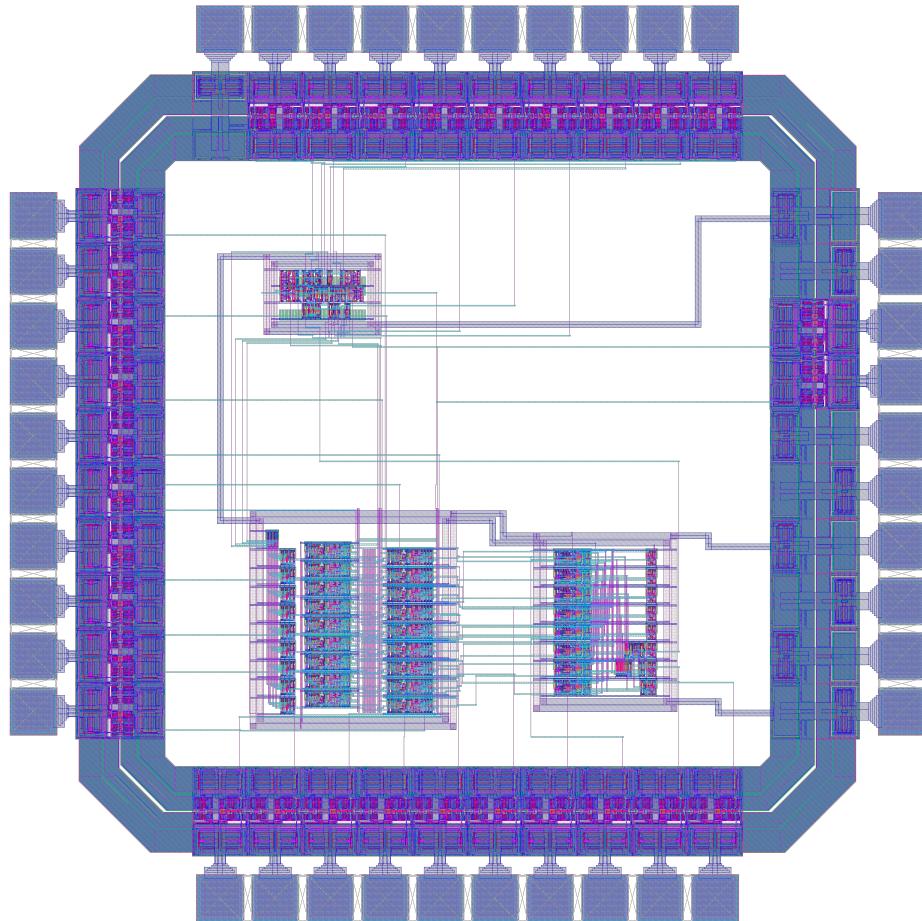


# E158 - VLSI Final Project: Tic-Tac-Toe

Katherine Yang and Guillaume Legrain

April 2015



# 1 Introduction

This project is a 3x3 game of Tic-tac-toe built on a  $0.6 \mu m$  process on a  $1.5 \times 1.5 mm$  40-pin MOSIS “TinyChip”. The goal of this project was to create a Tic-Tac-Toe board which is able to monitor the state of a game and return the win/lose/draw state of the game after each player’s move. Specifically, for each player’s turn, the game records the player’s inputs and at the end of the game determines if the game is done and who the winner is. The game allows two players to play against each other. Each of the nine spaces can be “X”, “O” or blank.

## 1.1 Game representation

The Tic-tac-toe game board is represented by 8 cells. Each cell is either empty, a cross or a circle using 2-bits as shown in fig. 1. Thus, the game board is represented by an array of 18-bits as shown in fig. 2.

## 1.2 Architecture

The architecture consists of 3 main modules, one of which was synthesized, and two other were hand laid.

### 1.2.1 Memory Array

The memory array remembers the status of the tic-tac-toe game board through sequential logic. It consists of enable reset flip-flops and a 4 to 8 bit decoder.

### 1.2.2 Check Win Status

The win status module is a combinational logic block that checks the win state of the tic-tac-toe board. There are two custom made leaf cells in this module. This cell is very repetitive and can be organised as a datapath.

### 1.2.3 Game Controller

The game controller module is a finite state machine which switches between players, player input and game board. This module will be synthesized as the structure is more irregular and harder to be hand laid.

## 2 Specifications

The chip has a total of 31 I/O pins plus 5 GND pins and 4 VDD pins. User input is a 4-bit vector (`playerInput<3:0>`) to describe the cell number at which to play. The write input signals the game controller that the `playerInput` is ready to read/write. `gameState<1:0>` is used to represent the state of the game: player 1’s turn, player 2’s turn or end. As the name implies `winner<1:0>` describes who won the game: player 1, player 2, tie when the game is not done. To prevent hold time issues, the system will be using two non-overlapping clocks. Thus, the chip uses a two-phase clock with pins `ph1` and `ph2`. A list of all inputs can be found in table 1.

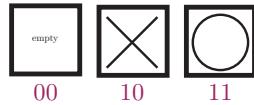


Figure 1: Each cell of the game board is represented using two bits. The MSB is used to describe if the cell has been played or not. The LSB is used to describe who played in the cell.

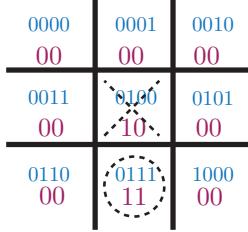


Figure 2: The playable game board is represented by 18-bits. Each pair of bits are used to describe a cell. Each cell holds a two bit value and is accessed with a 4-bit address.

Table 1: List of I/O pins for a total of 40 pins

Function ( <i>Name</i> )	I/O type	Bus Width	Description
Power ( <i>vdd</i> )	Input/Output	1 (x4)	Provides power
Ground ( <i>gnd</i> )	Input/Output	1 (x5)	Ground
Reset ( <i>reset</i> )	Input	1	Resets the board to empty
Clock ( <i>ph1</i> , <i>ph2</i> )	Input	2	Two phase clock
First Player ( <i>isPlayer1Start</i> )	Input	1	Determines which player plays "X" or "O"
Write Confirmation ( <i>playerWrite</i> )	Input	1	Confirms the current player input
Input Position ( <i>playerInput</i> )	Input	4	Indicates which position on the game board the player played
Winner ( <i>winner</i> )	Output	2	Displays 11 if player 1 wins, 10 if player 2 wins, and 01 if there is a draw
Game State ( <i>gameState</i> )	Output	2	Indicates which player's turn it is to play or if the game is done
Game Board ( <i>gBoard</i> )	Output	18	Displays the current state of the game

### 3 Floorplan

Figure 4 shows the chip floorplan for the tic-tac-toe chip including the pad frame. The top-level blocks are the *game controller*, the *datapath* and the *memory*. A wiring channel is located between the controllers and datapath and between the datapath and memory to provide room to route control signals to the datapath. The *pad frame* includes 40 I/O pads, which are wired to the pins on the chip package. As listed in table 1, there are 31 pads used for signal; the remainder are  $V_{DD}$  and GND. Figure 4 shows the dimensions of each block on the chip and figure 3 shows the pinout diagram indicating names and pin numbers for each pin.

The floorplan is drawn to estimated scale size in fig. 4. The chip is designed in a  $0.6 \mu m$  process on  $1.5 \times 1.5$  mm die so the die is  $5000 \lambda$  on a side. Each pad is  $750\lambda \times 350\lambda$  [1].

#### 3.1 Slice Plans

Both the memory array and the win logic were laid out with 8 words of 2-bit length. A slice plan for the datapath is shown in fig. 5.

### 4 Verification

All Verilog code successfully simulates using their respective testbenches and sets of testvectors. All schematics were also successfully tested against the same testbenches.

