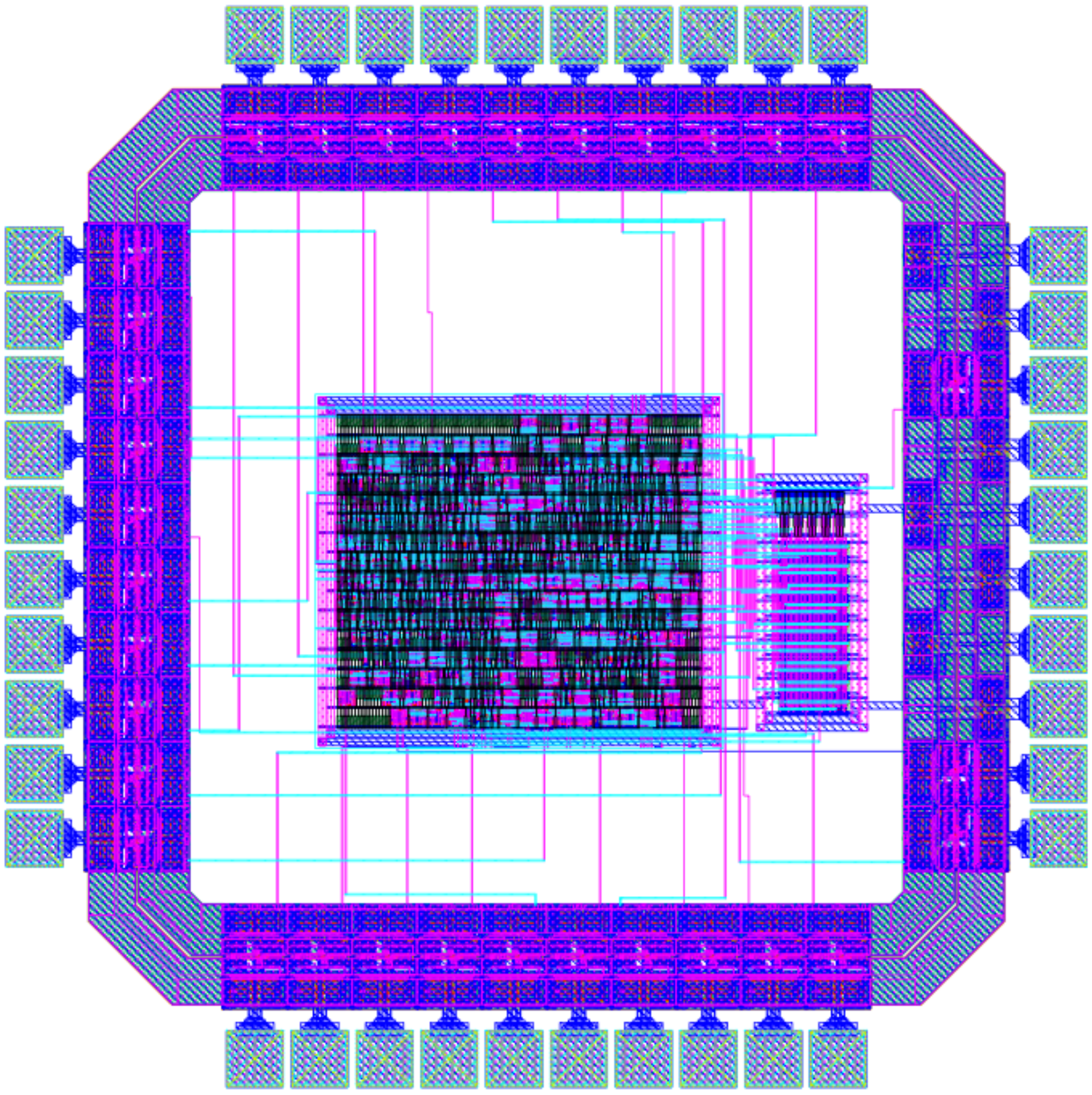
Chomp

E158 – CMOS VLSI Final Report

Kevin Aiach & Christian Jolivet - 04/18/2011



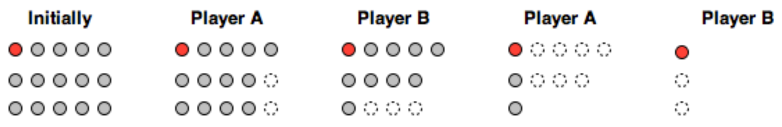
Introduction

The goal of this project is to implement the game Chomp in a 0.6 µm process and packaged in a 1.5 × 1.5 mm 40-pin MOSIS “TinyChip”.

What is Chomp?

Chomp is a 2-player game of strategy played on a rectangular board, made up of smaller blocks. The players take turns choosing one block to “eat” (remove from the board), together with those that are below it and to its right. The top left block is "poisoned" and the player who eats this loses.

Shown below is a sequence of moves on a 3x5 board. In this example, player A is forced to eat the poisoned block and loses.



Specifications

Pinout

Input	Output	Bi-directionnal
Clocks: ph1, ph2	Column button [8:0]	3*GND
Row button [4:0]	Column LEDs [8:0]	3*V_{DD}
reset	Row LEDs [4:0]	
	2*LED indicating player turn	

Theory of operation

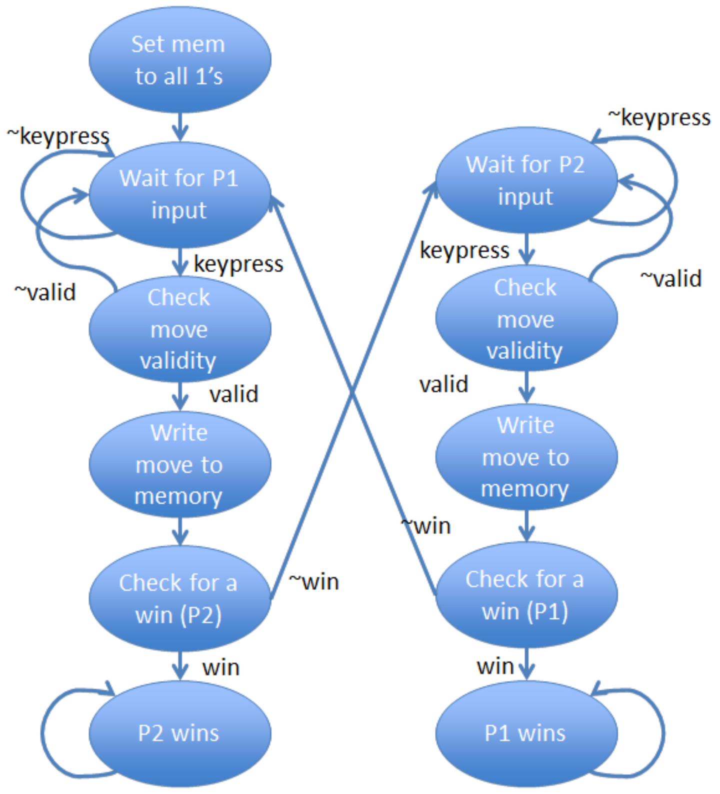
Our chip will be responsible for displaying the game board, indicating which player is taking a turn, accepting user input to determine which blocks should be removed, and indicating a winner at the end of the game. The board will be represented by a 5x9 array of LEDs. When LEDs are lit, blocks haven’t been eaten yet. A 5x9 array of buttons will be available for players to choose what block they want to eat. The 45 LEDs will be operated using time multiplexing, and the buttons will be interrogated using a scanning algorithm (similar to the key-scanner lab from E155). Two additional LEDs will indicate which player’s turn it is, and will flash to indicate a winner.

An invalid input is when the players try to eat a block that has already been eaten or if they try to press different buttons at the same time.

The time multiplexing will run at 100Hz at least so the players won’t see the LEDs flickering.

Preliminary Design

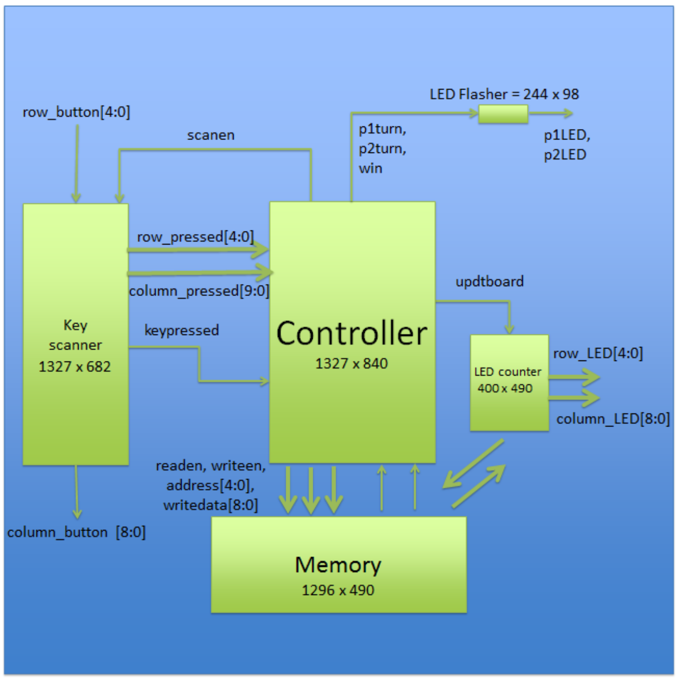
This finite state machine represents the logic of our controller. We need our memory to be only 1's at the beginning because a 1 will represent a block not eaten yet. Then player 1 gets to choose which block he wants to eat by pressing the corresponding button. The keyscanner module will poll the keyboard until a button is pressed. At this point, the button information is passed to the controller, and the "keypress" signal is driven high. On the next cycle, this signal will become low again in preparation for the next player's turn. The controller then temporarily disables the LED display, and reads the memory to see if the button press corresponds to a cell which has not yet been eaten. If the move is valid, it updates the memory with the new state of the game, and proceeds to check for a win. If the entire top row of the game board is empty, player 2 wins. Otherwise, the LEDs are relit, and the same cycle begins for player 2.



