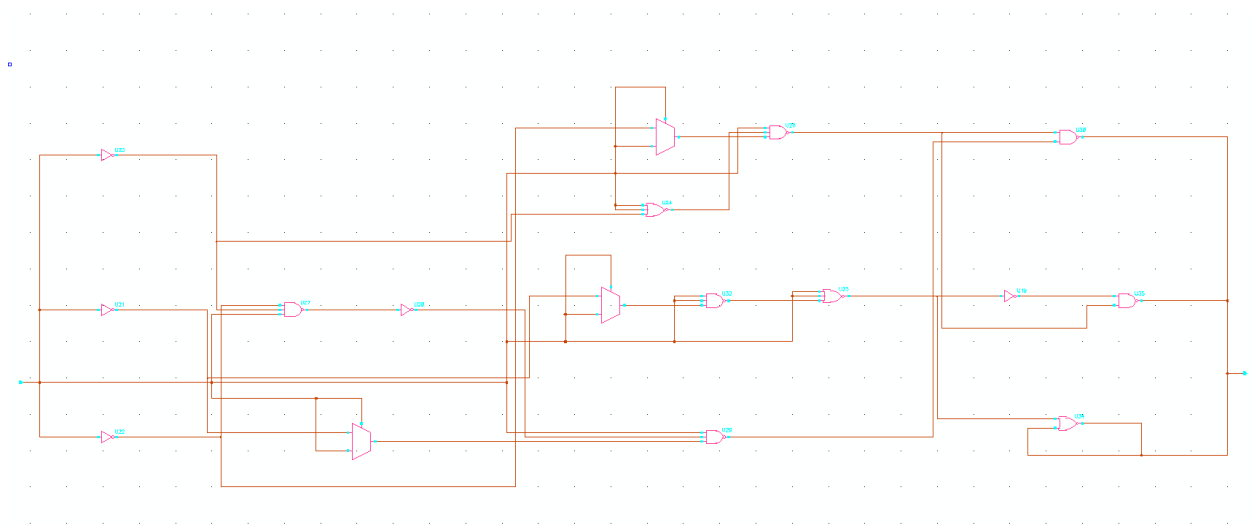


Figure 22: isTwoSame schematic

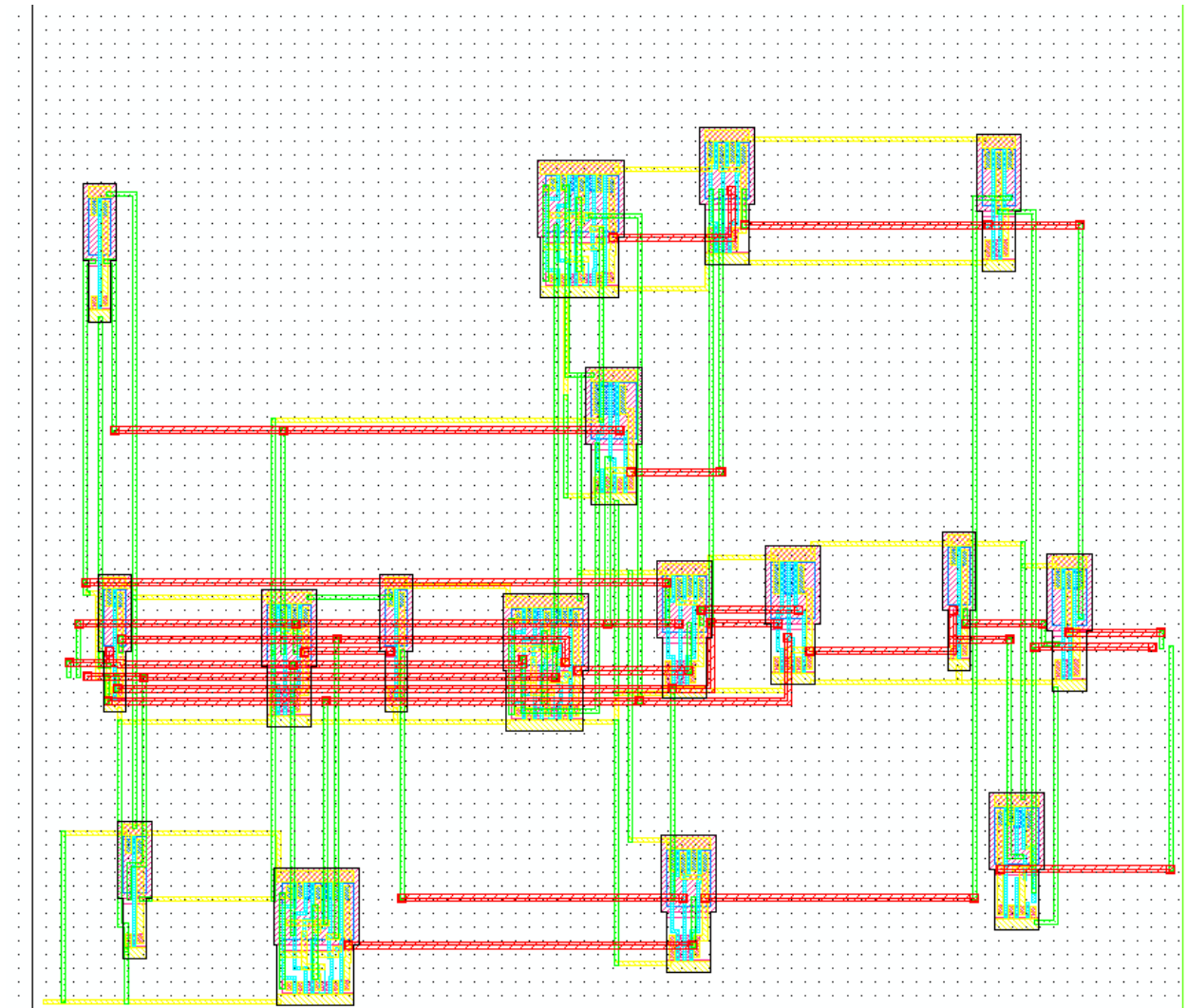


Figure 23: isTwoSame layout

```
DRC started.....Tue Mar 29 15:31:47 2016
completed ....Tue Mar 29 15:31:48 2016
CPU TIME = 00:00:00  TOTAL TIME = 00:00:01
***** Summary of rule violations for cell "istwoSame2 layout2" *****
Total errors found: 0
```