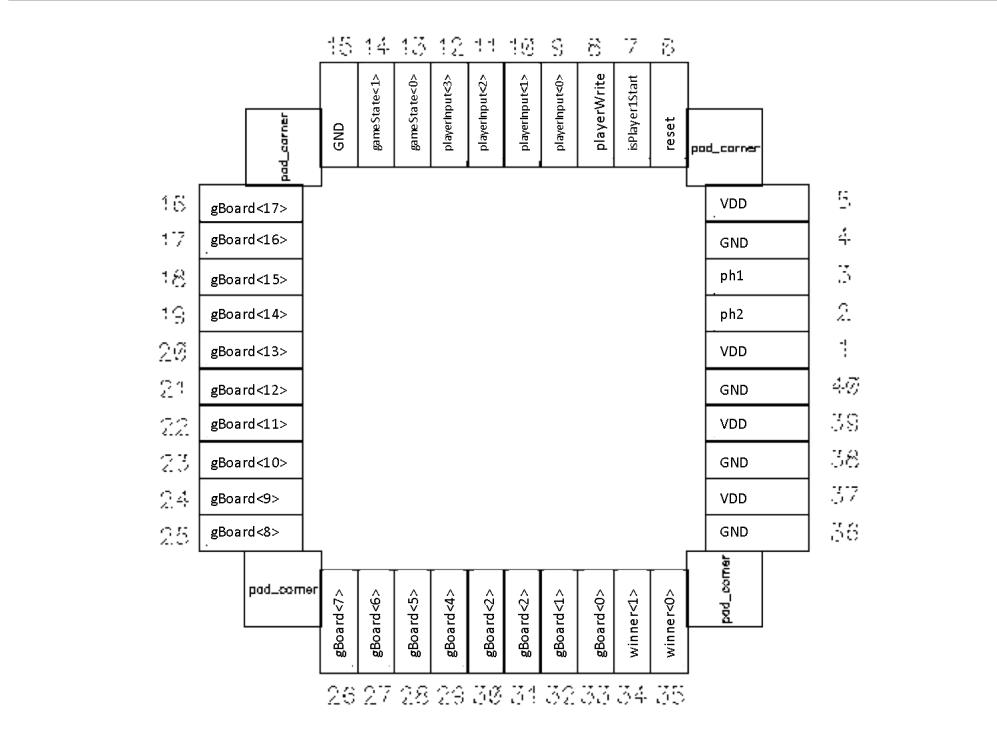


Figure 3: Tic-tac-toe pinout diagram

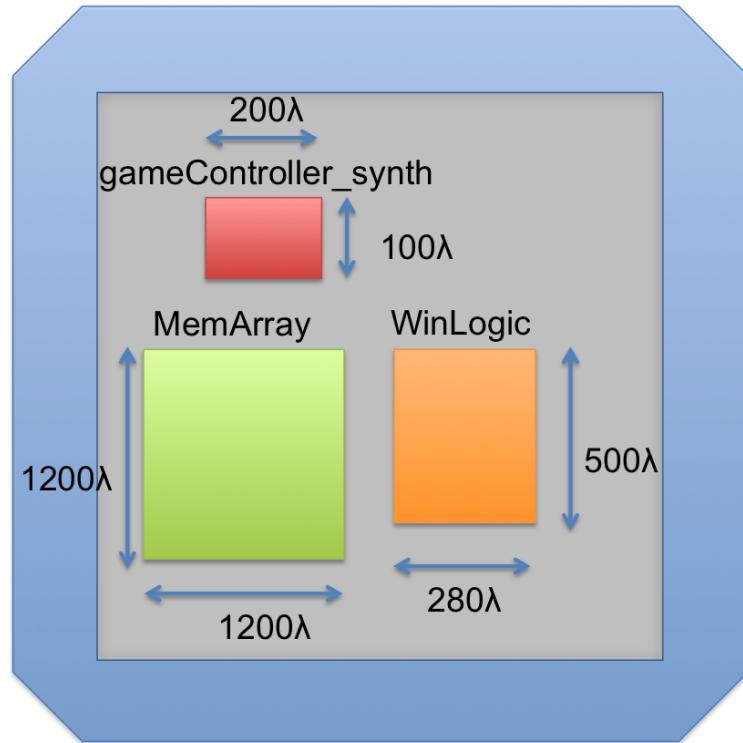


Figure 4: Tic-tac-toe floorplan

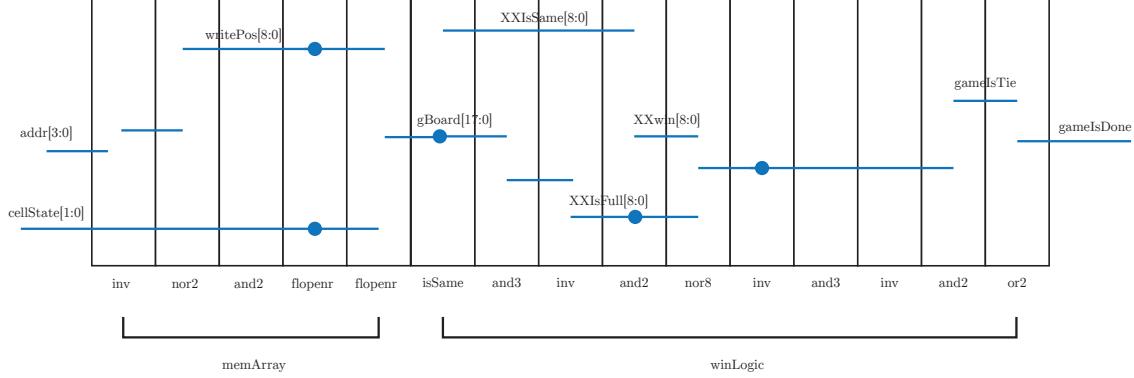


Figure 5: Slice plan for the datapath includes both the *memArray* and the *winLogic*

Every layout component of the system passed DRC and LVS. The exported GDS chip passes DRC with 144 errors related to optional rule 10.4 about spacing from the pad to unrelated metal.

#### 4.1 Postfabrication test plan

If the chip was to be fabricated, recommended would include:

1.  $V_{DD}$  and  $GND$  test by applying power to the chip.
2. inspecting the clock ( $ph1$  and  $ph2$ )
3. connecting switches to inputs and leds to outputs and run parts or all of the test vectors.

### 5 Summary

#### 5.1 Design Time

A summary of design time spent on each component of the project is shown in table 2.

Table 2: Table of design time for each component

Component	Time Spent (hrs)
Project Proposal	7
Verilog	30
Schematic	20
Layout	40
<b>Total Time spent:</b>	<b>97</b>

#### 5.2 File Locations

Location of the files for this project are listed in table 3. Some of these files can also be found in the appendices.

Table 3: File locations for each component

Item	Location
Verilog testbench	<a href="https://github.com/glegrain/Tic-tac-toe">https://github.com/glegrain/Tic-tac-toe</a>
Testvectors	<a href="https://github.com/glegrain/Tic-tac-toe">https://github.com/glegrain/Tic-tac-toe</a>
Synthesis Results	/home/glegrain/IC_CAD/cadence/texttt*.rep, *pow
GDS	/home/glegrain/IC_CAD/cadence/chip.gds
Cadence Libraries	/home/glegrain/IC_CAD/cadence
PDF of chip plot	<a href="https://github.com/glegrain/Tic-tac-toe/report/chip-layout.png">https://github.com/glegrain/Tic-tac-toe/report/chip-layout.png</a>
Final Report	<a href="https://github.com/glegrain/Tic-tac-toe/blob/master/report/main.pdf">https://github.com/glegrain/Tic-tac-toe/blob/master/report/main.pdf</a>

## References

- [1] David Harris & Sarah Harris, *Digital Design and Computer Architecture*, Morgan Kaufmann; 2nd edition, 2012.
- [2] Crowley, Kevin & Siegler, Robert S. *Flexible Strategy Use in Young Children's Tic-Tac-Toe*, Cognitive Science; Issue 4, Volume 2, 1993.
- [3] Max Korbel & Ian Jimenez *Simplified Checkers*, E158: Introduction to CMOS VLSI Design Lab Report, 2011.

# Appendices

## A Logic Design

A top level diagram in fig. 6 show the different blocks of the architecture.

The blocks are described in each of the sections below. The game controller is in charge of controlling which player is writing to the game board. Player 1 and player 2 enters the address they want to play at, which updates to the memory FF block when it is their respective turn and a write switch is activated. The datapath uses combinational logic determine the winning/losing status of the players.

### A.1 Memory Array

The *Memory array* is an array of 18-bit flip-flops (9 words of 2-bits) to store the state of the game board. The memory array incorporates a row decoder using the address (`addr[3:0]`) to activate one of the rows by asserting the wordline (`cellState[1:0]`). The decoder logic for the memory can be seen in figure 7.

The decoder was estimated to use 4 cells per “write” to enable each pairs of flip-flop. Thus, the decoder logic was estimated to be using an estimate of  $4 \times 9$  logic gates which rounds to a rough size estimate of  $1000\lambda \times 188\lambda$ . The flip-flop array height estimate was  $18 \times 100\lambda = 1800\lambda$  and the width to  $300\lambda$  wide . The total size estimate for the memory block was rounded up to  $2000\lambda \times 600\lambda$ . The actual size ended up being  $1250\lambda \times 1210\lambda$ .

### A.2 Win logic

Using only the game Board memory array, the win logic checks if a column, row or diagonal is full by looking at the MSB of each cell. If yes, then the logic checks if all the cells in the column, row or diagonal are all the same by looking at the LSB of each cell using a custom leaf cell called `isSame` described in sec. A.3. The win logic is also able to determine who is the winner is by checking if each row, column or diagonal has the same cell. Partial schematics for the win logic module can be found in fig. 8, fig. 9 and fig. 10.

### A.3 `isSame` leaf cell

CMOS logic is used to check if 3 bits are identical. This cell is used to determine if the cells are taken by the same agent. The CMOS schematic can be seen in fig. 11.

### A.4 FSM desgin: Game Controller

The *Game Controller* contains a finite state machine (FSM) to decide when each player has to play and whether or not the game is done. A state transition diagram for the game controller FSM is shown in figure 12.

## B Code, testbenches and test vectors

For clarity reasons, only the chip testbench and testvectors are provided here. All other test files can be found in the paths described in sec. 5.2.

```
..../chip_tb.sv
// Set delay unit to 1 ns and simulation precision to 0.1 ns (100 ps)
'timescale 1ns / 100ps

module chip_tb();
    logic      ph1, ph2, reset;
    logic      isPlayer1Start;
    logic      playerWrite;
    logic [3:0] playerInput;
```

```

logic [17:0] gBoard;
logic [1:0] gameState;
logic [1:0] winner;

logic [17:0] gBoardExpected;
logic [31:0] vectornum, errors;
logic [23:0] testvectors [10000:0];

// instantiate device under test
chip dut(.ph1, .ph2, .reset,
         .isPlayer1Start,
         .playerWrite,
         .playerInput,
         .gBoard,
         .gameState,
         .winner);

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

// tell the simulator to store the waveform into a file for inspection
// and start dumping all signal to the .vcd file
initial
begin
    $dumpfile("chip.vcd");
    $dumpvars;
end

// at start of test, load test vectors
initial
begin
    $readmemb("chip.tv", testvectors);
    vectornum=0; errors=0;
    reset=1; #18; reset=0;
end

// apply test vectors on rising edge of clk
always @(posedge ph2)
begin
    #1; {playerInput, playerWrite, isPlayer1Start, gBoardExpected} =
        testvectors [vectornum];
end

// Custom input signals
initial
begin
    isPlayer1Start = 0;

```

```

    end

    // check results on falling edge of clk
    always @(negedge ph2)
        if(~reset) begin // skip during reset
            if ((winner == 2'b01) | (winner == 2'b11)| (winner == 2'b10)) begin
                $display("Game is done: player 2'b%b wins", winner);
                $finish;
            end
            vectornum = vectornum + 1;
            if(testvectors[vectornum] == 24'bx) begin
                $display("%d tests completed with %d errors", vectornum, errors);
                $finish;
            end
            if (gBoard != gBoardExpected) begin // check result
                $display("Error: vectornum=%d", vectornum);
                $display("inputs: playerInput=%d playerWrite=%b", playerInput,
                         playerWrite);
                $display("outputs: gameBoard=%b (%b expected)", gBoard, gBoardExpected
                         );
                errors = errors + 1;
            end
        end
    end

```

```
endmodule
```

./chip.tv

0000_0_1_000000_000000_000000
0000_1_1_000000_000000_000000
0000_0_1_000000_000000_000001
0001_0_1_000000_000000_000001
0001_1_1_000000_000000_000001
0001_0_1_000000_000000_001101
0010_0_1_000000_000000_001101
0010_1_1_000000_000000_001101
0010_0_1_000000_000000_011101
0011_0_1_000000_000000_011101
0011_1_1_000000_000000_011101
0011_0_1_000000_000011_011101
0100_0_1_000000_000011_011101
0100_1_1_000000_000011_011101
0100_0_1_000000_000111_011101
0101_0_1_000000_000111_011101
0101_1_1_000000_000111_011101
0101_0_1_000000_110111_011101
0110_0_1_000000_110111_011101

```

0110_0_1_000000_110111_011101
0110_0_1_000001_110111_011101

0111_0_1_000001_110111_011101
0111_1_1_000001_110111_011101
0111_0_1_000001_110111_011101

1000_0_1_000001_110111_011101
1000_1_1_000001_110111_011101
1000_0_1_000001_110111_011101

XXXX_X_X_XXXXXX_XXXXXX_XXXXXX