Floorplan

Below is our initial floorplan estimate. The memory will be custom-built as an array of settable, enable-able flip-flops, so its size was estimated by creating a slice plan. All other top level modules were to be synthesized. All of the top level modules except for the controller were simple enough for us to estimate how much hardware we thought they would use per bit of information. The calculations are shown below. The controller was estimated to be as tall as the keyscanner, so it could contain the 14 registers required to record an entire button press. Its width was chosen to match that of the MIPS8 controller in the mudd11 library, with the assumption that the Chomp controller would be the less complex of the two.

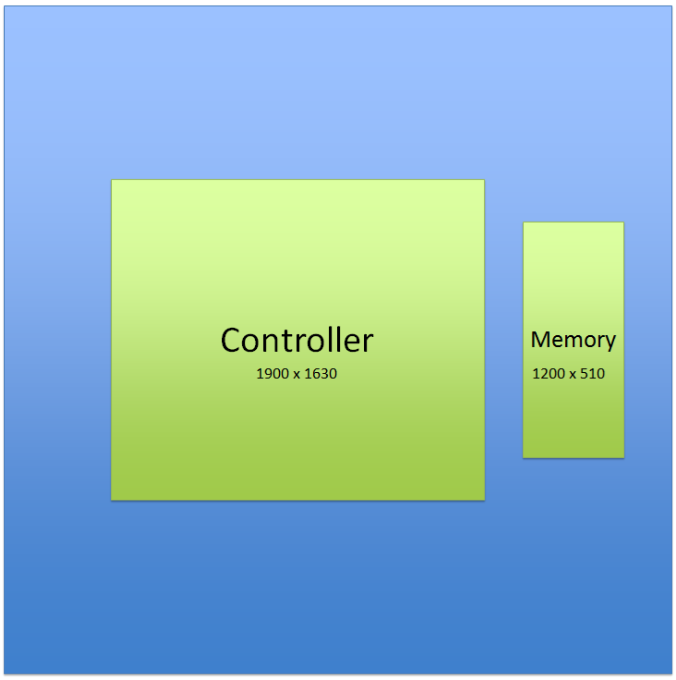
Predicted Layout

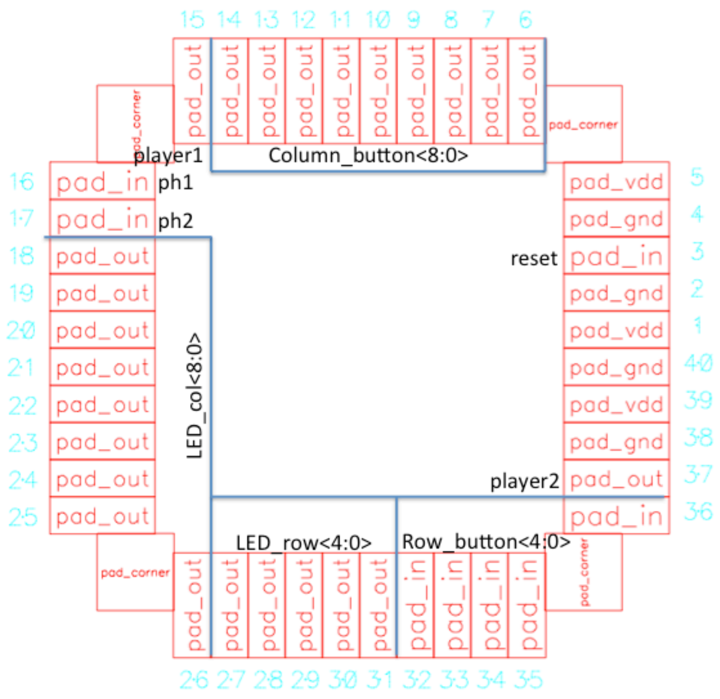


Total Core Area: 3400 x 3400

Our final layout differed some from our estimates. First, all of the synthesized blocks were combined into a single module. The width and height of this module are, however, reasonably close to the combined widths and heights of the individual modules originally proposed, with some extra space to accommodate the power rings. The memory size from the proposal was created assuming a flip-flop-based design, and did not take power rings into account, while the memory actually created was built using 12-transistor SRAM cells. Coincidentally, the smaller SRAM cell size combined with the additional area required by the power ring resulted in a final memory layout similar in size to that of the original estimate.

Final Layout





Verification

All components of the design passed our design verification tests. The only module with additional errors was the imported CIF, which contained one metal geometry error. Upon further investigation, however, the irregular geometry appeared to be overlapped by additional metal in the same layer, and would not show up if a mask of the entire layer were produced. We considered this to still be a DRC "pass".

Facet	Schematic/Layout	DRC	LVS
Full memory	Schematic	YES	YES
	Layout		
Synthesized verilog	Schematic	YES	YES
	Layout		
Core	Schematic	YES	YES
	Layout		
Padframe	Schematic	YES	YES
	Layout		
Chip	Schematic	YES	YES
	Layout		
CIF	Schematic	YES	YES
	Layout		

Post fabrication test plan

After receiving our chip, we would first test VDD and GND using a multimeter. Resistance between two VDD pins or two GND pins should be low, while resistance between VDD and GND should be several orders of magnitude higher. Then, we would connect the pseudo keyboard and all the LEDs. After applying power and the two non-overlapping clock signals, all the LEDs should turn on. Then we would need to test our finite state machine and make sure it works correctly.

Below are a few tests that we would do:

- Press the top left button after a reset, all the LEDs should turn off and the player 2 LED should be blinking, indicating that player 2 won.
- Play through a normal game, making sure that the lights respond to key presses according to the rules of Chomp, and that the correct winner is indicated.
- Try to press a button from a region that has already been eaten, nothing should change
- Simultaneously press 2 different buttons at the time.

A straightforward way to accomplish this would be to run through all of the shorter sets of test vectors used to verify the initial design. Done by hand, this should require no more than a couple of minutes per chip. The signals could be generated significantly more quickly by programming an FPGA to generate the button signals, and check for the appropriate outputs; however, this does not test to make sure that the clock frequency is appropriate for de-bouncing the buttons and eliminating flicker on the LEDs. If a shorter test cycle is required, one could still run an FPGA version of the test to make sure that the logic and memory function correctly, and check a smaller number of chips by hand to make sure that the buttons and LEDs are functioning appropriately.

Design time

The proof of concept Verilog was not that complicated to produce but we kept changing it when we were debugging using the self-checking testbench. We spent quite some time writing the self-checking testbench but it also includes all the debugging of our Verilog code.

We also spent a lot of time debugging the full memory schematic. A couple of wires from the 12T SRAM have to be their "nettype" as tristate, otherwise the memory doesn't behave correctly. We also wasted a lot of time because we weren't using one of the clock signals with our wordline logic.

Facet	Version	Time spent (h)
Proposal		6
Proof of concept Verilog		12
Self-checking testbench		30
Full memory	Schematic	20
	Layout	4
Synthesized logic	Schematic	0.5
	Layout	1
Padframe	Schematic	0.5
	Layout	2
Chip	Schematic	1
	Layout	1
CIF	Layout	0.5
Final report		5
Total		83.5

File locations

File	Location on Chips
Verilog code	/home/kaiach/chomp/chomp.v
Test fixture	/home/kaiach/chomp/chomp_testfixture.sv
Synthesis results	/home/kaiach/chomp/chomp_syn.sv
Test vectors	/home/kaiach/chomp/finalvectors.tv
Libraries	/home/kaiach/IC_CAD/cadence/proj2_complete
CIF	/home/kaiach/chomp/chomp_final.cif
PDF chip plots	/home/kaiach/chomp/report
PDF of the report	/home/kaiach/chomp/report/Final_report.pdf

References

http://lpcs.math.msu.su/~pentus/abacus.htm#rules

http://en.wikipedia.org/wiki/Chomp

Appendices

1. Verilog code

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: Harvey Mudd College
// Engineer: Kevin Aiach & Christian Jolivet
// Date: 03/13/2011
// Module Name:  chomp_top
// Project Name: Chomp Game
// Description:
// TO DO
// Please note that the modules flop, flopen,
// flopenr, latch, mux2, and mux3 were written by Professor Harris.
////////////////////////////////////////////////////////////////////////////////

module chomp(input ph1, ph2,
                                    input reset,
                                    input [4:0] row_button,
                                    output [8:0] column_button,
                                    output [8:0] column_LED,
                                    output [4:0] row_LED,
                                    output player1, player2);

wire writeen, readen;
wire [8:0] memread;
wire [8:0] memwrite;
wire [4:0] address;


synthesized  s(ph1,  ph2,reset,row_button,  readen,  writeen,  address,  memwrite,  memread,  column_button,
column_LED, row_LED, player1, player2);

fullmemory mem1(memread,readen,address,writeen,memwrite);