Figure 24: isTwoSame passes DRC

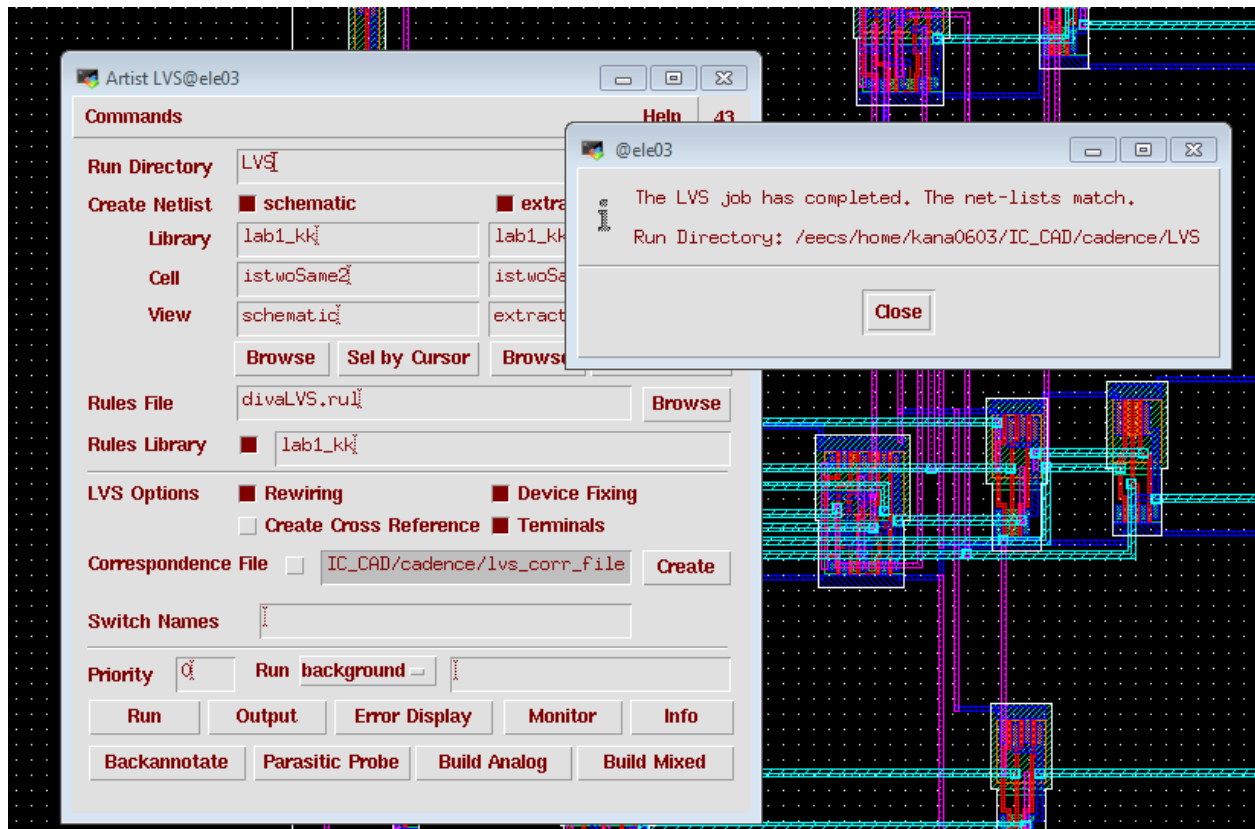


Figure 25: isTwoSame passes LVS

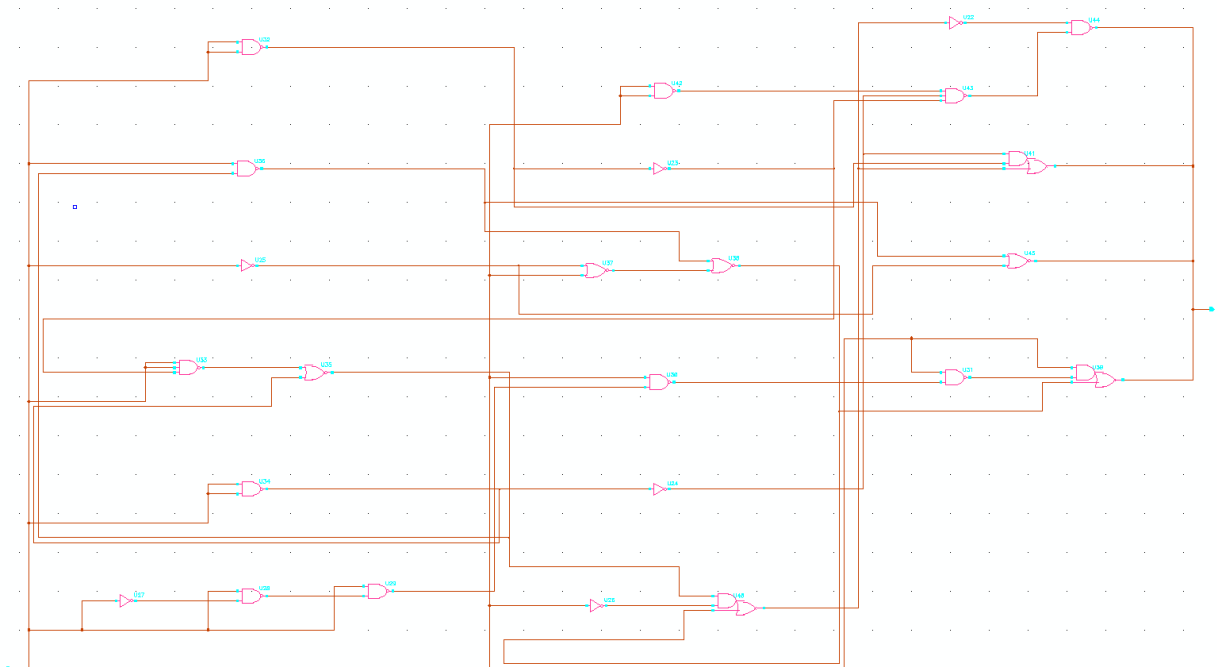


Figure 26: traverse schematic

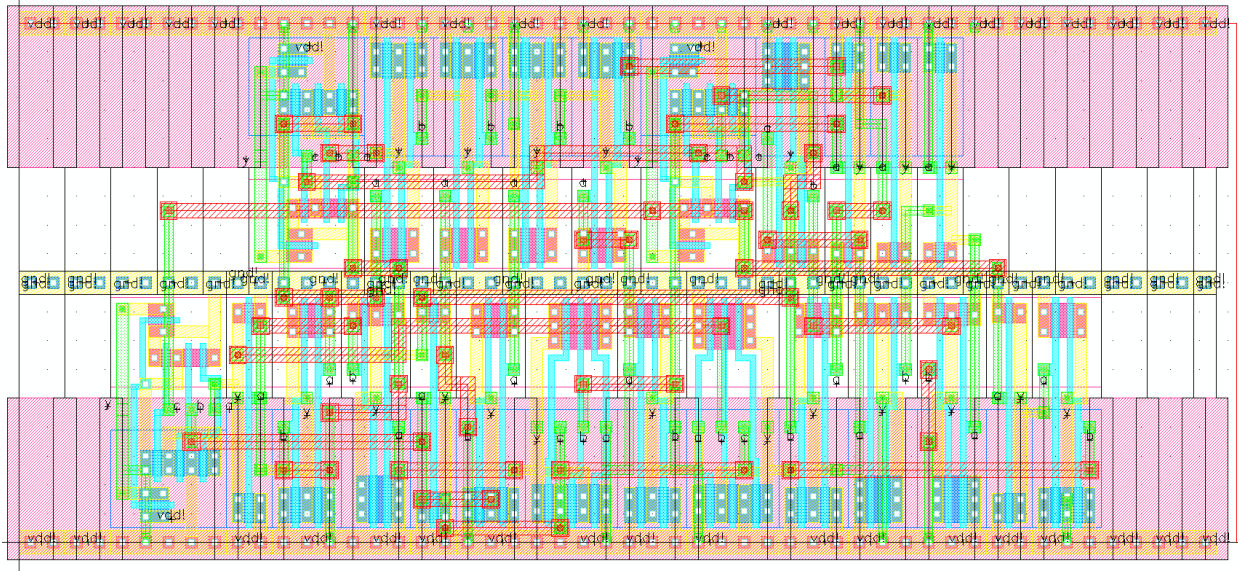


Figure 27: traverse layout

```
DRC started.....Tue Mar 29 14:46:26 2016
completed ....Tue Mar 29 14:46:27 2016
CPU TIME = 00:00:00 TOTAL TIME = 00:00:01
***** Summary of rule violations for cell "traverse2 layout" *****
Total errors found: 0
```