```

..../chip.sv

```

// Game Controller top module for tic-tac-toe project
// Written by Katherine Yang and Guillaume Legrain
// Written in: March 25, 2015
// player1:11, player2:10, tie:01, noWin:00
// cellState: empty:00, player1:11, player2:10
///////////////////////////////
module chip(input logic ph1, ph2, reset ,
            input logic isPlayer1Start ,
            input logic playerWrite ,
            input logic [3:0] playerInput ,
            output logic [17:0] gBoard ,
            output logic [2:0] gameState ,
            output logic [1:0] winner);

    logic [3:0] addr;
    logic [1:0] cellState;
    logic gameIsDone;

    gameController gameControllerFSM(.ph1, .ph2, .reset ,
                                    .isPlayer1Start ,
                                    .playerWrite ,
                                    .playerInput ,
                                    .gameIsDone ,
                                    .addr ,
                                    .cellState ,
                                    .gameState);

    memArray gameBoard (.ph1, .ph2, .reset , .addr , .cellState , .gBoard);

    winLogic winLogic1 (.gBoard , .gameIsDone , .winner);

endmodule

```

..../memarray.sv

```

// Memory array block for tic-tac-toe project
// Written by Katherine Yang and Guillaume Legrain
// Written in: March 23, 2015
// Last edited: March 27, 2015
// player1:11, player2:10, tie:01, noWin:00

```

```

// cellState: empty:00, player1:11, player2:10
///////////////////////////////
module memArray(input logic ph1, ph2, reset ,
               input logic [3:0] addr,
               input logic [1:0] cellState ,
               output logic [17:0] gBoard);

logic [8:0] writePos;
logic [17:0] prevGB;

// Write decoder
addr2writePos decoder(addr[3:0], writePos[8:0]);

// array of Flip-flops
// NOTE: cellState bit order is flip to have the LSB of the gBoard be the
// MSB
// of the cellState
fopenr #(2) cell0(ph1, ph2, reset , writePos[0] , {cellState[0],cellState
[1]}, prevGB[1:0]);
fopenr #(2) cell1(ph1, ph2, reset , writePos[1] , {cellState[0],cellState
[1]}, prevGB[3:2]);
fopenr #(2) cell2(ph1, ph2, reset , writePos[2] , {cellState[0],cellState
[1]}, prevGB[5:4]);
fopenr #(2) cell3(ph1, ph2, reset , writePos[3] , {cellState[0],cellState
[1]}, prevGB[7:6]);
fopenr #(2) cell4(ph1, ph2, reset , writePos[4] , {cellState[0],cellState
[1]}, prevGB[9:8]);
fopenr #(2) cell5(ph1, ph2, reset , writePos[5] , {cellState[0],cellState
[1]}, prevGB[11:10]);
fopenr #(2) cell6(ph1, ph2, reset , writePos[6] , {cellState[0],cellState
[1]}, prevGB[13:12]);
fopenr #(2) cell7(ph1, ph2, reset , writePos[7] , {cellState[0],cellState
[1]}, prevGB[15:14]);
fopenr #(2) cell8(ph1, ph2, reset , writePos[8] , {cellState[0],cellState
[1]}, prevGB[17:16]);

assign gBoard = prevGB;

endmodule

// Write decoder logic
module addr2writePos(input logic [3:0] addr,
                      output logic [8:0] writePos);
always_comb
begin

```

```

    case (addr)
        4'b0000: writePos = 9'b000000001; // upperleft cell is writePos[0]
        4'b0001: writePos = 9'b000000010;
        4'b0010: writePos = 9'b000000100;
        4'b0011: writePos = 9'b000001000;
        4'b0100: writePos = 9'b000010000;
        4'b0101: writePos = 9'b000100000;
        4'b0110: writePos = 9'b001000000;
        4'b0111: writePos = 9'b010000000;
        4'b1000: writePos = 9'b100000000; // lowerleft cell
        4'b1111: writePos = 9'b000000000; //bad case
    default: writePos = 9'b000000000; // else, don't write
    endcase
end
endmodule

```

..../winLogic.sv

```

// Memory array block for tic-tac-toe project
// Written by Katherine Yang and Guillaume Legrain
// Written in: March 21, 2015
// Last edited: March 22, 2015
// player1:11, player2:10, tie:01, noWin:00
// cellState: empty:00, player1:11, player2:10
///////////////////////////////
module winLogic(input logic[17:0] gBoard,
                output logic gameIsDone,
                output logic[1:0] winner); // player1: 11, player2: 10, tie:
                01, noWin: 00
logic col0IsFull, col1IsFull, col2IsFull;
logic col0IsSame, col1IsSame, col2IsSame;
logic row0IsFull, row1IsFull, row2IsFull;
logic row0IsSame, row1IsSame, row2IsSame;
logic diag0IsFull, diag1IsFull;
logic diag0IsSame, diag1IsSame;
logic gameIsTie, thereIsAWin;

// each cell is represented by two bits in gBoard
// The MSB tells if the cell has been played or not
// The LSB tells
// empty: 00
// player1: 11
// player2: 10

// Check the LSB of each cell to see if they are the same
isSame col0IsSameMod(gBoard[1], gBoard[7], gBoard[13], col0IsSame);
isSame col1IsSameMod(gBoard[3], gBoard[9], gBoard[15], col1IsSame);
isSame col2IsSameMod(gBoard[5], gBoard[11], gBoard[17], col2IsSame);

isSame row0IsSameMod(gBoard[1], gBoard[3], gBoard[5], row0IsSame);
isSame row1IsSameMod(gBoard[7], gBoard[9], gBoard[11], row1IsSame);
isSame row2IsSameMod(gBoard[13], gBoard[15], gBoard[17], row2IsSame);

isSame diag0IsSameMod(gBoard[1], gBoard[9], gBoard[17], diag0IsSame);
isSame diag1IsSameMod(gBoard[5], gBoard[9], gBoard[13], diag1IsSame);

```

```

always_comb begin
    // Check the MSB of the cell to see if it has been written
    col0IsFull = (gBoard[0] & gBoard[6] & gBoard[12]);
    col1IsFull = gBoard[2] & gBoard[8] & gBoard[14];
    col2IsFull = gBoard[4] & gBoard[10] & gBoard[16];

    row0IsFull = gBoard[0] & gBoard[2] & gBoard[4];
    row1IsFull = gBoard[6] & gBoard[8] & gBoard[10];
    row2IsFull = gBoard[12] & gBoard[14] & gBoard[16];

    diag0IsFull = gBoard[0] & gBoard[8] & gBoard[16];
    diag1IsFull = gBoard[4] & gBoard[8] & gBoard[12];

    // If all columns are filled , there is no more free space.
    gameIsTie = col0IsFull & col1IsFull & col2IsFull;

    // Check to see if somebody made a line
    thereIsAWin = (col0IsFull & col0IsSame) | (col1IsFull & col1IsSame) | (
        col2IsFull & col2IsSame) |
        (row0IsFull & row0IsSame) | (row1IsFull & row1IsSame) | (
            row2IsFull & row2IsSame) |
            (diag0IsFull & diag0IsSame) | (diag1IsFull & diag1IsSame);

    // Game is done if a player wins or the board is full.
    gameIsDone = thereIsAWin | gameIsTie;

    // winner logic
    if      (col0IsFull & col0IsSame) winner <= {gBoard[0], gBoard[1]};
    else if (col1IsFull & col1IsSame) winner <= {gBoard[2], gBoard[3]};
    else if (col2IsFull & col2IsSame) winner <= {gBoard[4], gBoard[5]};
    else if (row0IsFull & row0IsSame) winner <= {gBoard[0], gBoard[1]};
    else if (row1IsFull & row1IsSame) winner <= {gBoard[6], gBoard[7]};
    else if (row2IsFull & row2IsSame) winner <= {gBoard[12], gBoard[13]};
    else if (diag0IsFull & diag0IsSame) winner <= {gBoard[0], gBoard[1]};
    else if (diag1IsFull & diag1IsSame) winner <= {gBoard[4], gBoard[5]};
    else if (gameIsTie) winner <= 2'b01;
    else winner <= 2'b00;
end
endmodule

```

..../gameController.sv

```

// Game Controller FSM block for tic-tac-toe project
// Written by Katherine Yang and Guillaume Legrain
// Written in: March 23, 2015
// Last edited: March 25, 2015
// player1:11, player2:10, tie:01, noWin:00
// cellState: empty:00, player1:11, player2:10
///////////////////////////////
// Cell states constants
// 0 is the human player , X is the AI

```

```

typedef enum logic [1:0] {EMPTY = 2'b00, WRITE_O = 2'b11, WRITE_X = 2'b10}
cellStateType;

// states
typedef enum logic [1:0] {START, PLAYER1, PLAYER2, END} statetype;

module gameController(input logic ph1, ph2, reset,
                      input logic isPlayer1Start,
                      input logic playerWrite,
                      input logic [3:0] playerInput, // cell address to
                        play. the cell state is based on the FSM state
                      input logic gameIsDone,
                      output logic [3:0] addr, // outputs the user input,
                        otherwise output 4'b1111
                      output cellStateType cellState, // outputs the user
                        number
                      output statetype gameState); //outputs state type

statetype state;
assign gameState = state;
// control FSM
statelogic statelog(.ph1, .ph2, .reset,
                     .isPlayer1Start, .gameIsDone, .playerWrite, .state);
outputlogic outputlog(.state, .reset, .playerWrite, .playerInput, .addr, .
cellState);

endmodule

module statelogic(input logic ph1, ph2, reset,
                  input logic isPlayer1Start,
                  input logic gameIsDone,
                  input logic playerWrite,
                  output statetype state);

statetype nextstate;
logic [1:0] ns, state_logic;

// resetable state register with initial value of START
mux2 #(2) resetmux(nextstate, START, reset, ns);
flop #(2) stateregister(ph1, ph2, ns, state_logic);
assign state = statetype'(state_logic);

// next state logic
always_comb
begin
  case (state) // Note: The game controller stays at the same state for
    one cycle after reset is released?
    START: nextstate = (isPlayer1Start) ? PLAYER1 : PLAYER2;
    PLAYER1: if (gameIsDone & (~reset)) nextstate = END;
              else if (~reset) nextstate = (playerWrite) ? PLAYER2 :
                PLAYER1;
              else nextstate = START;
    PLAYER2: if (gameIsDone & (~reset)) nextstate = END;

```

```

        else if (~reset) nextstate = (playerWrite) ? PLAYER1 :
                                PLAYER2;
        else nextstate = START;
    END:      nextstate = (reset) ? START : END;
    default: nextstate = START;
    endcase
end
endmodule

////////////////////////////////////////////////////////////////
module outputlogic(input statetype state ,
                   input logic reset ,
                   input logic playerWrite ,
                   input logic [3:0] playerInput ,
                   output logic [3:0] addr ,
                   output cellStateType cellState);

always_comb
begin
// NOTE: Assuming playerWrite is enable bit active on one clock cycle.
// A sperate module can be used to detect the playerWrite rising
edge.

// always send cellState to memory but will write only
// on valid addr (addr of 4'b1111 won't write anything)
addr = (playerWrite & (~reset)) ? playerInput : 4'b1111;
if (state == PLAYER1)
    cellState = WRITE_O;
else if (state == PLAYER2)
    cellState = WRITE_X;
else
    cellState = EMPTY;
end
endmodule