//memory mem1(ph1, ph2, reset, readen, writeen, address, memwrite, memread);

endmodule

module synthesized(input ph1, ph2,
                                    input reset,
                                    input [4:0] row_button,
                                    output readen,
                                    output writeen,
                                    output [4:0] address,
                                    output [8:0] memwrite,
                                    input [8:0] memread,
                                    output [8:0] column_button,
                                    output [8:0] column_LED,
                                    output [4:0] row_LED,
                                    output player1, player2);

wire [13:0] lastkey;
wire keypressed;
wire updtboard;
wire scanen;
wire win;
wire [8:0] memprewrite;
wire [4:0] preaddress;
wire [4:0] address0;
wire [4:0] address1;
wire player1turn;
wire player2turn;


controller       control1(ph1, ph2, reset, keypressed, lastkey[13:9], lastkey[8:0], memread,
                                   writeen, readen, updtboard, scanen, win, address0,
                                   memwrite, player1turn, player2turn);

LEDcounter LEDcont1(ph1, ph2, reset, updtboard, memread, address1, column_LED, row_LED);

mux2 #5 addressmux(address0, address1, updtboard, address);

//flop #14 memprotect(ph1, ph2, {memprewrite,preaddress} , {memwrite,address});

keyreport keyreport1(ph1, ph2, reset, scanen, row_button, column_button, lastkey, keypressed);

LEDflasher flasher(ph1, ph2, reset, win, player1turn, player2turn, player1, player2);

endmodule


module controller(input ph1, ph2,
                                     input reset,
                                     input keypressed,
                                     input [4:0] row_button,
                                     input [8:0] column_button,
                                     input [8:0] memread,
                                     output reg writeen, readen,
                                     output reg updtboard,
                                     output reg scanen,
                                     output reg win,
                                     output reg [4:0] address,
                                     output reg [8:0] memwrite,
                                     output player1, player2);

parameter SIZE = 12;

//List of states
parameter START = 12'h000;

parameter P1_PLAYS = 12'h001;
parameter BUTTON_OFF_P1 = 12'h002;

parameter MEM_READ4_P1  = 12'h003;
parameter BUFFERSTATE0  = 12'h004;
parameter MEM_WRITE4_P1 = 12'h005;
parameter BUFFERSTATE1  = 12'h006;
parameter MEM_READ3_P1  = 12'h007;
parameter BUFFERSTATE2  = 12'h008;
parameter MEM_WRITE3_P1 = 12'h009;
parameter BUFFERSTATE3  = 12'h00A;
parameter MEM_READ2_P1  = 12'h00B;
parameter BUFFERSTATE4  = 12'h00C;
parameter MEM_WRITE2_P1 = 12'h00D;
parameter BUFFERSTATE5  = 12'h00E;
parameter MEM_READ1_P1  = 12'h00F;
parameter BUFFERSTATE6  = 12'h010;
parameter MEM_WRITE1_P1 = 12'h011;
parameter BUFFERSTATE7  = 12'h012;
parameter MEM_READ0_P1  = 12'h013;
parameter BUFFERSTATE8  = 12'h014;
parameter MEM_WRITE0_P1 = 12'h015;
parameter BUFFERSTATE9  = 12'h016;
parameter CHECKWIN_P2  = 12'h017;
parameter P1_WINS = 12'h018;


parameter P2_PLAYS = 12'h101;
parameter BUTTON_OFF_P2 = 12'h102;

parameter MEM_READ4_P2  = 12'h103;
parameter BUFFERSTATE10  = 12'h104;
parameter MEM_WRITE4_P2 = 12'h105;
parameter BUFFERSTATE11  = 12'h106;
parameter MEM_READ3_P2  = 12'h107;
parameter BUFFERSTATE12  = 12'h108;
parameter MEM_WRITE3_P2 = 12'h109;
parameter BUFFERSTATE13  = 12'h10A;
parameter MEM_READ2_P2  = 12'h10B;
parameter BUFFERSTATE14  = 12'h10C;
parameter MEM_WRITE2_P2 = 12'h10D;
parameter BUFFERSTATE15  = 12'h10E;
parameter MEM_READ1_P2  = 12'h10F;
parameter BUFFERSTATE16  = 12'h110;
parameter MEM_WRITE1_P2 = 12'h111;
parameter BUFFERSTATE17  = 12'h112;
parameter MEM_READ0_P2  = 12'h113;
parameter BUFFERSTATE18  = 12'h114;
parameter MEM_WRITE0_P2 = 12'h115;
parameter BUFFERSTATE19  = 12'h116;
parameter CHECKWIN_P1 = 12'h117;
parameter P2_WINS = 12'h118;

wire [SIZE-1:0] state;
reg valid;
reg [SIZE-1:0] next_state;
reg player;
wire [8:0] lastmemread;
reg [8:0] decodedcol;
reg [4:0] decodedrow;

// state register
flopenrval #SIZE statereg(ph1, ph2, reset, 1'b1, START, next_state, state);

// read register
flopenr     #9   readreg(ph1, ph2, reset, readen, memread, lastmemread);

// nextstate logic
always @(*)
        case(state)
                START:
                begin
                        next_state = P1_PLAYS;
                end
                P1_PLAYS:
```

```
begin
        if(keypressed == 1) begin
                next_state = BUTTON_OFF_P1;
        end else begin
                next_state = P1_PLAYS;
        end
end
BUTTON_OFF_P1:
begin
        //valid = |(memread & column_button);
        if(valid == 1) begin
                next_state = MEM_READ4_P1;
        end else begin
                next_state = P1_PLAYS;
        end
end

MEM_READ4_P1:
begin
        next_state = BUFFERSTATE0;
end
BUFFERSTATE0:
begin
        next_state = MEM_WRITE4_P1;
end
MEM_WRITE4_P1:
begin
        next_state = BUFFERSTATE1;
end

BUFFERSTATE1:
begin
        next_state = MEM_READ3_P1;
end

MEM_READ3_P1:
begin
        next_state = BUFFERSTATE2;
end
BUFFERSTATE2:
begin
        next_state = MEM_WRITE3_P1;
end
MEM_WRITE3_P1:
begin
        next_state = BUFFERSTATE3;
end
BUFFERSTATE3:
begin
        next_state = MEM_READ2_P1;
end
MEM_READ2_P1:
begin
        next_state = BUFFERSTATE4;
end
BUFFERSTATE4:
begin
        next_state = MEM_WRITE2_P1;
end
MEM_WRITE2_P1:
begin
        next_state = BUFFERSTATE5;
end
BUFFERSTATE5:
begin
        next_state = MEM_READ1_P1;
end
MEM_READ1_P1:
begin
        next_state = BUFFERSTATE6;
end
BUFFERSTATE6:
begin
        next_state = MEM_WRITE1_P1;
end
MEM_WRITE1_P1:
begin
        next_state = BUFFERSTATE7;
end
BUFFERSTATE7:
begin
        next_state = MEM_READ0_P1;
end
MEM_READ0_P1:
begin
        next_state = BUFFERSTATE8;
end
BUFFERSTATE8:
begin
        next_state = MEM_WRITE0_P1;
```

```
end
MEM_WRITE0_P1:
begin
        next_state = BUFFERSTATE9;
end
BUFFERSTATE9:
begin
        next_state = CHECKWIN_P2;
end
CHECKWIN_P2:
begin
        if(memread == 9'b000000000)
        next_state = P2_WINS;
        else next_state = P2_PLAYS;
end
P2_PLAYS:
begin
        if(keypressed == 1) begin
                next_state = BUTTON_OFF_P2;
        end else begin
                next_state = P2_PLAYS;
        end
end
BUTTON_OFF_P2:
begin
        //valid = |(memread & column_button);
        if(valid == 1) begin
                next_state = MEM_READ4_P2;
        end else begin
                next_state = P2_PLAYS;
        end
end
MEM_READ4_P2:
begin
        next_state = BUFFERSTATE10;
end
BUFFERSTATE10:
begin
        next_state = MEM_WRITE4_P2;
end
MEM_WRITE4_P2:
begin
        next_state = BUFFERSTATE11;
end
BUFFERSTATE11:
begin
        next_state = MEM_READ3_P2;
end
MEM_READ3_P2:
begin
        next_state = BUFFERSTATE12;
end
BUFFERSTATE12:
begin
        next_state = MEM_WRITE3_P2;
end
MEM_WRITE3_P2:
begin
        next_state = BUFFERSTATE13;
end
BUFFERSTATE13:
begin
        next_state = MEM_READ2_P2;
end
MEM_READ2_P2:
begin
        next_state = BUFFERSTATE14;
end
BUFFERSTATE14:
begin
        next_state = MEM_WRITE2_P2;
end
MEM_WRITE2_P2:
begin
        next_state = BUFFERSTATE15;
end
BUFFERSTATE15:
begin
        next_state = MEM_READ1_P2;
end
MEM_READ1_P2:
begin
        next_state = BUFFERSTATE16;
end
BUFFERSTATE16:
begin
        next_state = MEM_WRITE1_P2;
end
MEM_WRITE1_P2:
begin
```

```
                        next_state = BUFFERSTATE17;
                end
                BUFFERSTATE17:
                begin
                        next_state = MEM_READ0_P2;
                end
                MEM_READ0_P2:
                begin
                        next_state = BUFFERSTATE18;
                end
                BUFFERSTATE18:
                begin
                        next_state = MEM_WRITE0_P2;
                end
                MEM_WRITE0_P2:
                begin
                        next_state = BUFFERSTATE19;
                end
                BUFFERSTATE19:
                begin
                        next_state = CHECKWIN_P1;
                end
                CHECKWIN_P1:
                begin
                        if(memread == 9'b000000000)
                        next_state = P1_WINS;
                        else next_state = P1_PLAYS;
                end