Figure 28: traverse passes DRC

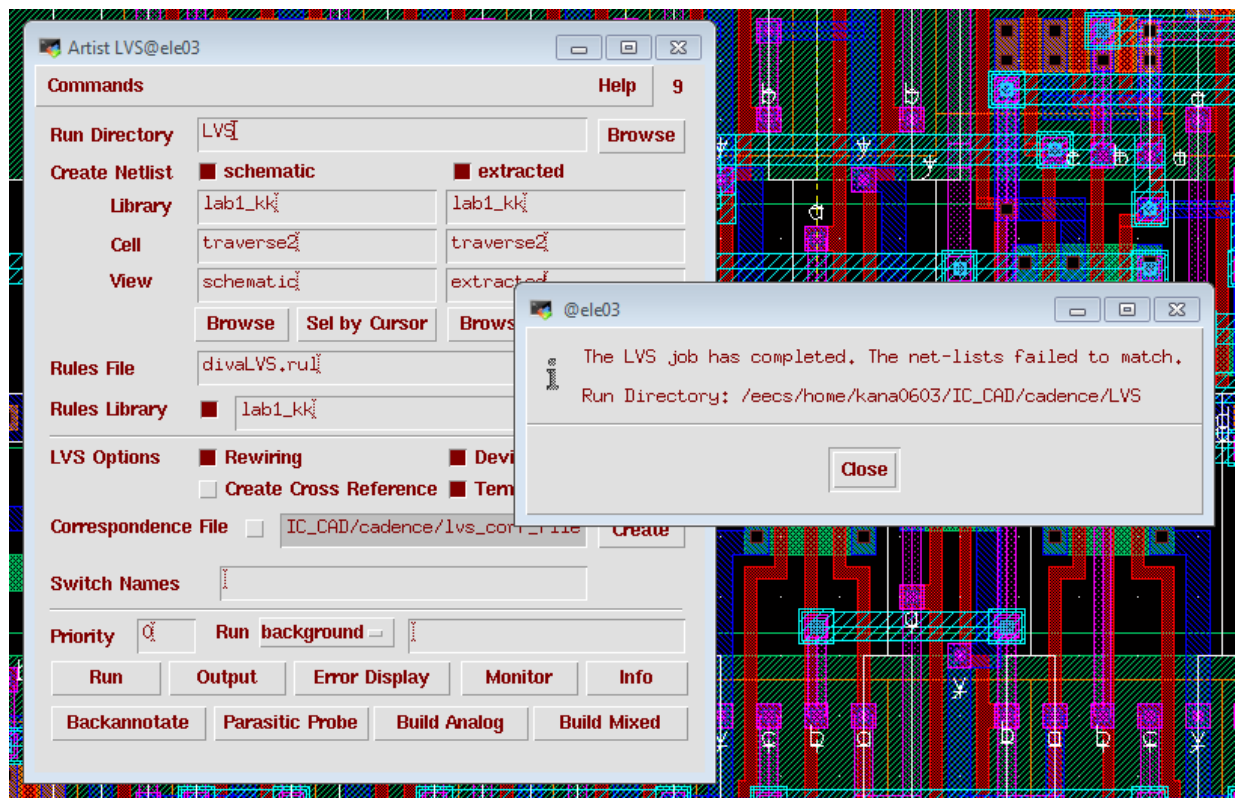


Figure 29: traverse passes LVS

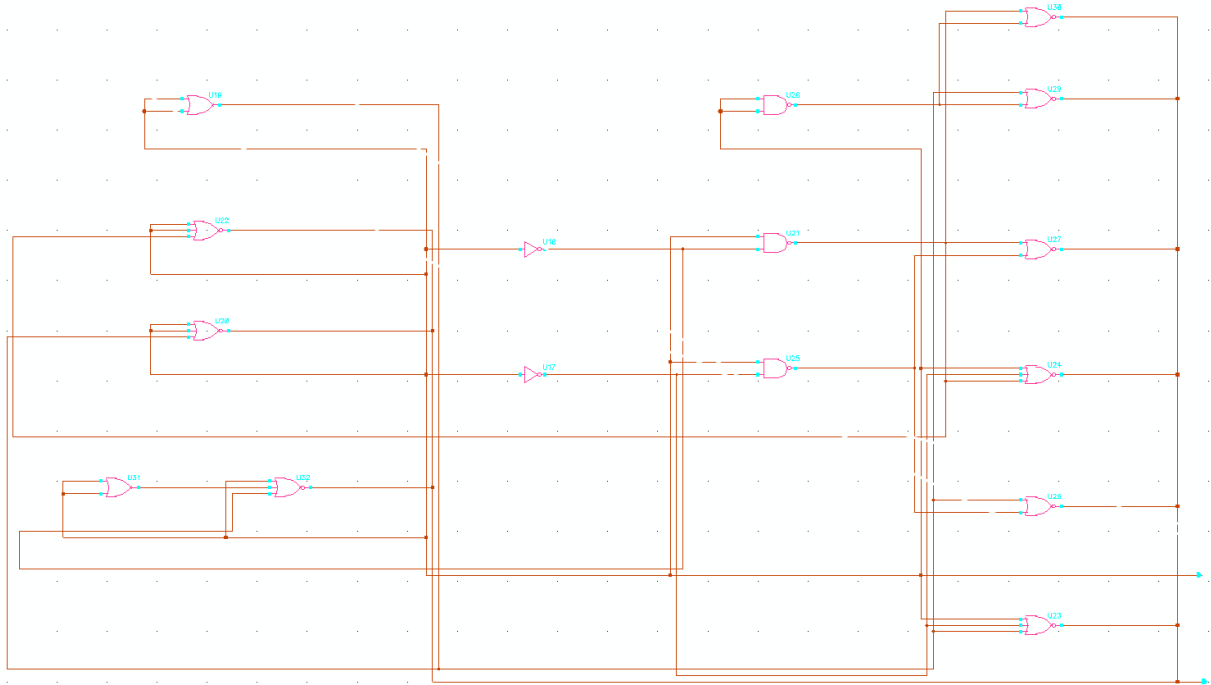


Figure 30: address schematic

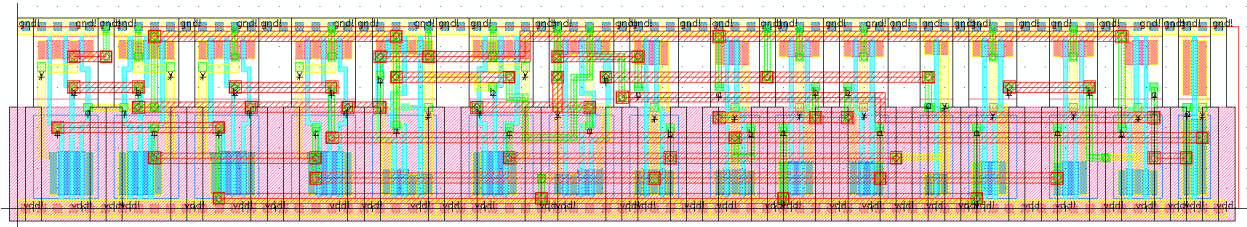


Figure 31: address layout

```
DRC started.....Tue Mar 29 14:43:42 2016
completed ....Tue Mar 29 14:43:42 2016
CPU TIME = 00:00:00  TOTAL TIME = 00:00:00
***** Summary of rule violations for cell "address2 layout" *****
Total errors found: 0
```