```

## C Schematics & layouts

### C.1 gameController

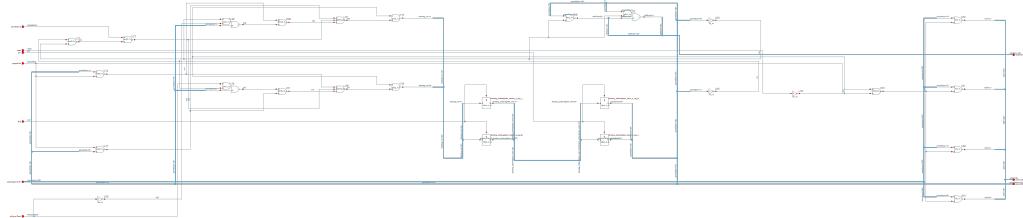


Figure 13: gameController synthesized schematic

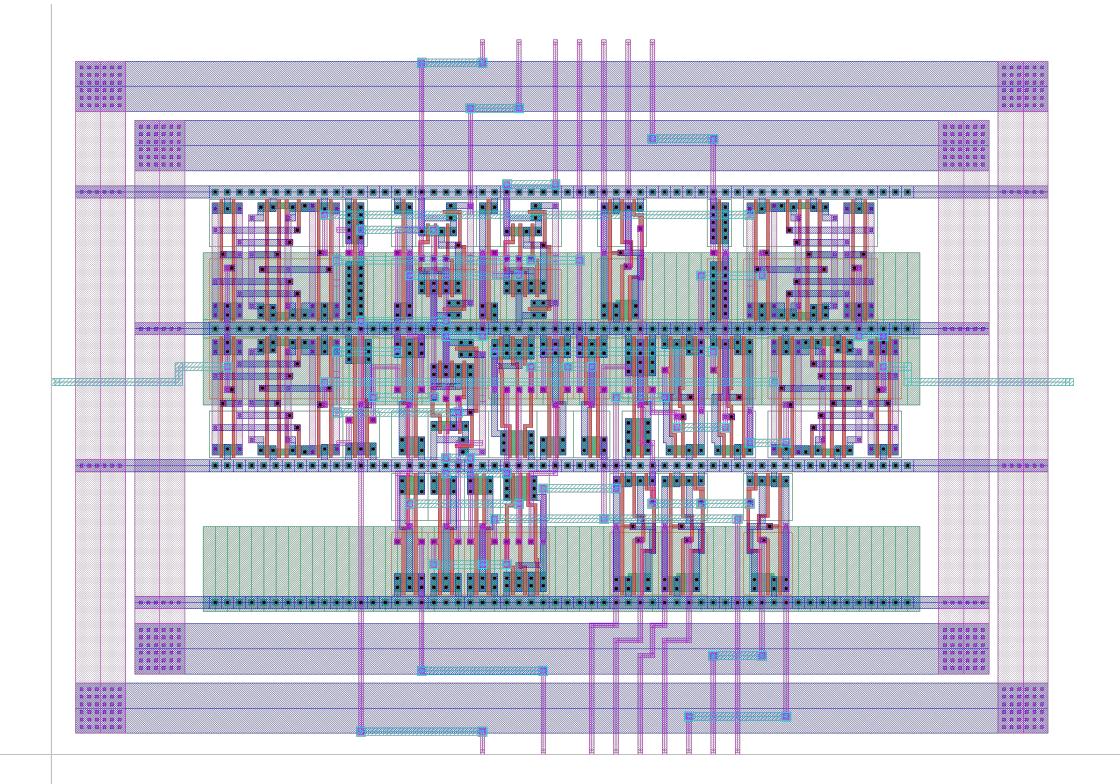


Figure 14: gameController synthetized layout

## C.2 winLogic

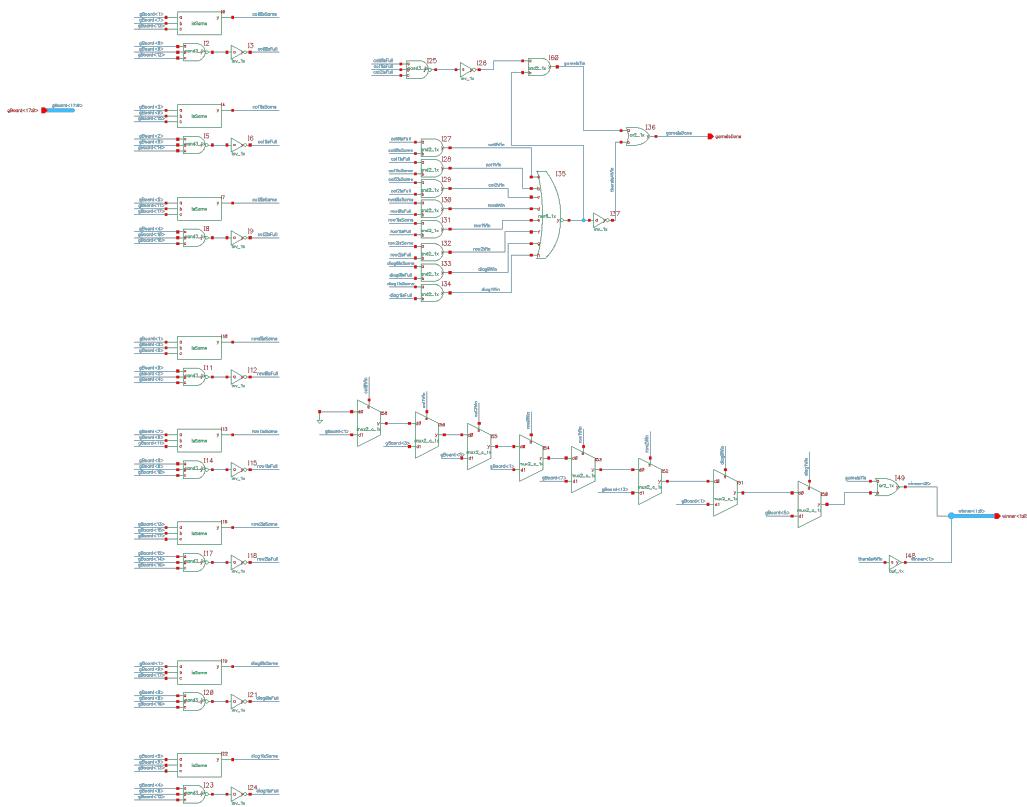


Figure 15: winLogic schematic

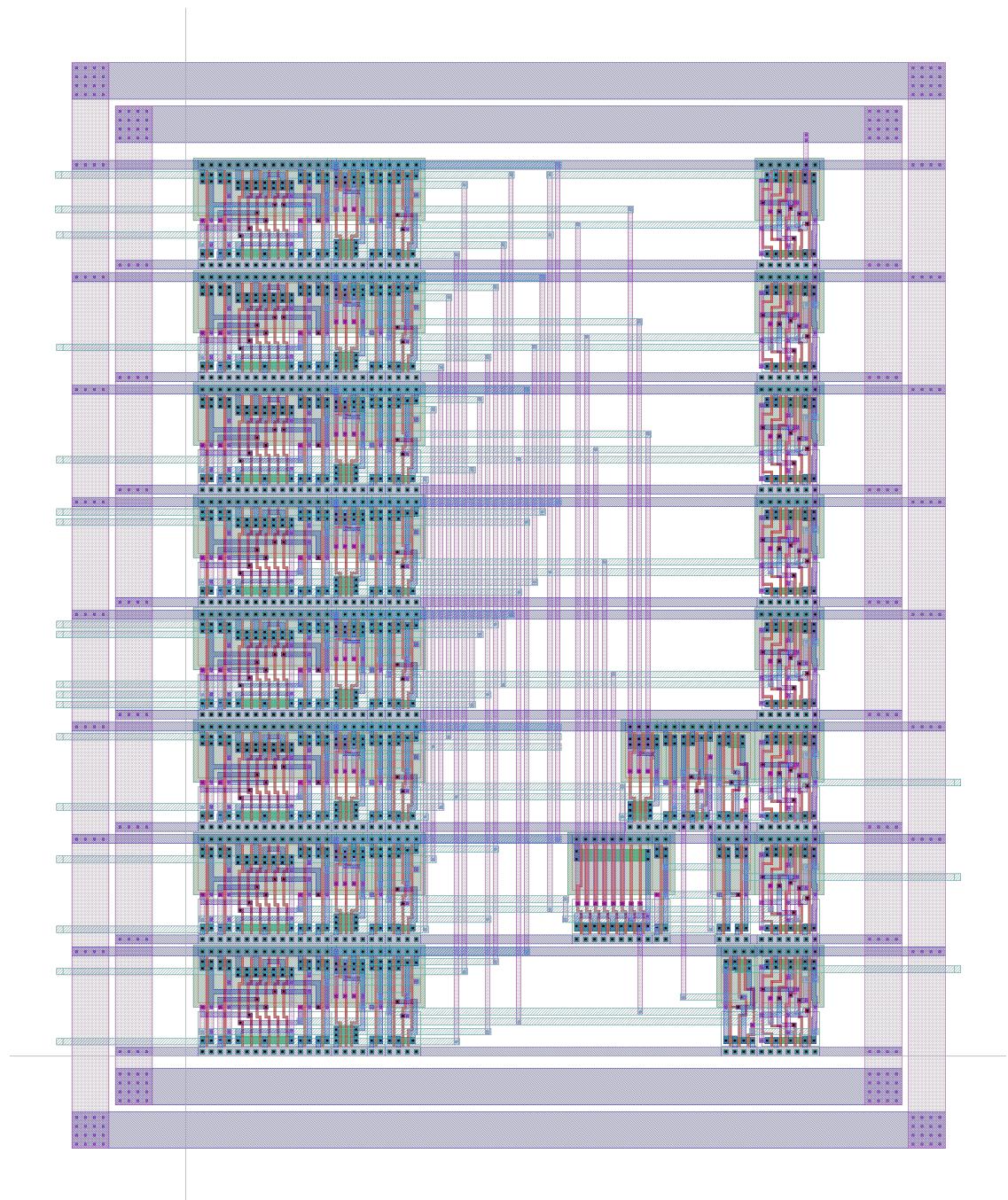


Figure 16: winLogic layout

### C.2.1 iSame leaf cell

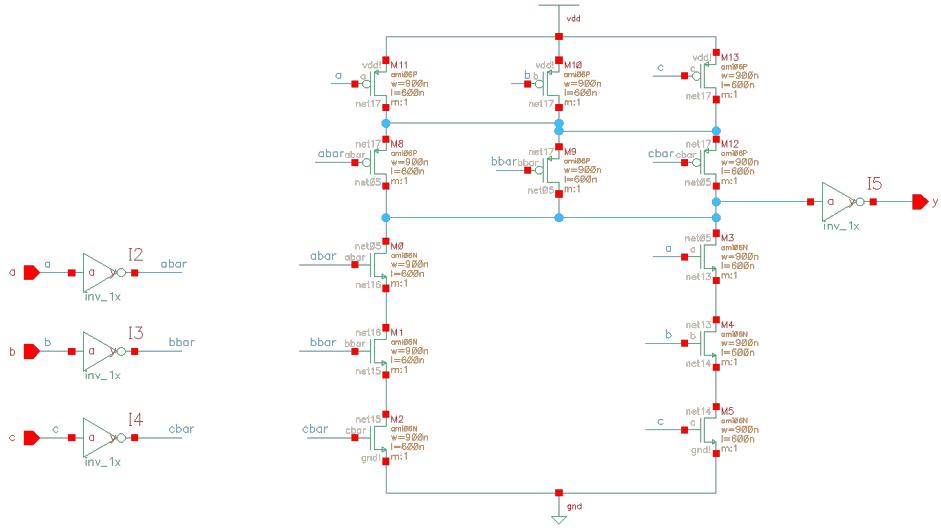