                P1_WINS:
                begin
                        next_state = P1_WINS;
                end

                P2_WINS:
                begin
                        next_state = P2_WINS;
                end

        default next_state = START;
        endcase


//output logic


always @(*)
        begin
        casex(column_button)
                                9'b000000001: decodedcol = 9'b111111110;
                                9'b00000001x: decodedcol = 9'b111111100;
                                9'b0000001xx: decodedcol = 9'b111111000;
                                9'b000001xxx: decodedcol = 9'b111110000;
                                9'b00001xxxx: decodedcol = 9'b111100000;
                                9'b0001xxxxx: decodedcol = 9'b111000000;
                                9'b001xxxxxx: decodedcol = 9'b110000000;
                                9'b01xxxxxxx: decodedcol = 9'b100000000;
                                9'b1xxxxxxxx: decodedcol = 9'b000000000;
                        default: decodedcol = 9'b111111111;
        endcase
        casex(row_button)
                                5'b00001: decodedrow = 5'b00001;
                                5'b0001x: decodedrow = 5'b00010;
                                5'b001xx: decodedrow = 5'b00100;
                                5'b01xxx: decodedrow = 5'b01000;
                                5'b1xxxx: decodedrow = 5'b10000;
                        default: decodedrow = 5'b00000;
        endcase
        case(state)
                START:
                        begin
                        writeen = 1;
                        address = 5'b11111;
                        memwrite = 9'b111111111;
                        readen = 0;
                        valid = 0;
                        scanen = 0;
                        player = 0;
                        updtboard = 0;
                        win = 0;
                        end
                P1_PLAYS:
                        begin
                        writeen = 0;
                        address = 5'b00000;
                        memwrite = 9'b111111111;
                        readen = 1;
                        valid = 0;
                        scanen = 1;
                        player = 0;
        updtboard = 1;
        win = 0;
        end
BUTTON_OFF_P1:
        begin
        address = decodedrow;
        writeen = 0;
        readen = 1;
        scanen = 0;
        updtboard = 0;
        player = 0;
        memwrite = 9'b111111111;
        win = 0;
        valid = |(memread & column_button);
        end
MEM_READ4_P1:
        begin
        valid = 0;
        scanen = 0;
        writeen = 0;
        readen = 1;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b10000;
        memwrite = 9'b111111111;
        end

BUFFERSTATE0:
        begin
        valid = 0;
        scanen = 0;
        writeen = 0;
        readen = 0;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b00000;

        casex(decodedrow)
                5'b10000: memwrite = decodedcol & lastmemread;
                default:  memwrite = lastmemread;
        endcase
        end

MEM_WRITE4_P1:
        begin
        valid = 0;
        scanen = 0;
        writeen = 1;
        readen = 0;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b10000;

        casex(decodedrow)
                5'b10000: memwrite = decodedcol & lastmemread;
                default:  memwrite = lastmemread;
        endcase
        end

BUFFERSTATE1:
        begin
        valid = 0;
        scanen = 0;
        writeen = 0;
        readen = 0;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b00000;

        casex(decodedrow)
                5'b10000: memwrite = decodedcol & lastmemread;
                default:  memwrite = lastmemread;
        endcase
        end

MEM_READ3_P1:
        begin
        valid = 0;
        scanen = 0;
        writeen = 0;
        readen = 1;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b01000;
        memwrite = 9'b111111111;
        end

BUFFERSTATE2:
        begin
        valid = 0;
        scanen = 0;
        writeen = 0;
        readen = 0;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b00000;

        casex(decodedrow)
                5'b10000: memwrite = decodedcol & lastmemread;
                5'b01000: memwrite = decodedcol & lastmemread;
                default:  memwrite = lastmemread;
        endcase
        end

MEM_WRITE3_P1:
        begin
        valid = 0;
        scanen = 0;
        writeen = 1;
        readen = 0;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b01000;

        casex(decodedrow)
                5'b10000: memwrite = decodedcol & lastmemread;
                5'b01000: memwrite = decodedcol & lastmemread;
                default:  memwrite = lastmemread;
        endcase
        end

BUFFERSTATE3:
        begin
        valid = 0;
        scanen = 0;
        writeen = 0;
        readen = 0;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b00000;

        casex(decodedrow)
                5'b10000: memwrite = decodedcol & lastmemread;
                5'b01000: memwrite = decodedcol & lastmemread;
                default:  memwrite = lastmemread;
        endcase
        end

MEM_READ2_P1:
        begin
        valid = 0;
        scanen = 0;
        writeen = 0;
        readen = 1;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b00100;
        memwrite = 9'b111111111;
        end

BUFFERSTATE4:
        begin
        valid = 0;
        scanen = 0;
        writeen = 0;
        readen = 0;
        player = 0;
        updtboard = 0;
        win = 0;
        address = 5'b00000;

        casex(decodedrow)
                5'b10000: memwrite = decodedcol & lastmemread;
                5'b01000: memwrite = decodedcol & lastmemread;
                5'b00100: memwrite = decodedcol & lastmemread;
                default:  memwrite = lastmemread;
        endcase
        end

MEM_WRITE2_P1:
        begin
```

```
                valid = 0;
                scanen = 0;
                writeen = 1;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00100;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        BUFFERSTATE5:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end
        MEM_READ1_P1:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 1;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00010;
                memwrite = 9'b111111111;
                end

        BUFFERSTATE6:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_WRITE1_P1:
                begin
                valid = 0;
                scanen = 0;
                writeen = 1;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00010;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        BUFFERSTATE7:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_READ0_P1:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 1;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00001;
                memwrite = 9'b111111111;
                end

        BUFFERSTATE8:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        5'b00001: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_WRITE0_P1:
                begin
                valid = 0;
                scanen = 0;
                writeen = 1;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00001;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        5'b00001: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        BUFFERSTATE9:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        5'b00001: memwrite = decodedcol & lastmemread;
```

```
                        default:  memwrite = lastmemread;
                endcase
                end

        CHECKWIN_P2:
                begin
                writeen = 0;
                readen = 1;
                scanen = 0;
                player = 0;
                updtboard = 0;
                address = 5'b10000;
                win = 0;
                memwrite = 9'b111111111;
                valid = 0;
                end
        P2_PLAYS:
                begin
                address = 5'b00000;
                writeen = 0;
                readen = 1;
                scanen = 1;
                player = 1;
                updtboard = 1;
                memwrite = 9'b111111111;
                win = 0;
                valid = 0;
                end
        BUTTON_OFF_P2:
                begin
                address = decodedrow;
                writeen = 0;
                readen = 1;
                scanen = 0;
                updtboard = 0;
                player = 1;
                memwrite = 9'b111111111;
                win = 0;
                valid = |(memread & column_button);
                end
        MEM_READ4_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 1;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b10000;
                memwrite = 9'b111111111;
                end

        BUFFERSTATE10:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_WRITE4_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 1;
                readen = 0;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b10000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        BUFFERSTATE11:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_READ3_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 1;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b01000;
                memwrite = 9'b111111111;
                end

        BUFFERSTATE12:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_WRITE3_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 1;
                readen = 0;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b01000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        BUFFERSTATE13:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_READ2_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 1;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b00100;
                memwrite = 9'b111111111;
                end

        BUFFERSTATE14:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_WRITE2_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 1;
                readen = 0;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b00100;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        BUFFERSTATE15:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_READ1_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 1;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b00010;
                memwrite = 9'b111111111;
                end

        BUFFERSTATE16:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
```

```
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_WRITE1_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 1;
                readen = 0;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b00010;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        BUFFERSTATE17:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_READ0_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 1;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b00001;
                memwrite = 9'b111111111;
                end

        BUFFERSTATE18:
                begin
                valid = 0;
                scanen = 0;
                writeen = 0;
                readen = 0;
                player = 0;
                updtboard = 0;
                win = 0;
                address = 5'b00000;

                casex(decodedrow)
                        5'b10000: memwrite = decodedcol & lastmemread;
                        5'b01000: memwrite = decodedcol & lastmemread;
                        5'b00100: memwrite = decodedcol & lastmemread;
                        5'b00010: memwrite = decodedcol & lastmemread;
                        5'b00001: memwrite = decodedcol & lastmemread;

                        default:  memwrite = lastmemread;
                endcase
                end

        MEM_WRITE0_P2:
                begin
                valid = 0;
                scanen = 0;
                writeen = 1;
                readen = 0;
                player = 1;
                updtboard = 0;
                win = 0;
                address = 5'b00001;
```

```
                        casex(decodedrow)
                                5'b10000: memwrite = decodedcol & lastmemread;
                                5'b01000: memwrite = decodedcol & lastmemread;
                                5'b00100: memwrite = decodedcol & lastmemread;
                                5'b00010: memwrite = decodedcol & lastmemread;
                                5'b00001: memwrite = decodedcol & lastmemread;
                                default:  memwrite = lastmemread;
                        endcase
                        end

                BUFFERSTATE19:
                        begin
                        valid = 0;
                        scanen = 0;
                        writeen = 0;
                        readen = 0;
                        player = 0;
                        updtboard = 0;
                        win = 0;
                        address = 5'b00000;