Figure 32: address passes DRC

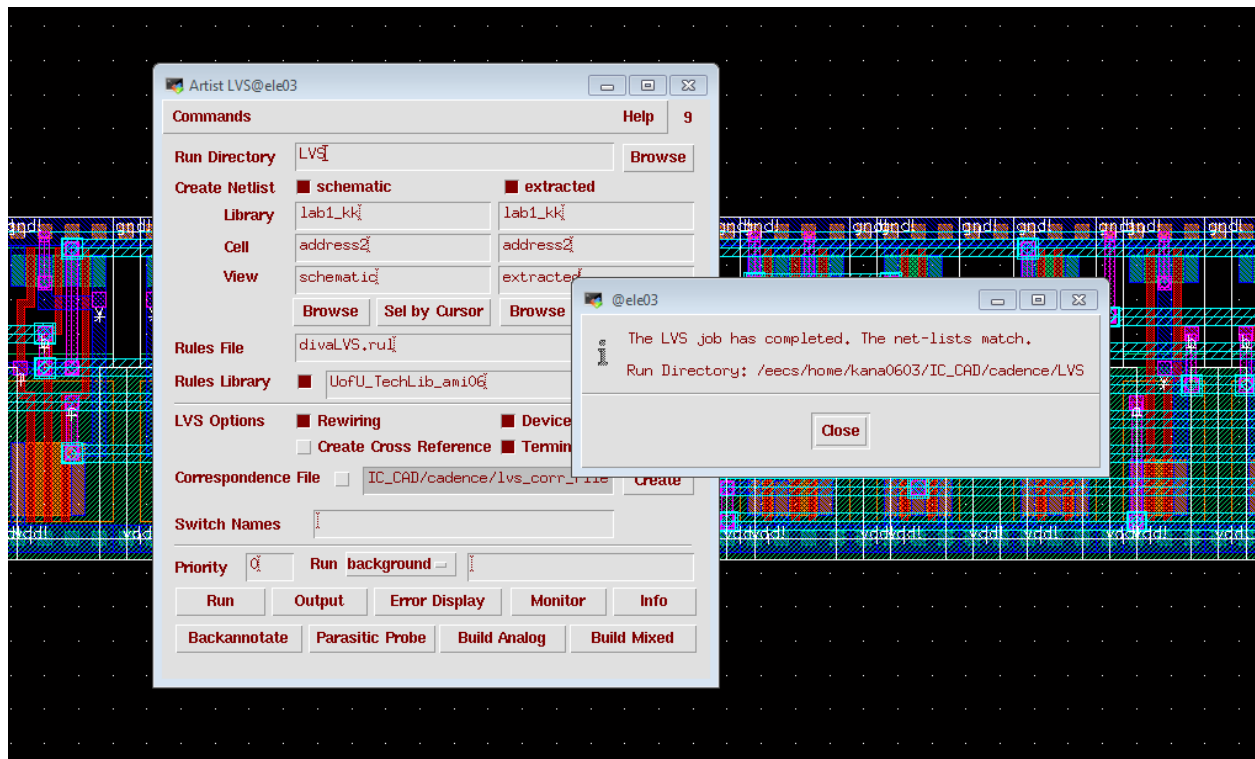


Figure 33: address passes LVS

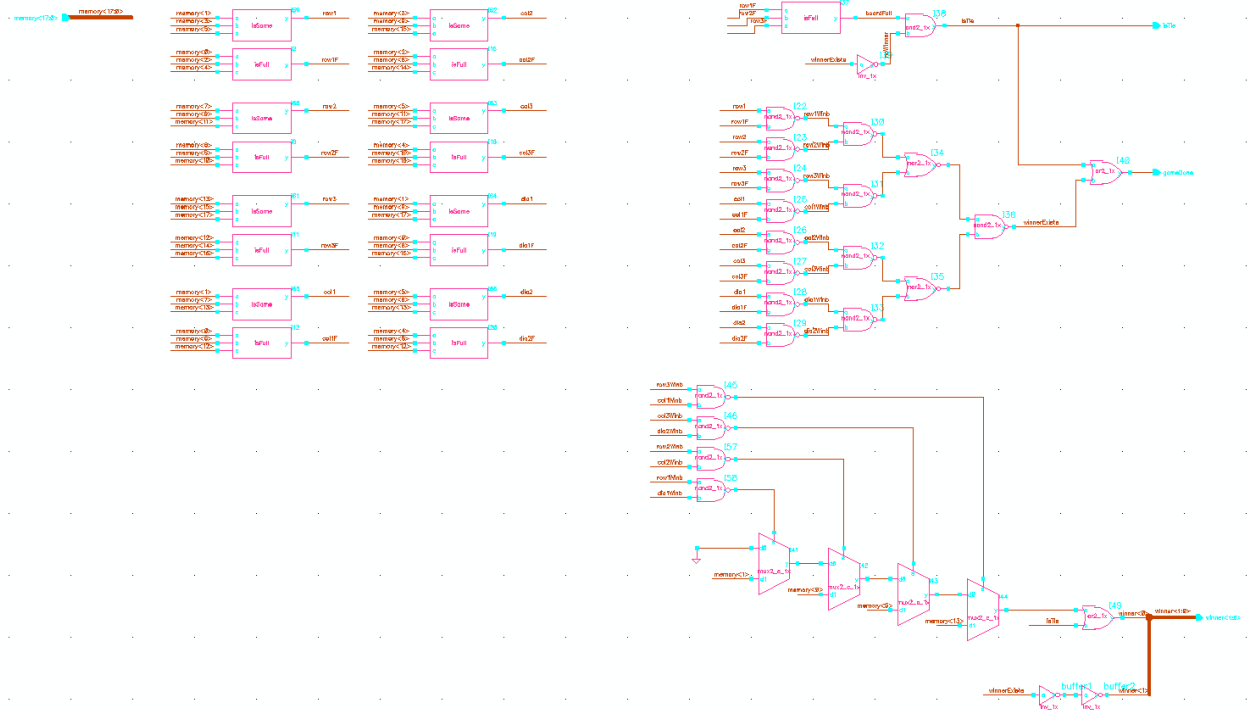


Figure 34: whoWon schematic

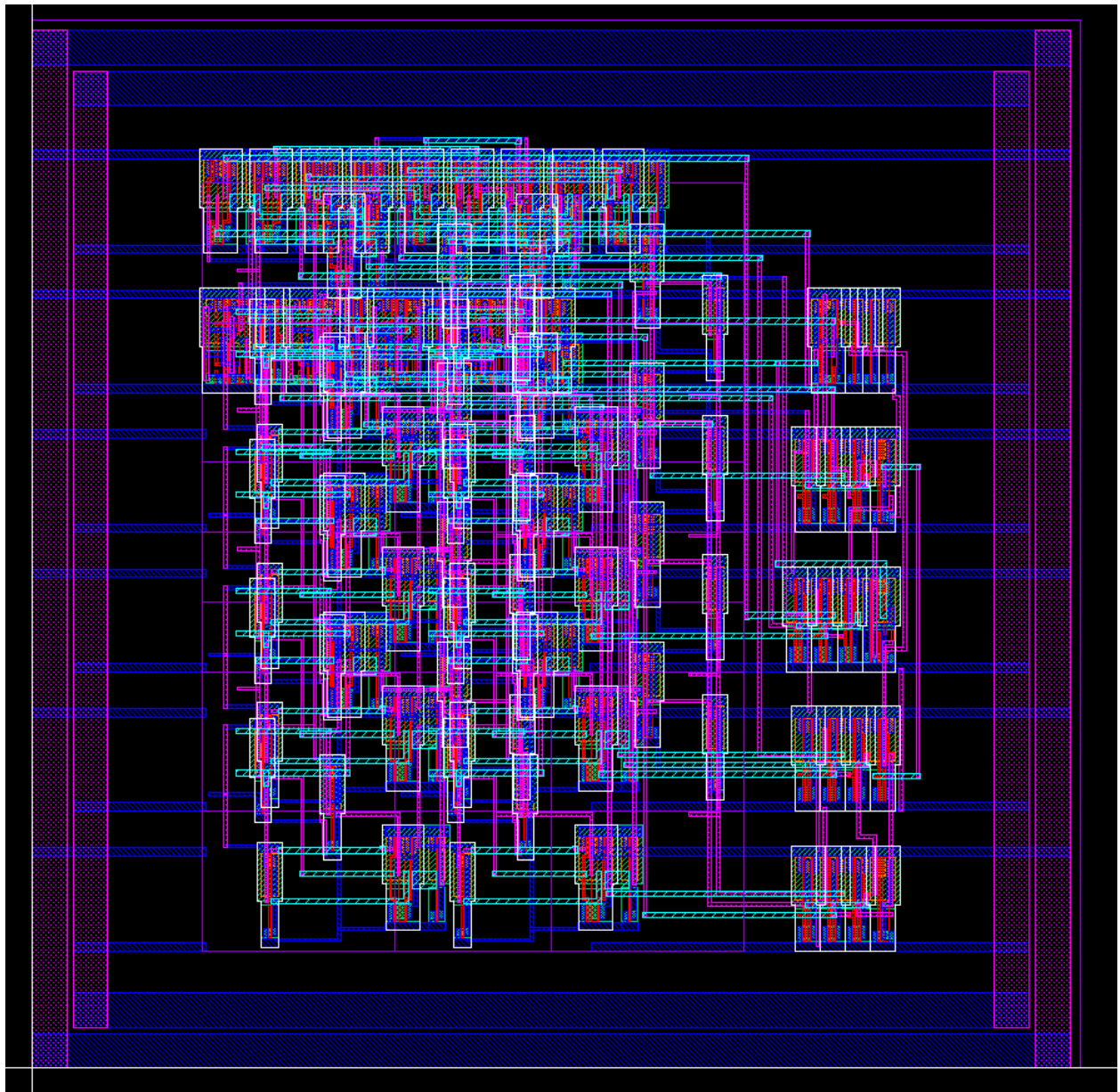


Figure 35: whoWon layout

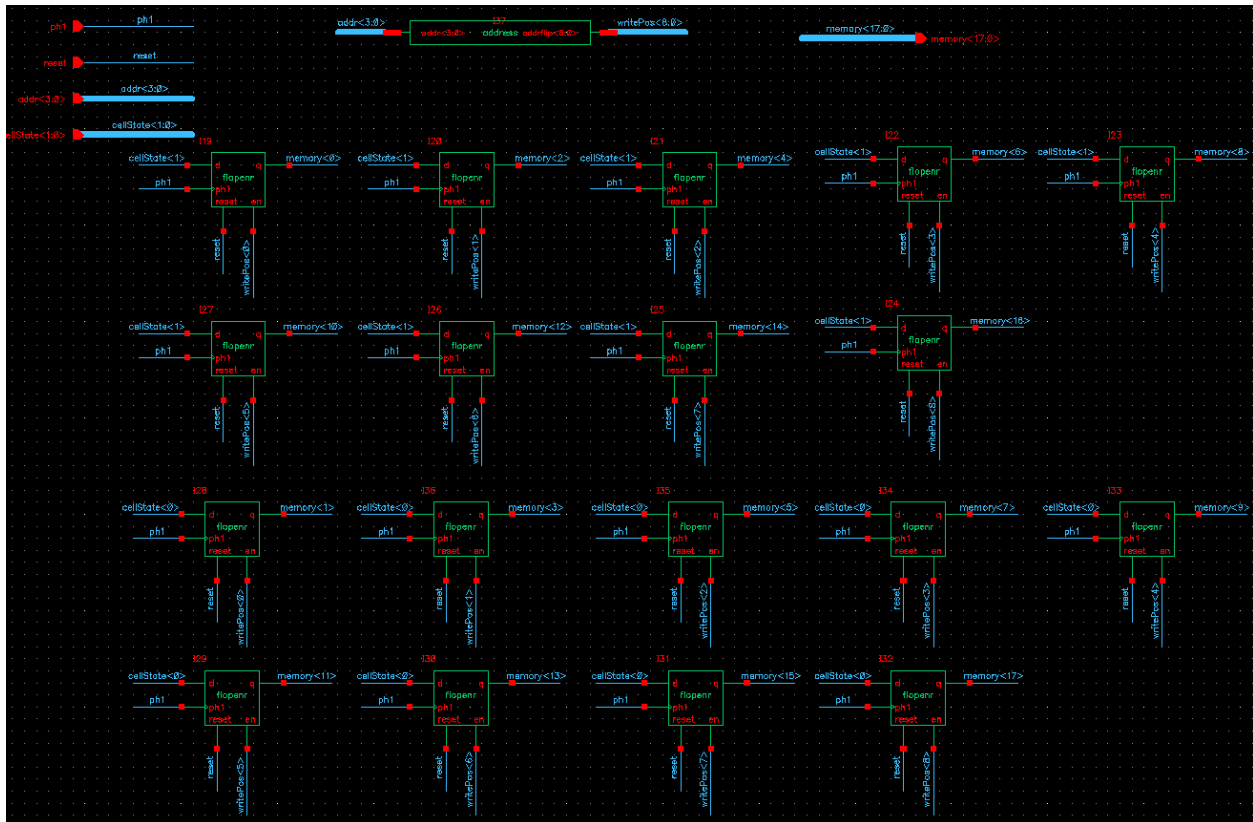


Figure 36: memBoard schematic

```
icfb - Log: /eecs/home/ariel555/CDS.log@ele03
File Tools Options
executing: drc(highresEdge geomGetEdge(geomAndNot(elec geomButting(elec elecHighres))) (sep < (...