Figure 17: isSame leaf cell schematic

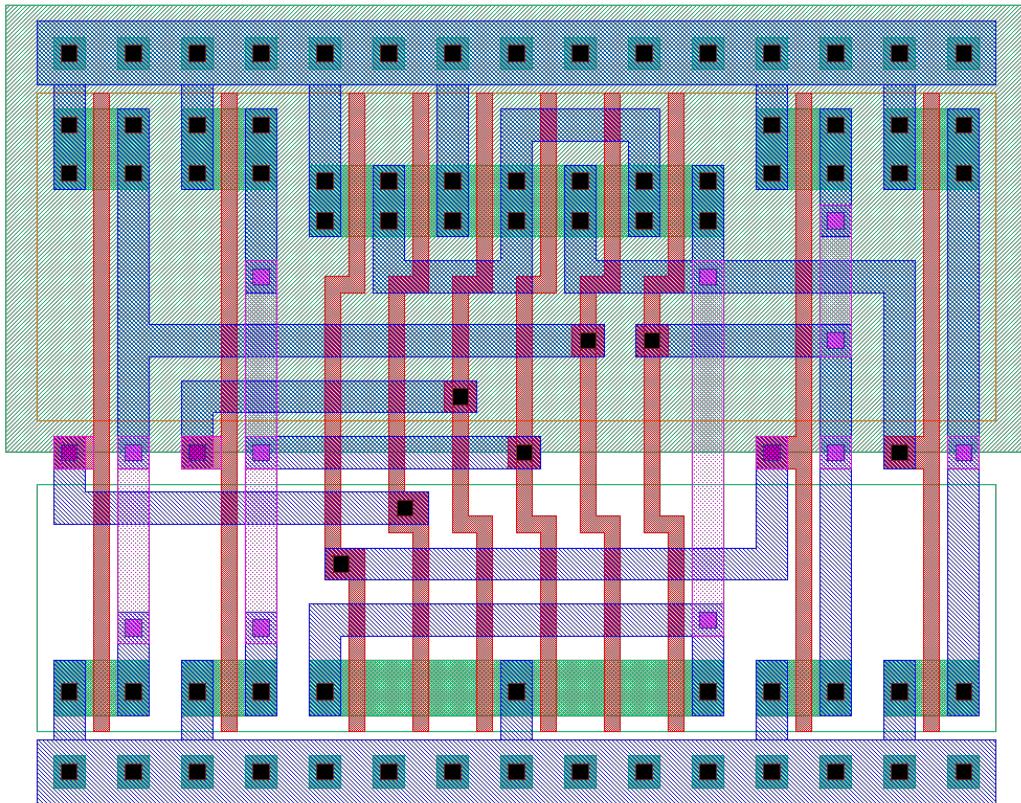


Figure 18: isSame leaf cell layout

### C.2.2 nor8\_1x leaf cell

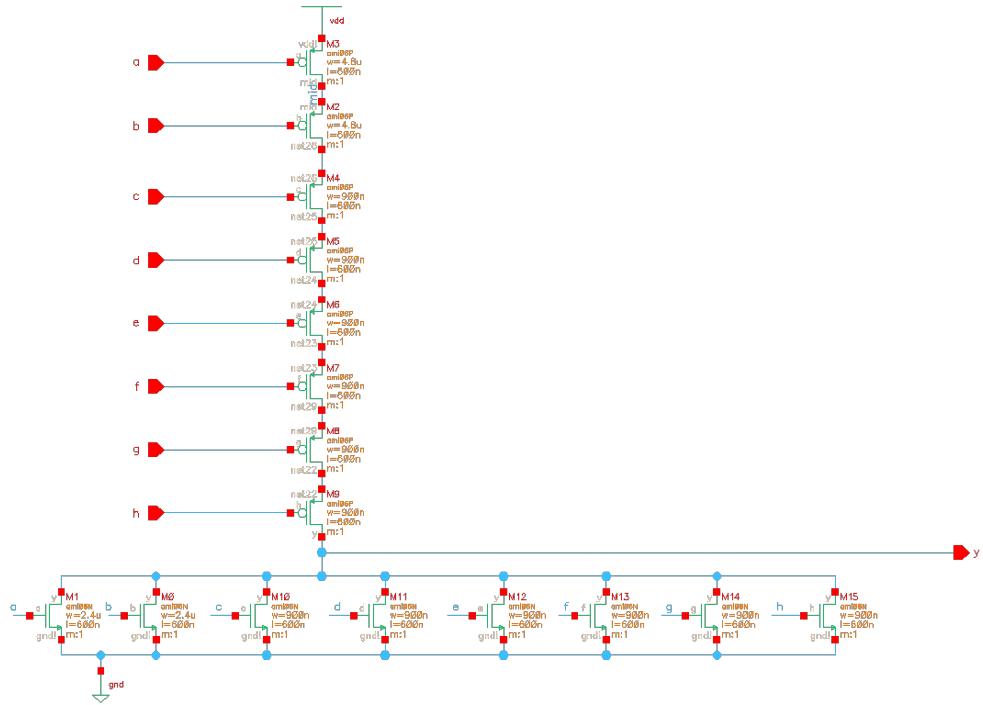


Figure 19: nor8\_1x cmos schematic

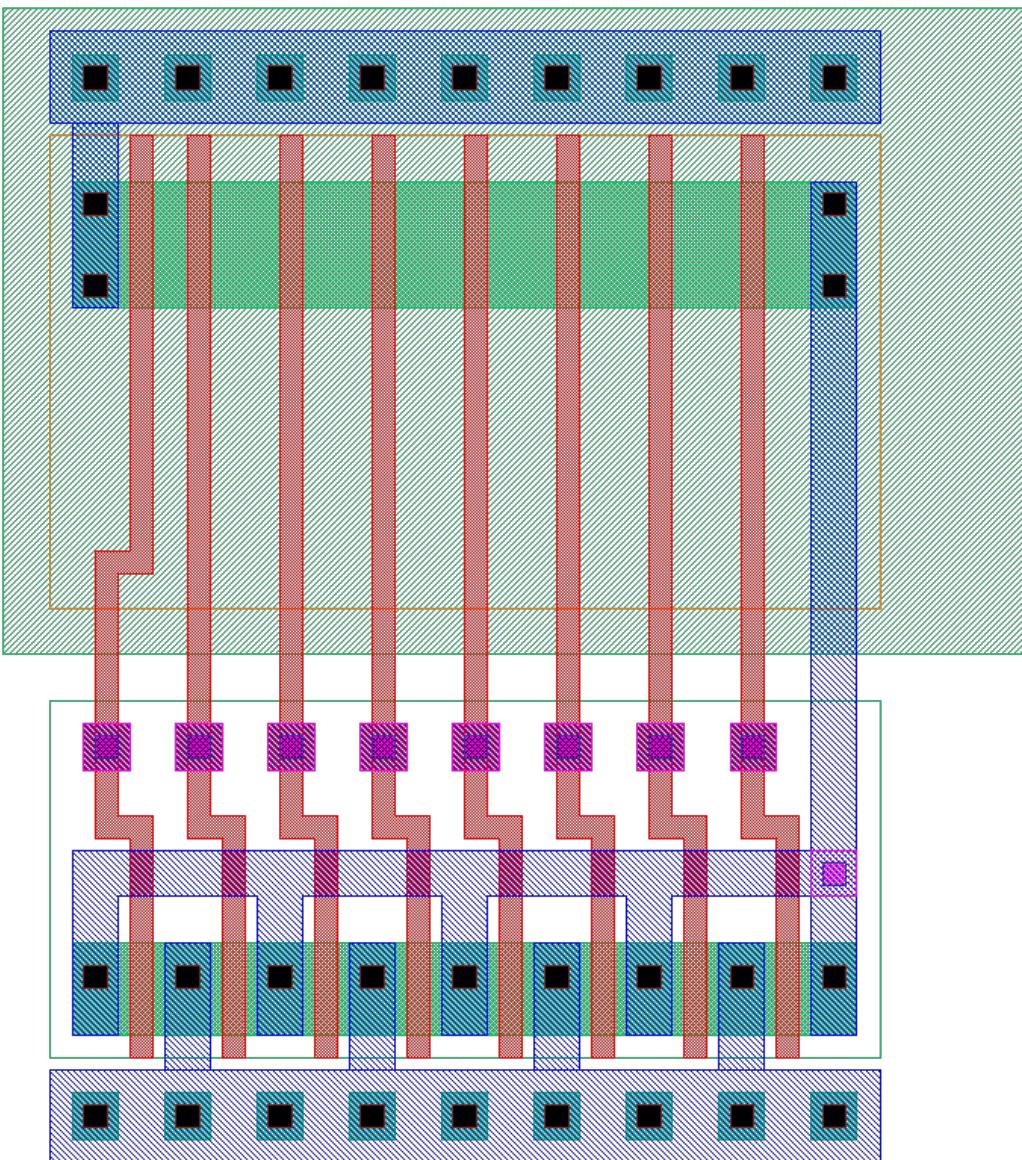


Figure 20: nor8\_1x layout

### C.3 memArray

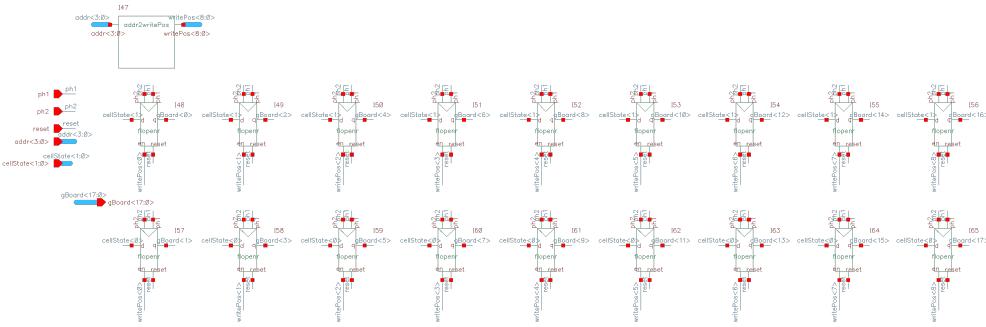


Figure 21: memArray schematic

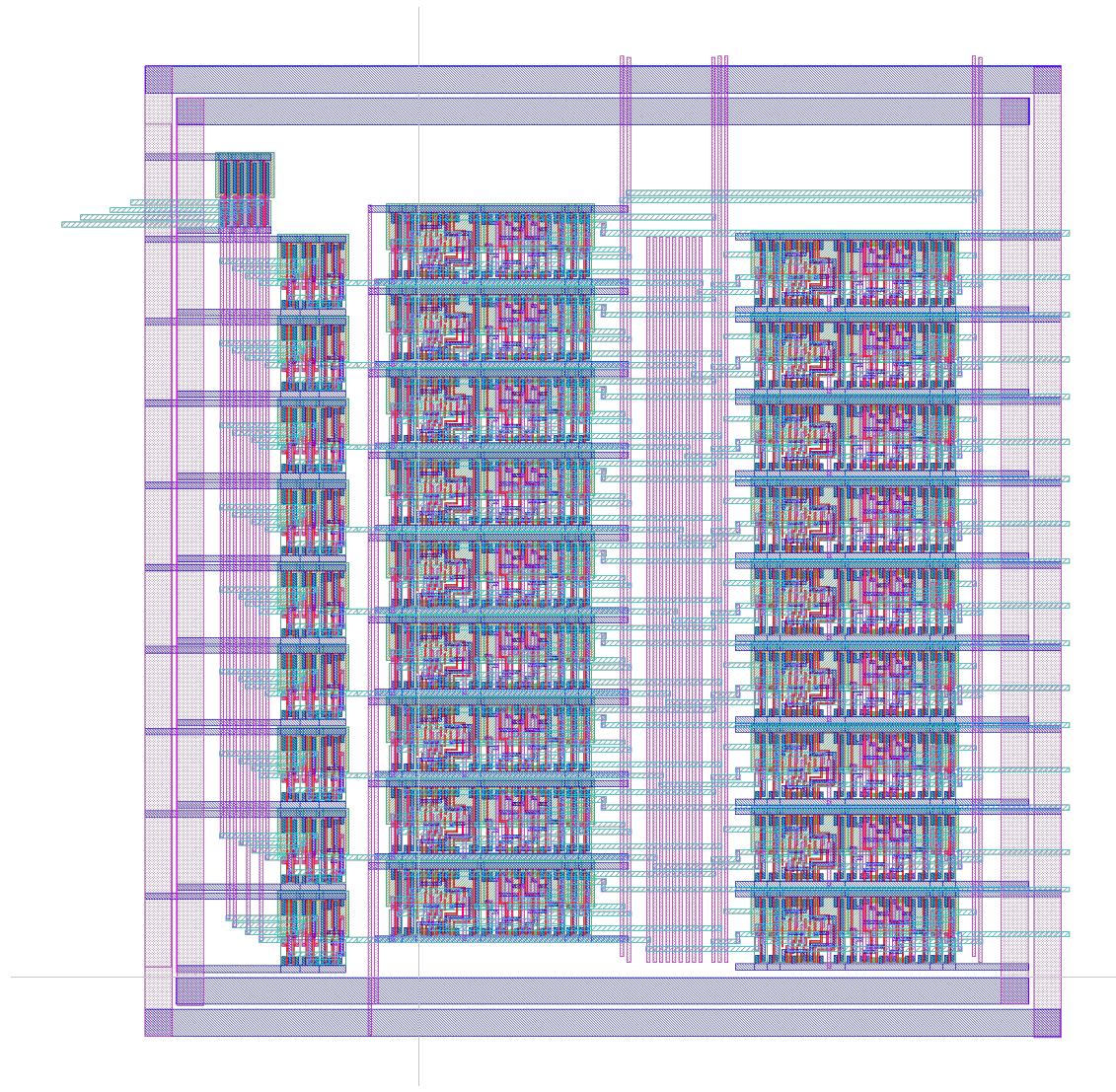


Figure 22: memArray layout