                        casex(decodedrow)
                                5'b10000: memwrite = decodedcol & lastmemread;
                                5'b01000: memwrite = decodedcol & lastmemread;
                                5'b00100: memwrite = decodedcol & lastmemread;
                                5'b00010: memwrite = decodedcol & lastmemread;
                                5'b00001: memwrite = decodedcol & lastmemread;
                                default:  memwrite = lastmemread;
                        endcase
                        end
                CHECKWIN_P1:
                        begin
                        writeen = 0;
                        readen = 1;
                        scanen = 0;
                        player = 1;
                        updtboard = 0;
                        address = 5'b10000;
                        win = 0;
                        memwrite = 9'b111111111;
                        valid = 0;
                        end
                P1_WINS:
                        begin
                        writeen = 0;
                        address = 5'b10000;
                        memwrite = 9'b111111111;
                        readen = 1;
                        valid = 0;
                        scanen = 0;
                        player = 0;
                        updtboard = 1;
                        win = 1;
                        end
                P2_WINS:
                        begin
                        writeen = 0;
                        address = 5'b10000;
                        memwrite = 9'b111111111;
                        readen = 1;
                        valid = 0;
                        scanen = 0;
                        player = 1;
                        updtboard = 1;
                        win = 1;
                        end
                default:
                        begin
                        writeen = 1;
                        address = 5'b11111;
                        memwrite = 9'b111111111;
                        readen = 0;
                        valid = 0;
                        scanen = 0;
                        player = 0;
                        updtboard = 0;
                        win = 0;
                        end
        endcase
        end

assign player1 = ~player;
assign player2 = player;

endmodule

// Module representing the memory
module memory(input ph1, ph2,
                          input reset,
                          input readen, writeen,
                            input [4:0] wordline,
                            input [8:0] writelinez,
                            output [8:0] readline);

        wire [8:0] row0, row1, row2, row3, row4;
        reg [8:0] readlinez;
        wire [8:0] writeline;

        //assign writeline = writeen ? writelinez : 9'bzzzzzzzzz;

        //flopenrval #width name(ph1, ph2, reset, en, [8:0]d, [8:0]q);
        flopenr #9 memRow0(ph1, ph2, reset, writeen&wordline[0], writeline, row0);
        flopenr #9 memRow1(ph1, ph2, reset, writeen&wordline[1], writeline, row1);
        flopenr #9 memRow2(ph1, ph2, reset, writeen&wordline[2], writeline, row2);
        flopenr #9 memRow3(ph1, ph2, reset, writeen&wordline[3], writeline, row3);
        flopenr #9 memRow4(ph1, ph2, reset, writeen&wordline[4], writeline, row4);

        always@(*)
                casez (wordline)
                        5'b1????: readlinez = row4;
                        5'b01???: readlinez = row3;
                        5'b001??: readlinez = row2;
                        5'b0001?: readlinez = row1;
                        5'b00001: readlinez = row0;
                        default: readlinez = 9'bxxxxxxxxx;
                endcase

        //assign readline = readen ? readlinez : 9'bzzzzzzzzz;
endmodule

// Modules provided by Prof. Harris
/*module tristate(input en,
                                   input [2:0] d,
                                   output [2:0] q);

        assign q = en ? d : 3'bzzz;
endmodule*/

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

        wire [WIDTH-1:0] mid;

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

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

        wire [WIDTH-1:0] d2;

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

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

        wire [WIDTH-1:0] d2, resetval;
        assign resetval = 0;

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

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

        wire [WIDTH-1:0] d2;

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

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

        always@(*)
```

```
                if (ph) q <= d;
endmodule

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

        assign y = s ? d1 : d0;
endmodule

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

        always@(*)
                casez (s)
                        2'b00: y = d0;
                        2'b01: y = d1;
                        2'b1?: y = d2;
                endcase
endmodule

module keyscan(
    input ph1, ph2,
    input reset,
    input [4:0] button,
    output [8:0] col,
         output keypress
    );

        wire [13:0] key;
        reg [8:0] nextcol;

        parameter S0 = 9'b100000000;
        parameter S1 = 9'b010000000;
        parameter S2 = 9'b001000000;
        parameter S3 = 9'b000100000;
        parameter S4 = 9'b000010000;
        parameter S5 = 9'b000001000;
        parameter S6 = 9'b000000100;
        parameter S7 = 9'b000000010;
        parameter S8 = 9'b000000001;


        assign keypress = | button;

        flopenrval #9 statereg(ph1, ph2, reset, 1'b1, S0, nextcol, col);

        always @(*)
        begin
                case(col)
                        S0: nextcol <= S1;
                        S1: nextcol <= S2;
                        S2: nextcol <= S3;
                        S3: nextcol <= S4;
                        S4: nextcol <= S5;
                        S5: nextcol <= S6;
                        S6: nextcol <= S7;
                        S7: nextcol <= S8;
                        S8: nextcol <= S0;
                        default: nextcol <= S0;
                endcase
        end
endmodule

module keyreport(
   input ph1, ph2,
   input reset,
        input scanen,
   input [4:0] button,            // input from buttons
   output [8:0] col,              // col getting scanned
   output [13:0] lastkey,   // most recent press
        output recorded);

        reg [13:0] newkey;
        wire keypress;
        reg newrecorded;

        keyscan keyreader(ph1, ph2, reset, button, col, keypress);

        flopenrval #14 statereg(ph1, ph2, reset, scanen, 14'b0, newkey, lastkey);
        flopenrval #1 statereg2(ph1, ph2, reset, 1'b1, 1'b0, newrecorded, recorded);

        always@(*)
                case(recorded)
                0:
                        if (keypress)
                                begin
```

```
                                        newkey = {button, col};
                                        newrecorded = keypress;
                                        end
                                else
                                        begin
                                        newkey = lastkey;
                                        newrecorded = keypress;
                                        end
                        1:
                                begin
                                        newkey = lastkey;
                                        newrecorded = keypress;
                                end
                        endcase
endmodule

module LEDcounter(
   input ph1, ph2,
   input reset,
        input updtboard,
        input [8:0] memread,
        output [4:0] address,
        output reg [8:0] LEDcol,
        output reg [4:0] LEDrow);

        reg [4:0] newaddress;

        parameter S0 = 5'b10000;
        parameter S1 = 5'b01000;
        parameter S2 = 5'b00100;
        parameter S3 = 5'b00010;
        parameter S4 = 5'b00001;

        flopenrval #5 statereg(ph1, ph2, reset, updtboard, 5'b00001, newaddress, address);

        always @(*)
        begin
                case(address)
                        S0: newaddress <= S1;
                        S1: newaddress <= S2;
                        S2: newaddress <= S3;
                        S3: newaddress <= S4;
                        S4: newaddress <= S0;
                        default: newaddress <= S0;
                endcase
        end

        always @(*)
                if(updtboard)
                        begin
                                LEDcol = memread;
                                LEDrow = ~address;
                        end
                else
                        begin
                                LEDcol = 9'b0;
                                LEDrow = 5'b0;
                        end
endmodule

module LEDflasher(
   input ph1, ph2,
   input reset,
        input win,
        input player1turn,
        input player2turn,
        output reg p1LED,
        output reg p2LED);

        wire state;
        wire newstate;

        flopenrval #1 statereg(ph1, ph2, reset, 1'b1, 1'b0, newstate, state);

        assign newstate = ~state;

        always @(*)
                if(win & player1turn)
                        begin
                                p1LED = state;
                                p2LED = player2turn;
                        end
                else if(win & player2turn)
                        begin
                                p1LED = player1turn;
                                p2LED = state;
                        end
                else
                        begin
                                p1LED = player1turn;
```

```
                                    p2LED = player2turn;
                            end
endmodule

module fullmemory( output [8:0] readline,
                   input readen,
                   input [4:0] wordline,
                   input writeen,
                   input [8:0] writeline );
wire [4:0] re;
wire [4:0] we;

assign re[4] = readen & wordline[4];
assign re[3] = readen & wordline[3];
assign re[2] = readen & wordline[2];
assign re[1] = readen & wordline[1];
assign re[0] = readen & wordline[0];

assign we[4] = writeen & wordline[4];
assign we[3] = writeen & wordline[3];
assign we[2] = writeen & wordline[2];
assign we[1] = writeen & wordline[1];
assign we[0] = writeen & wordline[0];


sram5by9 s59(we, re, writeline, readline);

endmodule

module sram5by9(/*input [4:0] address,*/ input [4:0] writeen, input [4:0] readen,  input [8:0] writeport,
output [8:0] readport);

sramword9 w4(writeen[4], readen[4], writeport, readport);
sramword9 w3(writeen[3], readen[3], writeport, readport);
sramword9 w2(writeen[2], readen[2], writeport, readport);
sramword9 w1(writeen[1], readen[1], writeport, readport);
sramword9 w0(writeen[0], readen[0], writeport, readport);

endmodule


module sramword9(input writeen,
                 input readen,
                 input  [8:0] writeport,
                 output [8:0] readport);

        sramcell c8(readport[8], writeport[8], readen, writeen);
        sramcell c7(readport[7], writeport[7], readen, writeen);
        sramcell c6(readport[6], writeport[6], readen, writeen);
        sramcell c5(readport[5], writeport[5], readen, writeen);
        sramcell c4(readport[4], writeport[4], readen, writeen);
        sramcell c3(readport[3], writeport[3], readen, writeen);
        sramcell c2(readport[2], writeport[2], readen, writeen);
        sramcell c1(readport[1], writeport[1], readen, writeen);
        sramcell c0(readport[0], writeport[0], readen, writeen);
endmodule