executing: saveDerived(geomButting(elecHighres geomAndNot(elec elecHighres) (ignore == 2)) errM...
executing: saveDerived(geomAnd(elecHighres nwell) "(SCMOS Rule 27,6) resistor must be outside w...
executing: drc(elecHighresEdge (width < (lambda * 5.0)) errMesg)
drc(elecHighresEdge (sep < (lambda * 7.0)) errMesg)
drc(elecHighresEdge (notch < (lambda * 7.0)) errMesg)
executing: drc(highresEdge elecHighresEdge (enc < (lambda * 2.0)) errMesg)
DRC started.....Mon Apr 4 23:40:37 2016
completed ....Mon Apr 4 23:40:40 2016
CPU TIME = 00:00:00 TOTAL TIME = 00:00:03
***** Summary of rule violations for cell "memBoard layout" *****
Total errors found: 0
***** Summary of rule violations for cell "memBoard layout" *****
mouse L: showClickInfo() M: leHiMousePopUp() R: setDRCForm()
>
```

Figure 37: memBoard passed DRC

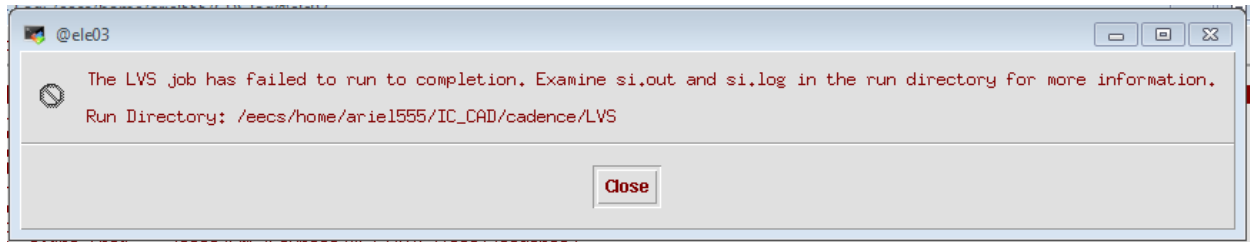


Figure 38: memBoard passed LVS but the file got corrupted so we could not get LVS to run (b).