### C.3.1 addr2pos

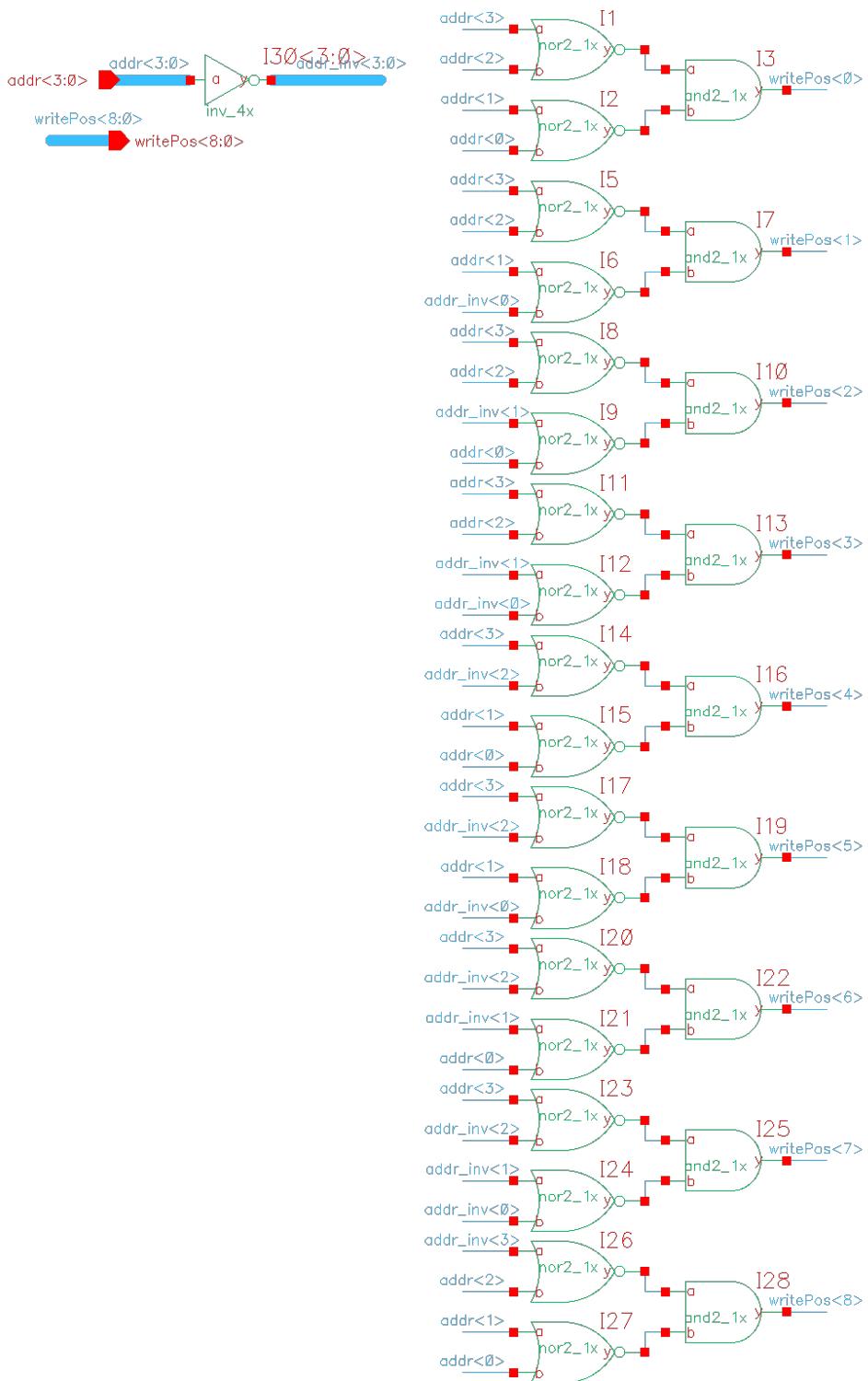


Figure 23: `addr2writePos` layout

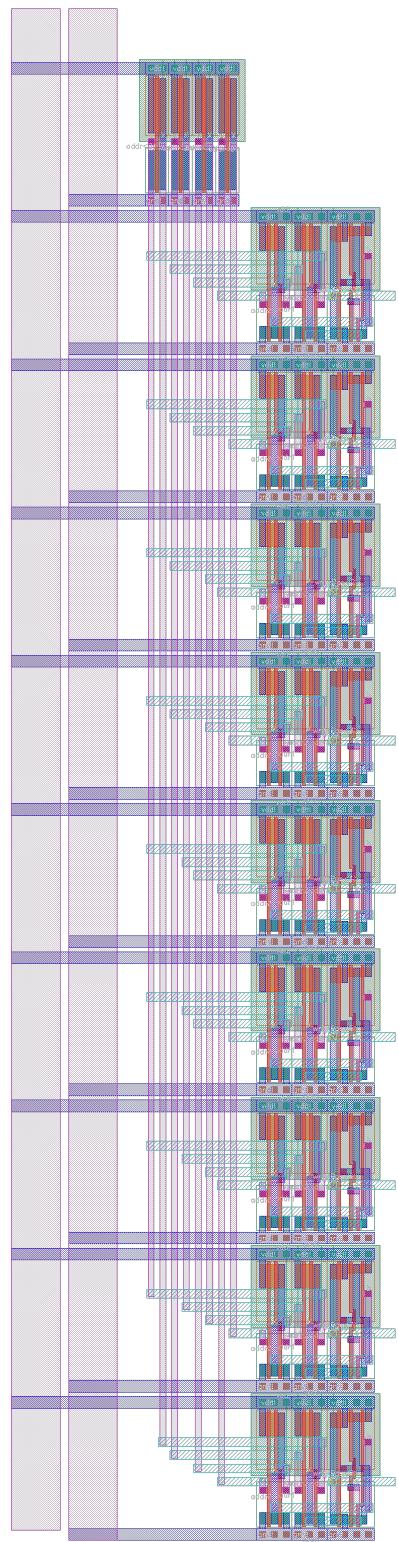


Figure 24: addr2writePos layout

### C.3.2 flopenr

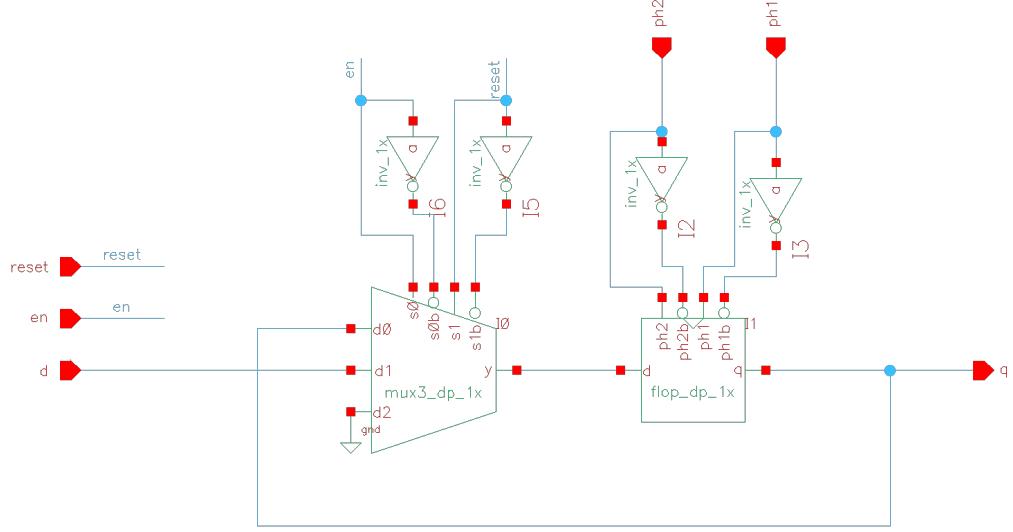


Figure 25: flopenr layout

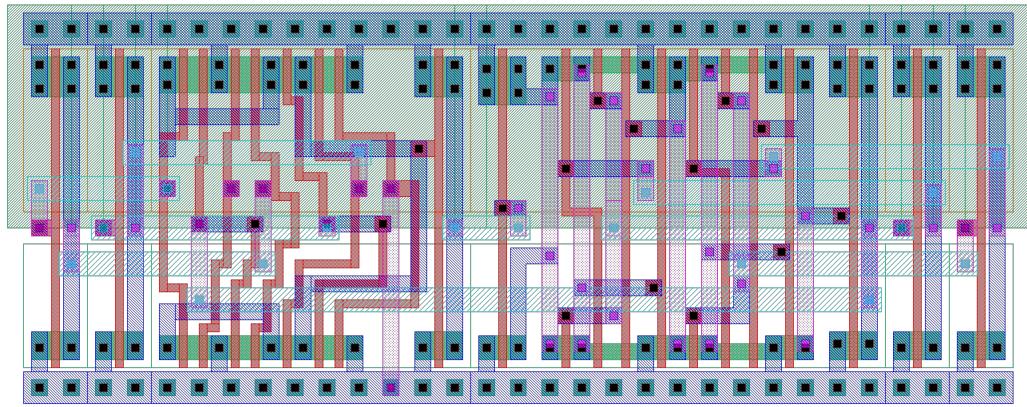


Figure 26: flopenr layout

## C.4 Padframe

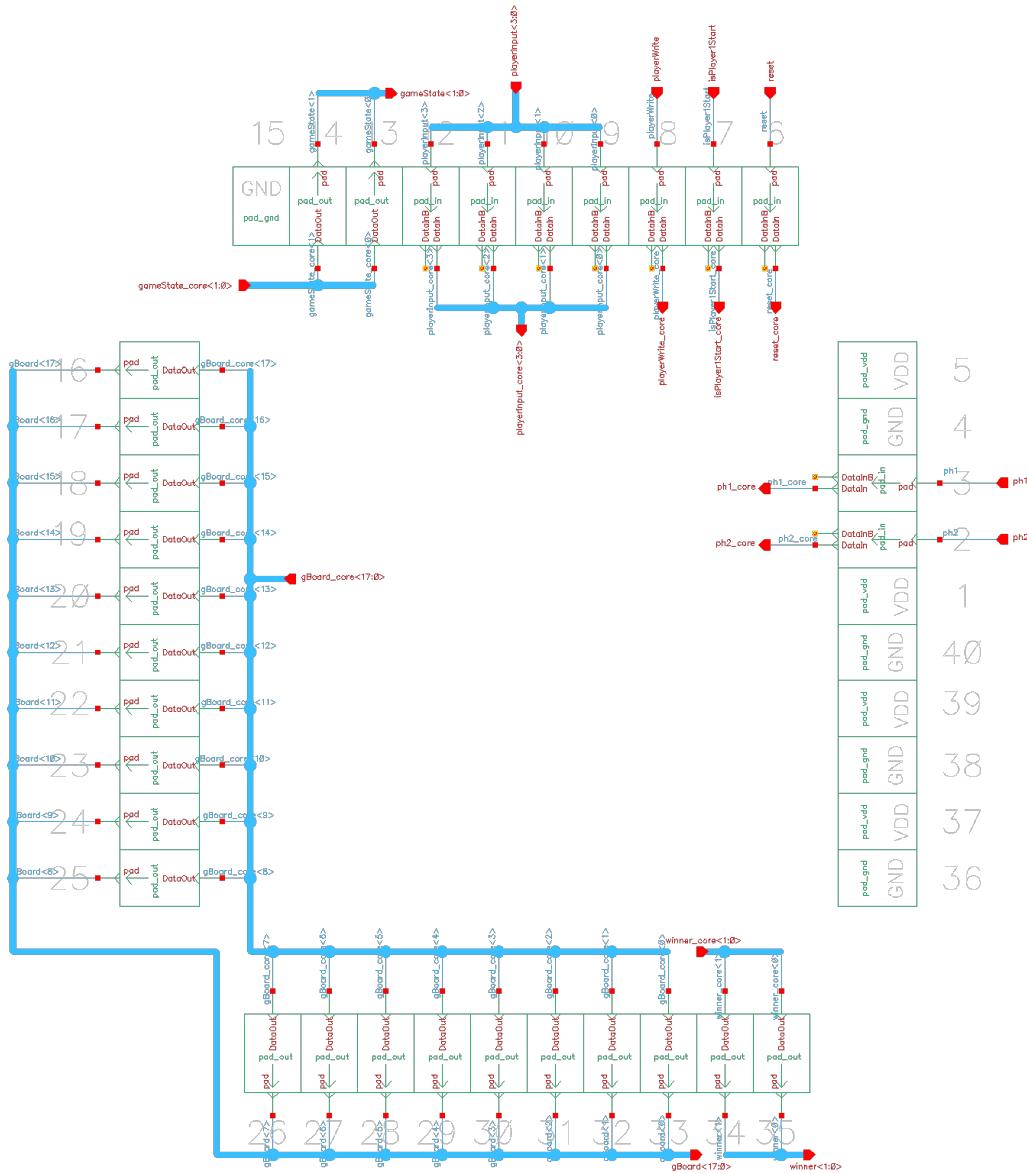


Figure 27: Padframe schematic

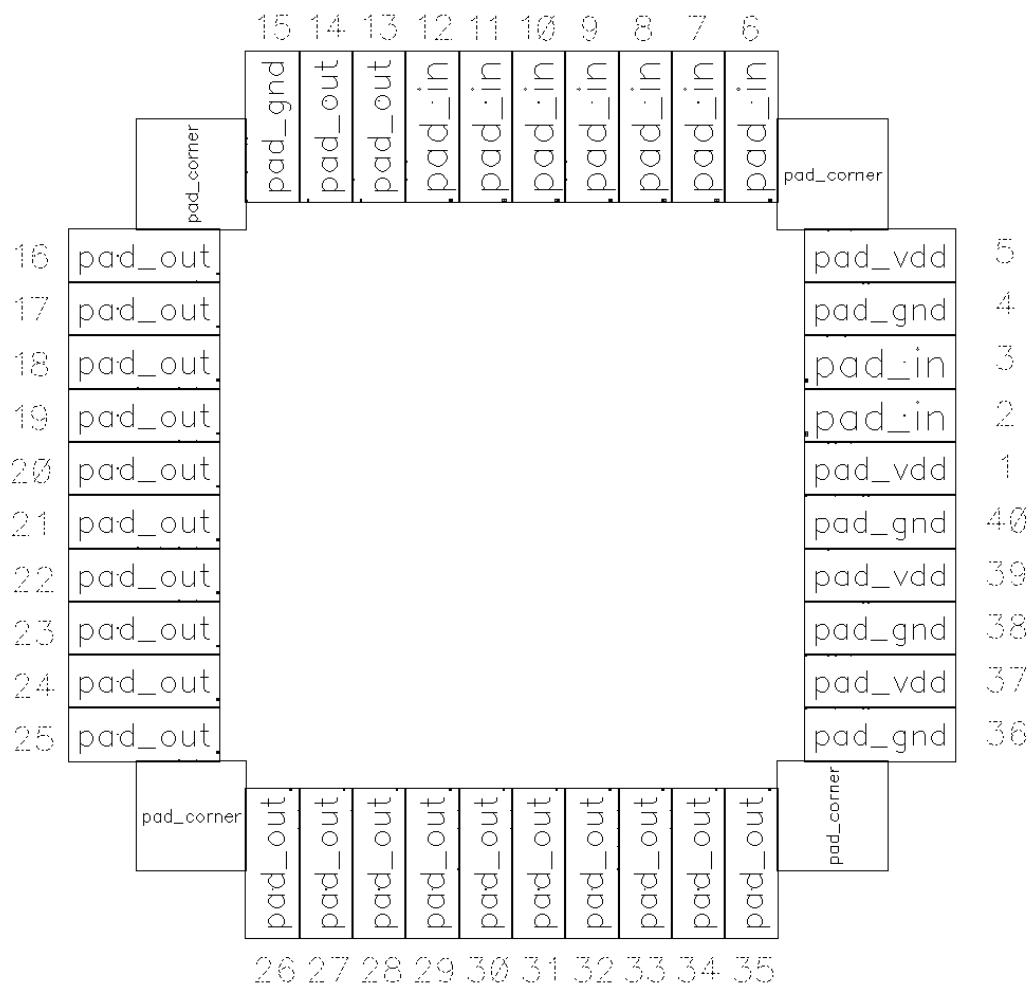


Figure 28: Padframe layout