module sramcell(output readport, input writeport, input readen, input writeen);

wire saveddata;
wire saveddatab;
wire writeenb;

inv iw(writeen, writeenb);

tristate t1(writeport, writeen, saveddata);
tristateinv timem(saveddatab, writeenb, saveddata);

inv imem(saveddata, saveddatab);

tristateinv ir(saveddatab, readen, readport);
endmodule

/*module sramcell2(input writeen,
                                            input readen,
                                            input writeport,
                                            output reg readport);

reg saveddata;
wire saveddatab;
always@(*)
        begin
                case(writeen)
                        1: saveddata = writeport;
                        0: saveddata = ~saveddatab;
                endcase
                casez(readen)
                        1: readport = ~saveddatab;
                        0: readport = 1'bz;
                endcase
        end

assign saveddatab = ~saveddata;

endmodule*/



module tristate(input a,s,
                output y);
        assign y = s ? a:1'bz;
endmodule


module tristateinv(input a,s,
                   output y);
        assign y = s ? ~a:1'bz;
endmodule

module inv(input a, output y);
        assign y = ~a;
endmodule
```

2. Testing Materials

Below is our testbench for the top level chip. An example of a testvector file (which plays through a game) follows. During actual testing, we used several testbenches. Some were designed to test corner cases, while others were designed to be mostly random.

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

module test_synth();

        // Inputs
        logic ph1,ph2, reset;
        logic [4:0] row_button;
        logic [8:0] column_button;

        // Outputs
        logic player1;
        logic player2;
        logic [8:0] LED_col;
        logic [4:0] LED_row;

        // Test only
        logic [4:0] row;
        logic [8:0] col;

        logic [8:0] r4;
        logic [8:0] r3;
        logic [8:0] r2;
        logic [8:0] r1;
        logic [8:0] r0;

        logic p1expected;
        logic p2expected;

        logic flashing;
        logic winexpected;




        logic [9:0] cycle;
        logic [10:0] vectornum, errors;
        logic [61:0] testvectors [1000:0];
        logic [5:0] state;
        logic [5:0] newstate;

        //logic [20:0] expectedoutputs;


        //instantiate Device Under Test (DUT)
        //chomp dut(.*);
        chip_final  c(LED_col,  LED_row,  column_button,  player1,  player2,  ph1,  ph2,  reset,
row_button);


          // initialize test
          initial
            begin
                $readmemb("vectors2.tv", testvectors);
                vectornum = 0; errors = 0; cycle = 0; flashing = 0;
                reset <= 1; # 25; reset <= 0;
            end

        //Generate 2 phase non overlapping clock
         always
         begin
                ph1 = 0; ph2 = 0; #3;
                ph1 = 1; #4;
                ph1 = 0; #3;
                ph2 = 1; #4;
         end

         // apply test vectors on rising edge of clk
        always @(posedge ph1) begin
           #1;          {row,col,r4,r3,r2,r1,r0,p1expected,p2expected,winexpected}          =
testvectors[vectornum];
           if(column_button == col & state == 6'b111111)
                row_button = row;
           else
                row_button = 5'b00000;
         end

        // state machine makes sure we check every row of lights
        always @(posedge ph1)
           if(reset)
                state = 6'b111111;
           else
                state = newstate;

        always @(*)
        begin
           if(state == 6'b111110)
           begin
                newstate = 6'b111111;
           end
           else if(state == 6'b111111)
                   if(LED_row == 5'b00000)
                        newstate = 6'b000000;
                   else
                        newstate = state;
           else
           case(LED_row)
                5'b01111:
                   begin
                   newstate = state | {(~LED_row),1'b0};
                   end
                5'b10111:
                   begin
                   newstate = state | {(~LED_row),1'b0};
                   end
                5'b11011:
                   begin
                   newstate = state | {(~LED_row),1'b0};
                   end
                5'b11101:
                   begin
                   newstate = state | {(~LED_row),1'b0};
                   end
                5'b11110:
                   begin
                   newstate = state | {(~LED_row),1'b0};
                   end
                5'b00000:
                   newstate = 6'b000000;
                default:
                   newstate = 6'b111111;
           endcase
        end




        always @(posedge ph2)
        begin
                if(!reset)
                begin
                        cycle = cycle + 1;
                        if(cycle == 15'b111111111111111 | testvectors[vectornum] === 61'bx)
                        begin
                                $display("timeout");
                                $finish;
                        end
                        if((state != 6'b111110) & (state != 6'b111111) & (LED_row != 5'b00000))
                            begin
                                if((player1 != p1expected | player2 != p2expected) & winexpected
=== 0)
                                begin
                                    errors = errors+1;
                                    $display("wrong player");
                                end
                                if(winexpected === 1)
```

```
                              begin
                                  if((player1 === 0) & (player2 === 0))
                                  begin
                                     $display("flasher working");
                                     flashing = 1;
                                  end
                                  if((flashing === 1) & (player1 === 1))
                                     begin
                                     $display("player 1 wins");
                                     $finish;
                                     end
                                  if((flashing === 1) & (player2 === 1))
                                     begin
                                     $display("player 2 wins");
                                     $finish;
                                     end
                              end
                              case(LED_row)
                                  5'b01111:
                                  begin
                                  if(LED_col != r4)begin
                                  errors = errors + 1;
                                  $display("r4 is wrong");
                                  end
                                  end
                                  5'b10111:
                                  begin
                                  if(LED_col != r3)begin
                                     errors = errors + 1;
                                  $display("r3 is wrong");
                                  end
                                  end
                                  5'b11011:
                                  begin
                                  if(LED_col != r2)begin
                                     errors = errors + 1;
                                  $display("r2 is wrong");
                                  end
                                  end
                                  5'b11101:
                                  begin
                                  if(LED_col != r1)begin
                                  errors = errors + 1;
                                  $display("r1 is wrong");
                                  end
                                  end
                                  5'b11110:
                                  begin
                                  if(LED_col != r0)begin
                                    errors = errors + 1;
                                  $display("r0 is wrong");
                                  end
                                  end
                                  default:
                                     errors = errors;
                          endcase
                      end
                      if(state == 6'b111110)
                      vectornum = vectornum + 1;
               end

       end

endmodule
```

Testbench:

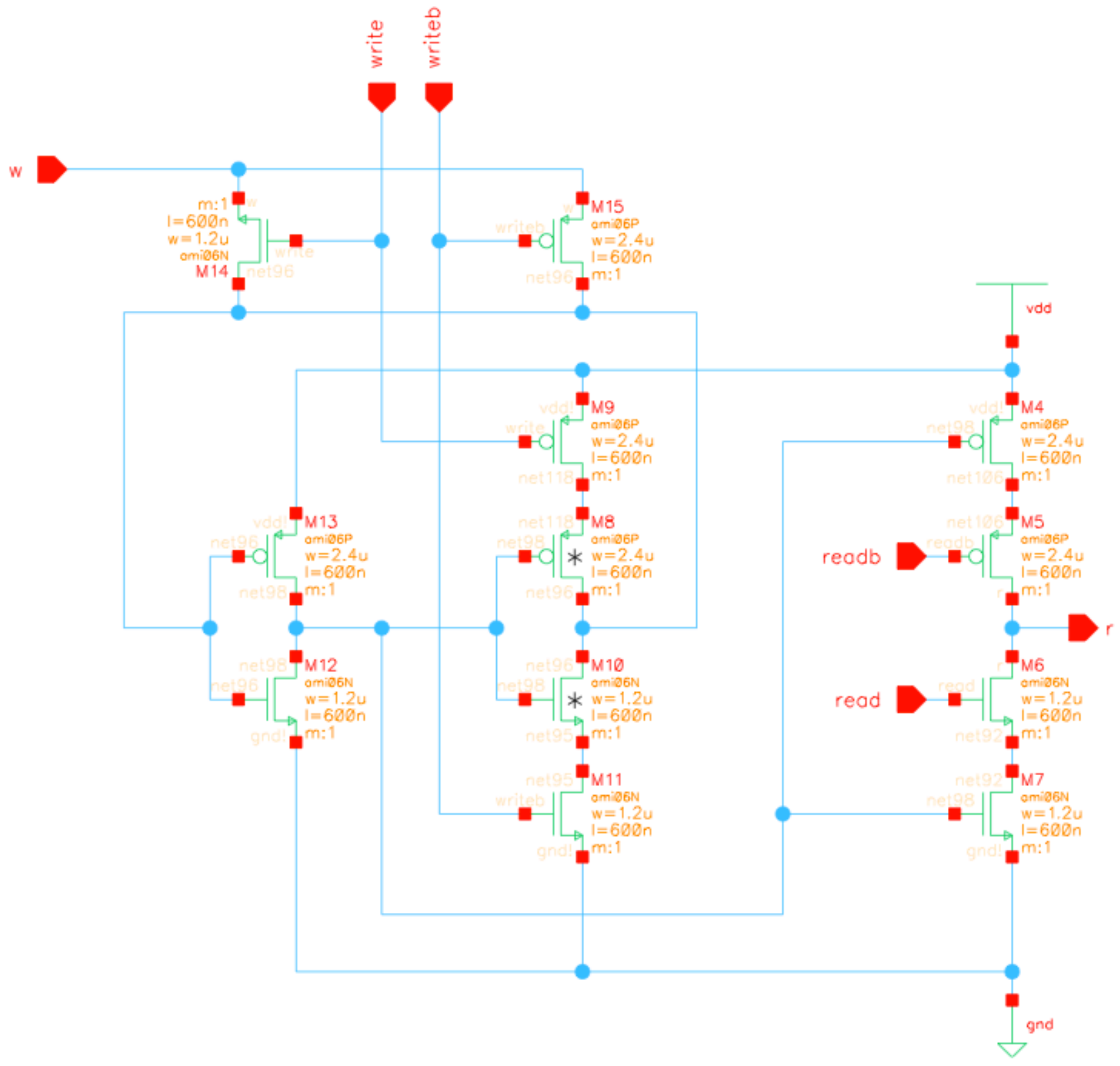
This testbench was generated using random walks to generate input button presses until the board was mostly blank, and then manually adjusting the last line to force one of the players to win. Other testbenches were designed to make sure each button press could be captured, each memory location could be uniquely addressed, and that the system would respond appropriately if more than one button was pressed.