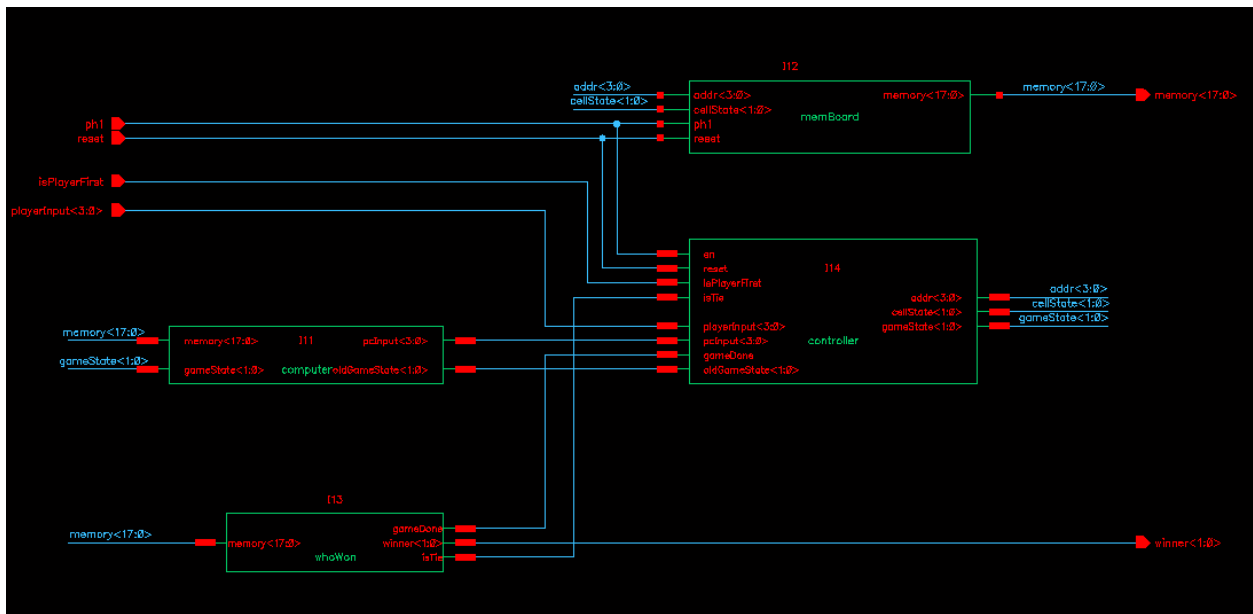


Figure 39: core schematic

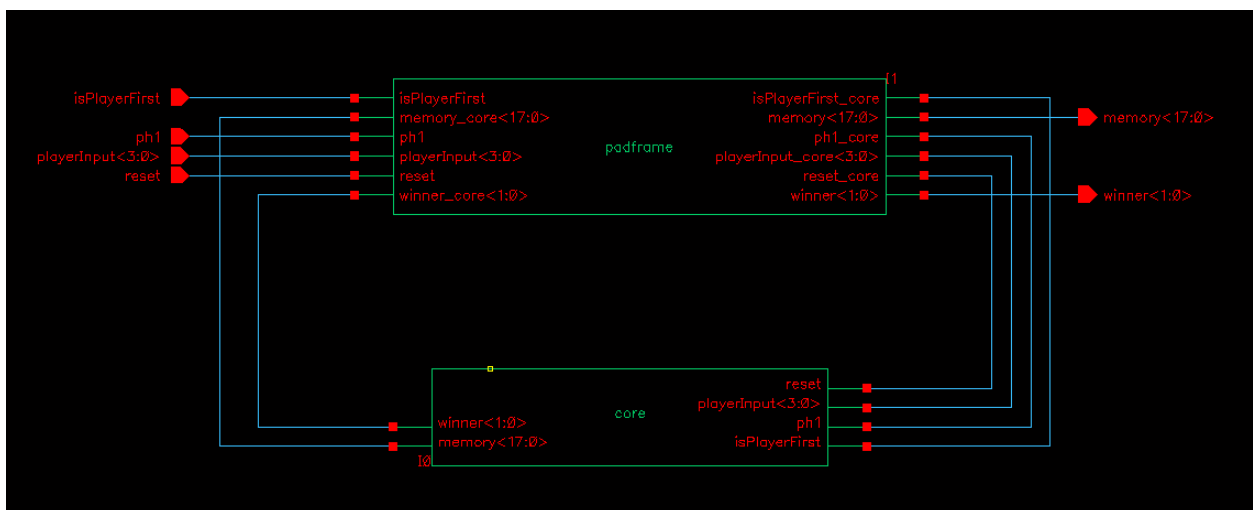


Figure 40: chip schematic

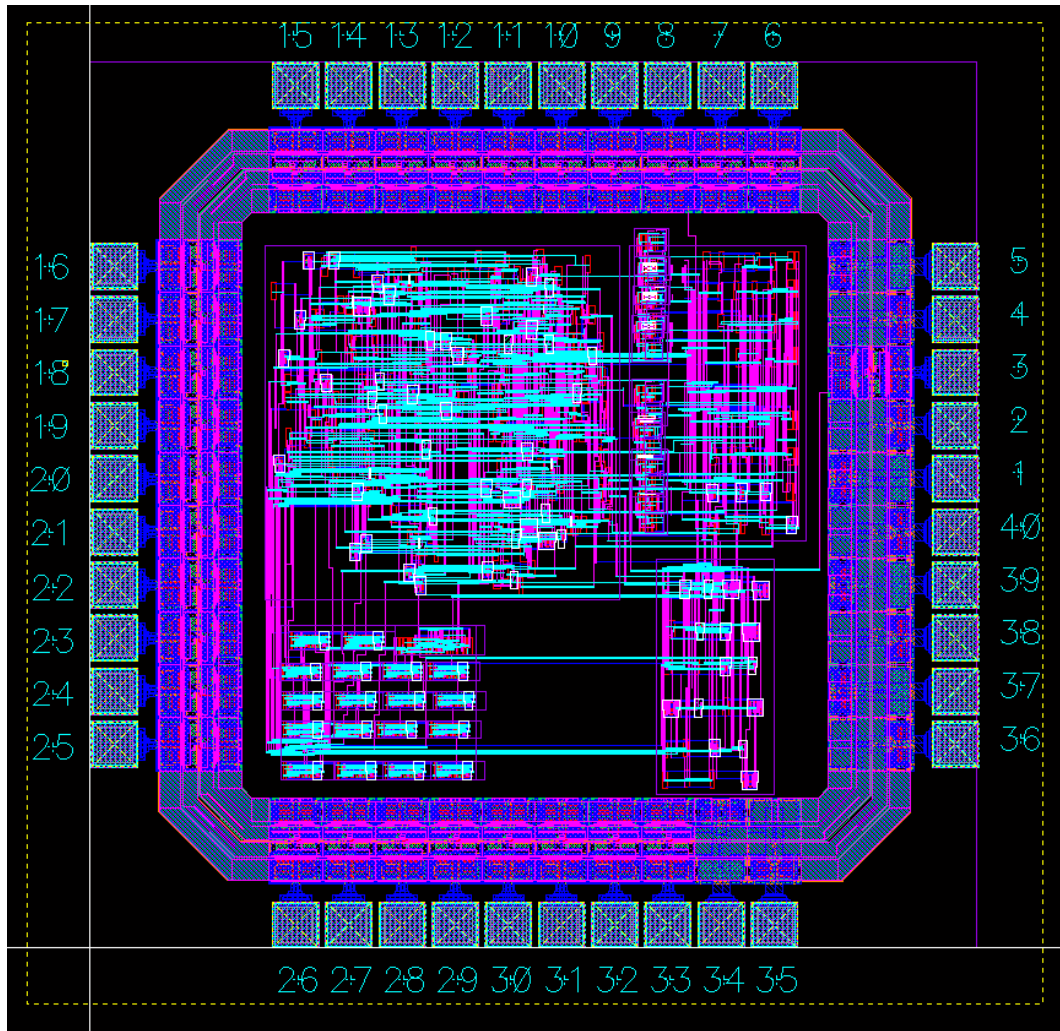


Figure 41: chip layout