## C.5 chip

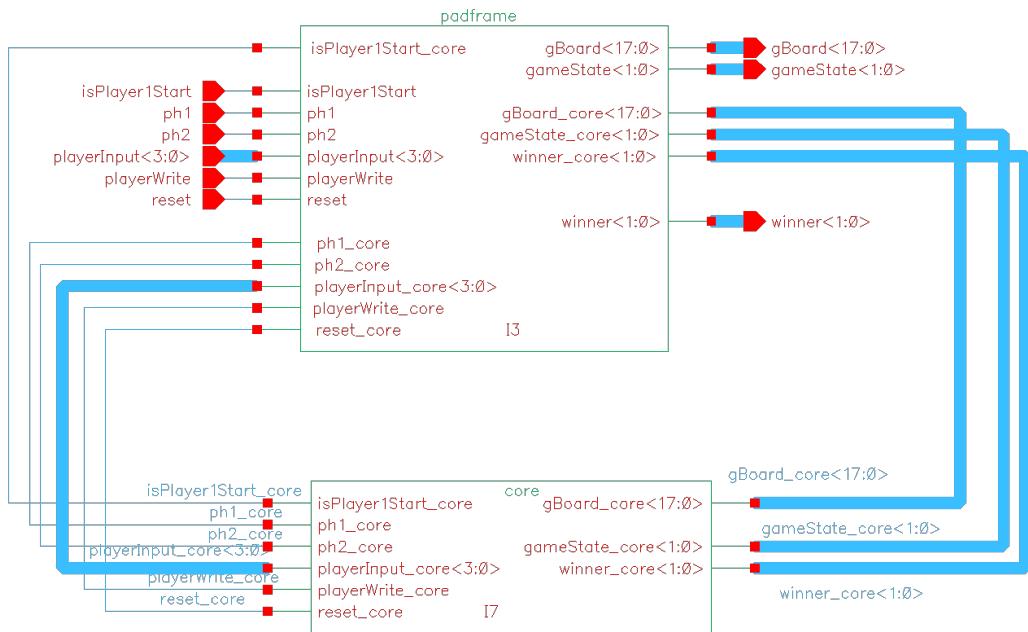


Figure 29: Chip schematic

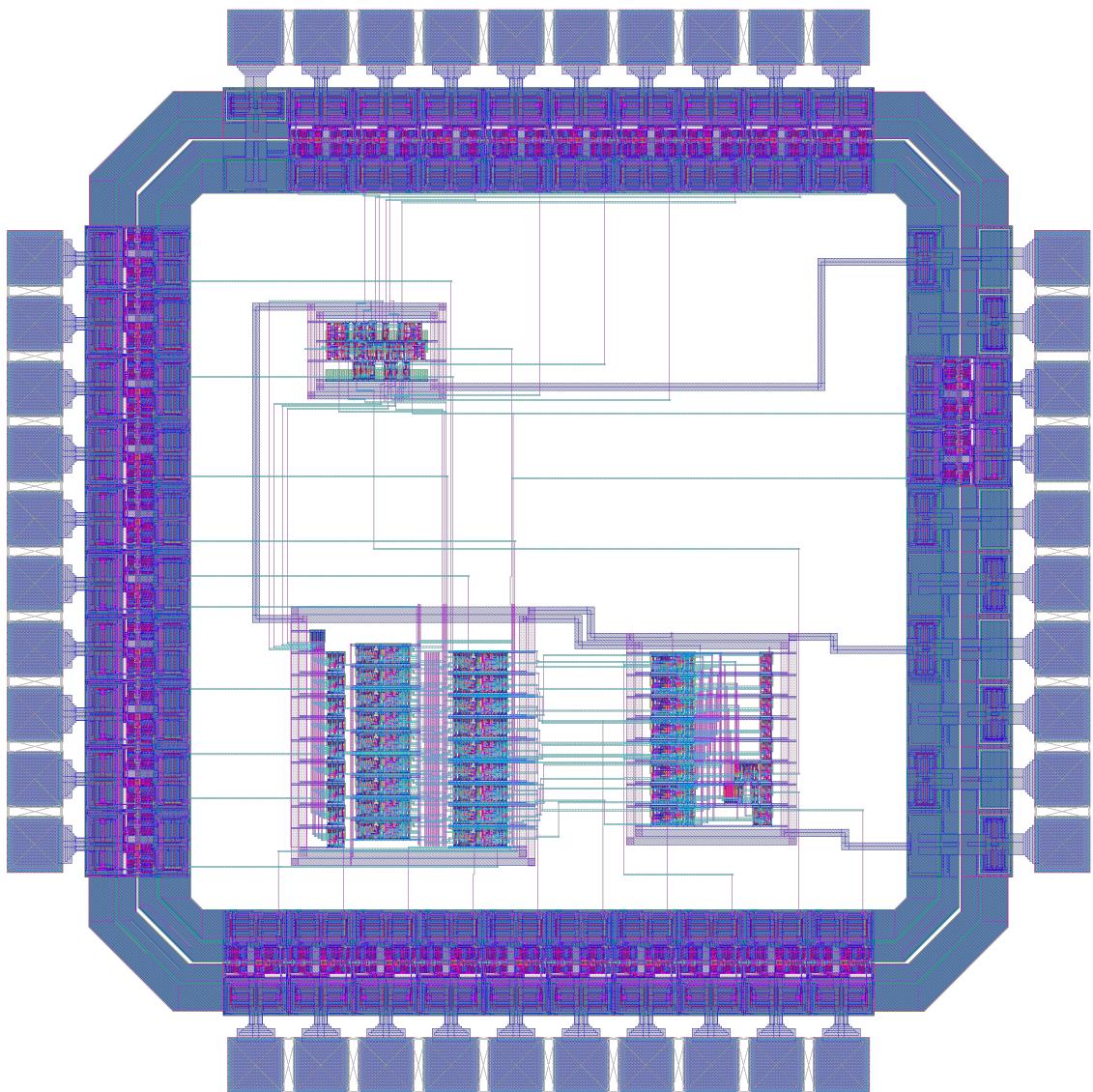


Figure 30: Chip layout

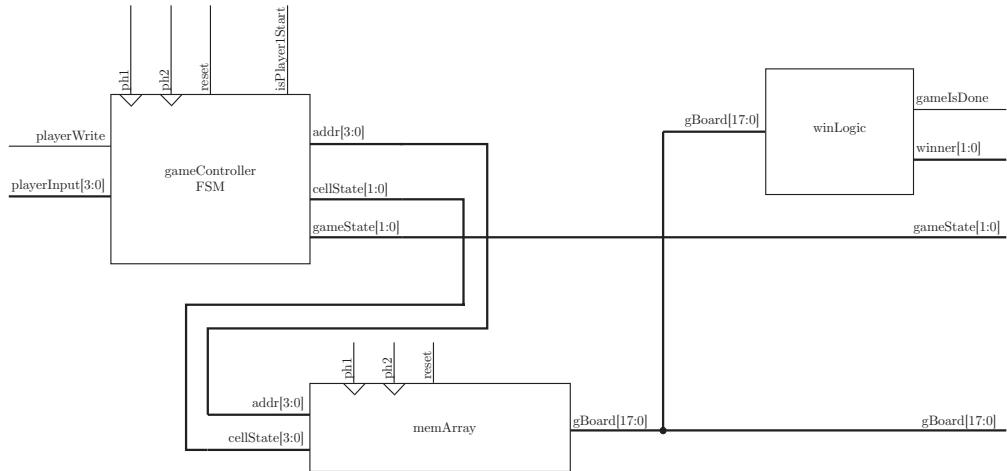


Figure 6: Top Level Block Diagram of the Tic-tac-toe architecture.

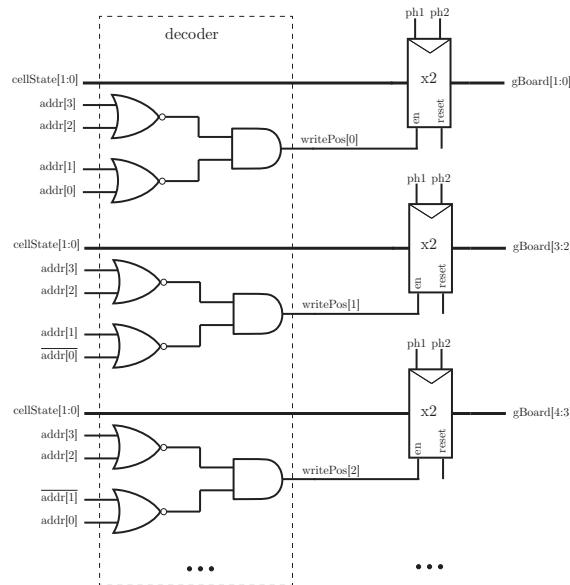


Figure 7: Memory array block made from 18 D-Flip-flops. The row decoder uses the address (**addr[3:0]**) to activate one of the rows by asserting the wordline (**cellState[1:0]**).