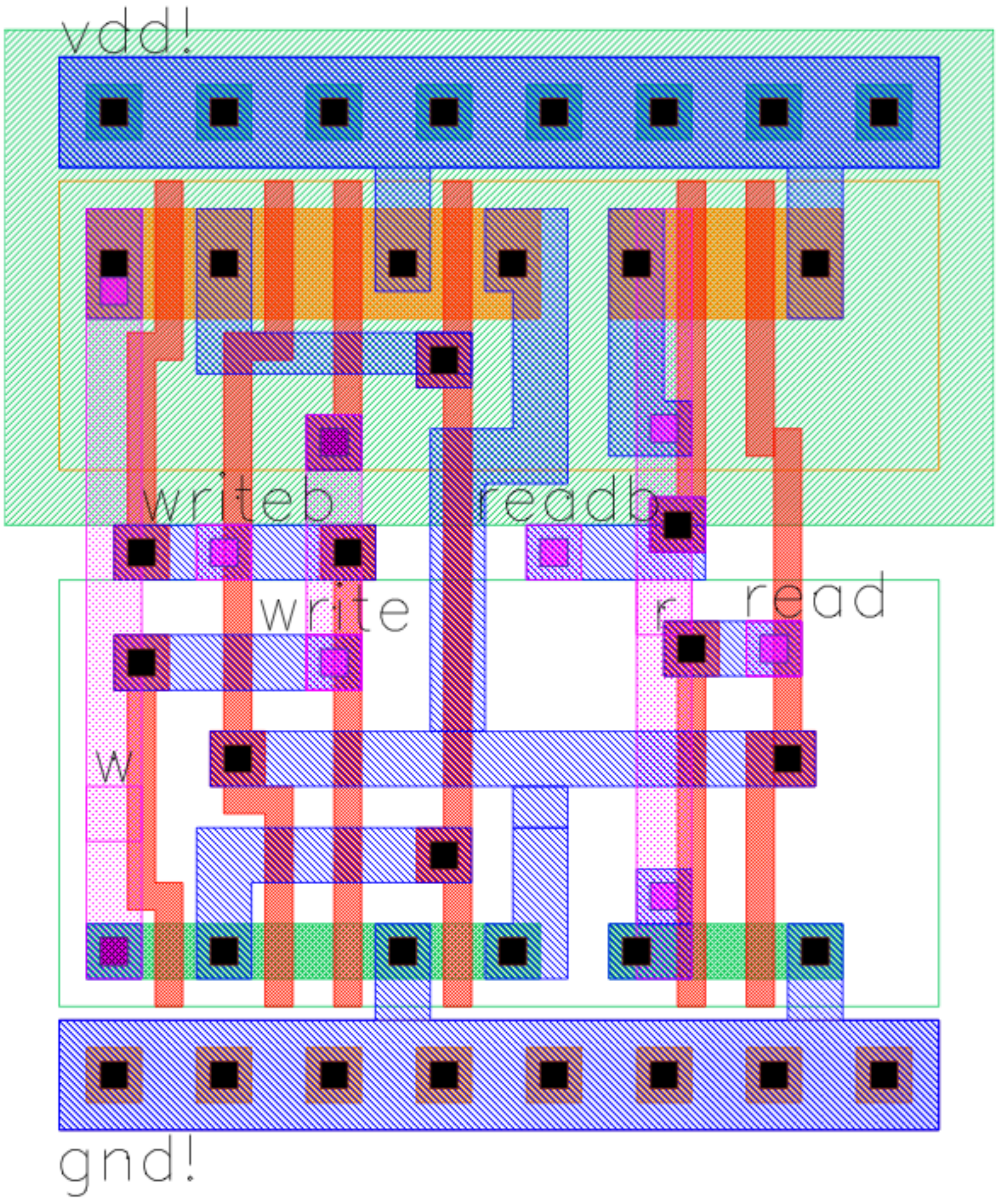
```
//row col    r4        r3        r2        r1        r0       p1 p2 win
00000_000000000_111111111_111111111_111111111_111111111_111111111_1_0_0
00001_000000001_111111111_111111111_111111111_111111111_111111110_0_1_0
00010_000000001_111111111_111111111_111111111_111111110_111111110_1_0_0
00100_000000001_111111111_111111111_111111110_111111110_111111110_0_1_0
01000_000000001_111111111_111111110_111111110_111111110_111111110_1_0_0
10000_000000010_111111100_111111100_111111100_111111100_111111100_0_1_0
01000_000000001_111111100_111111100_111111100_111111100_111111100_0_1_0
00100_000000010_111111100_111111100_111111100_111111100_111111100_0_1_0
00010_000000001_111111100_111111100_111111100_111111100_111111100_0_1_0
00100_000000010_111111100_111111100_111111100_111111100_111111100_0_1_0
00010_000000001_111111100_111111100_111111100_111111100_111111100_0_1_0
00100_000000001_111111100_111111100_111111100_111111100_111111100_0_1_0
01000_000000001_111111100_111111100_111111100_111111100_111111100_0_1_0
10000_000000010_111111100_111111100_111111100_111111100_111111100_0_1_0
10000_000000010_111111100_111111100_111111100_111111100_111111100_0_1_0
01000_000000100_111111100_111111000_111111000_111111000_111111000_1_0_0
01000_000000100_111111100_111111000_111111000_111111000_111111000_1_0_0
00100_000000100_111111100_111111000_111111000_111111000_111111000_1_0_0
01000_000000100_111111100_111111000_111111000_111111000_111111000_1_0_0
01000_000000100_111111100_111111000_111111000_111111000_111111000_1_0_0
10000_000000100_111111000_111111000_111111000_111111000_111111000_0_1_0
10000_000000100_111111000_111111000_111111000_111111000_111111000_0_1_0
10000_000000010_111111000_111111000_111111000_111111000_111111000_0_1_0
10000_000000010_111111000_111111000_111111000_111111000_111111000_0_1_0
01000_000000100_111111000_111111000_111111000_111111000_111111000_0_1_0
10000_000001000_111110000_111110000_111110000_111110000_111110000_1_0_0
10000_000010000_111100000_111100000_111100000_111100000_111100000_0_1_0
10000_000010000_111100000_111100000_111100000_111100000_111100000_0_1_0
01000_000001000_111100000_111100000_111100000_111100000_111100000_0_1_0
00100_000010000_111100000_111100000_111100000_111100000_111100000_0_1_0
00010_000100000_111100000_111100000_111100000_111000000_111000000_1_0_0
00010_000010000_111100000_111100000_111100000_111000000_111000000_1_0_0
00001_000100000_111100000_111100000_111100000_111000000_111000000_1_0_0
00001_000010000_111100000_111100000_111100000_111000000_111000000_1_0_0
00001_000100000_111100000_111100000_111100000_111000000_111000000_1_0_0
00001_001000000_111100000_111100000_111100000_111000000_110000000_0_1_0
00010_010000000_111100000_111100000_111100000_100000000_100000000_1_0_0
00001_100000000_111100000_111100000_111100000_100000000_000000000_0_1_0
00001_010000000_111100000_111100000_111100000_100000000_000000000_0_1_0
10000_100000000_000000000_000000000_000000000_000000000_000000000_1_0_1
```

3. Schematics and Layout

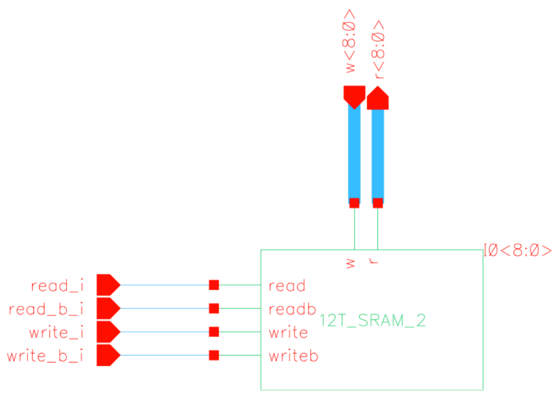
3.1. 12T SRAM

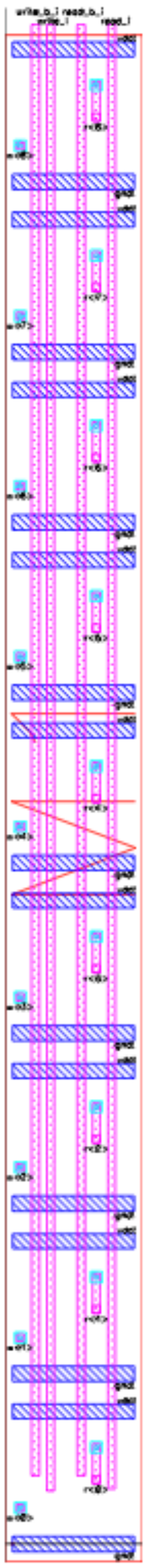
Schematic and layout based on professor's Harris Mips8 memory.