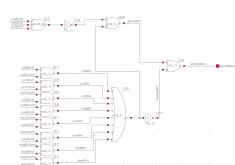


Figure 8: **winLogic** row, column diagonal check

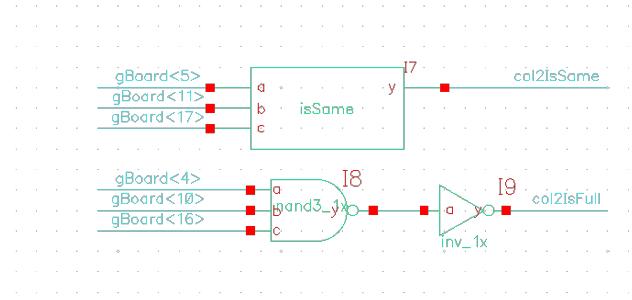


Figure 9: winLogic isSame and isFull check from game-board

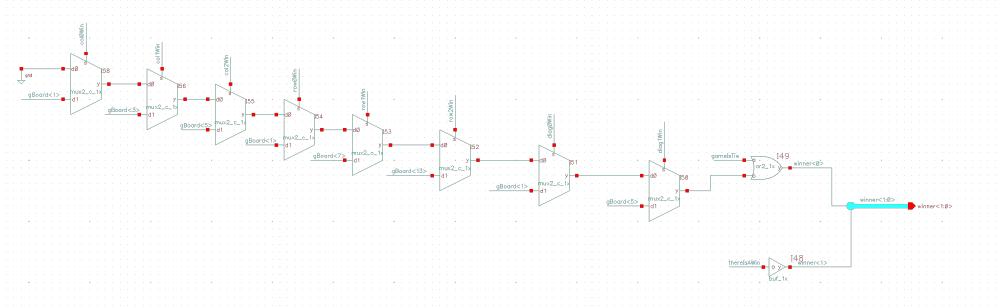


Figure 10: winLogic winner select block

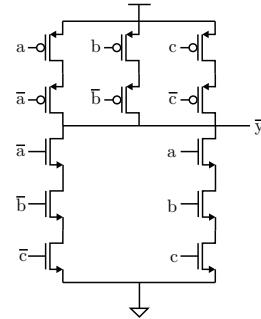


Figure 11: isSame leaf cell logic

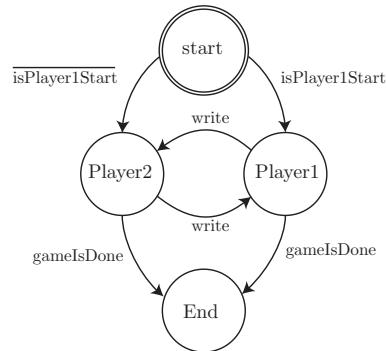


Figure 12: Game Controller Finite-State-Machine