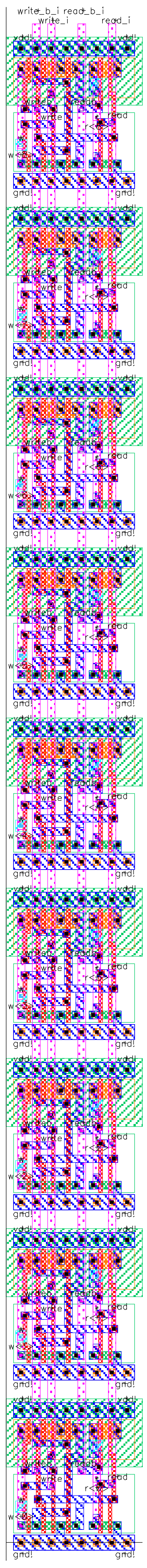



vdd!
writeb
readb
write
r
read
w
gnd!

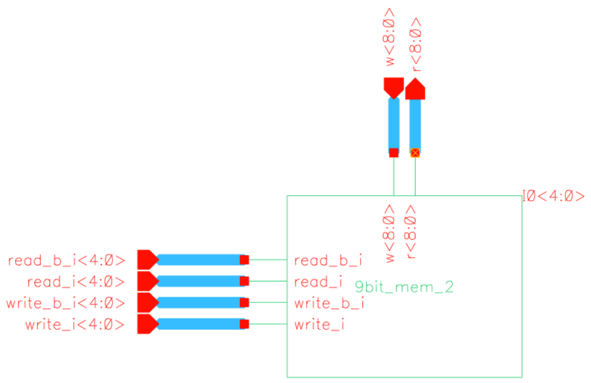
3.2. 9 bits memory

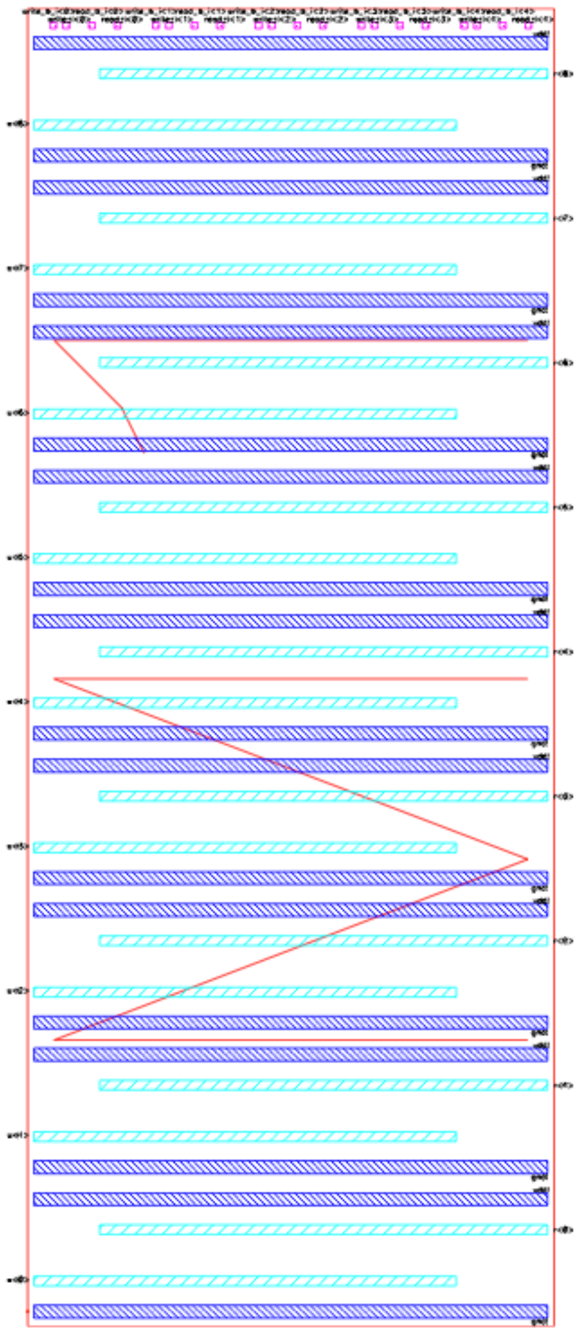


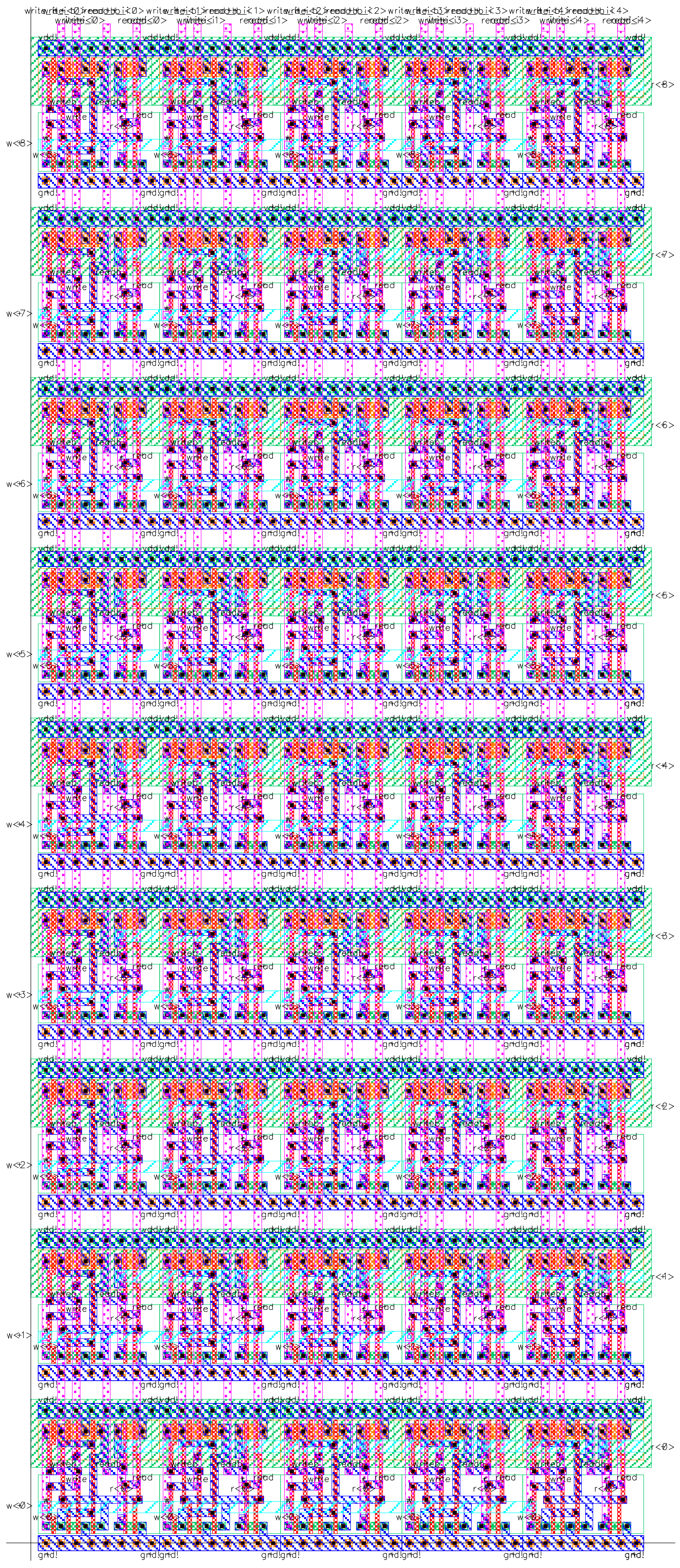



write_b_i
read_b_i
write_i
read_i
vdd!
writeb
readb
write
read
gnd!

3.3. 45 bits memory

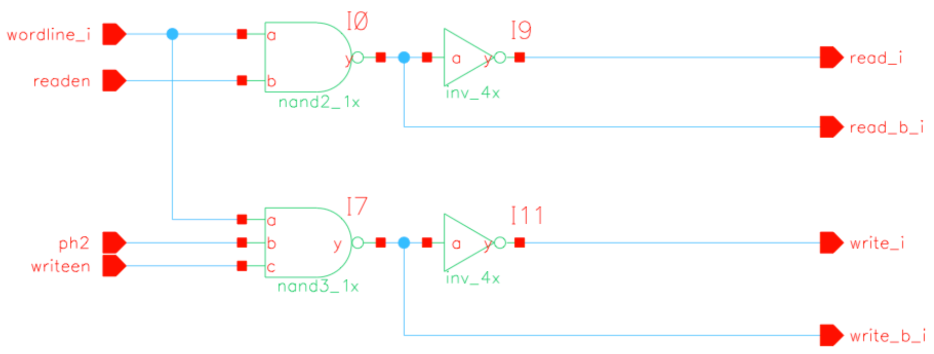


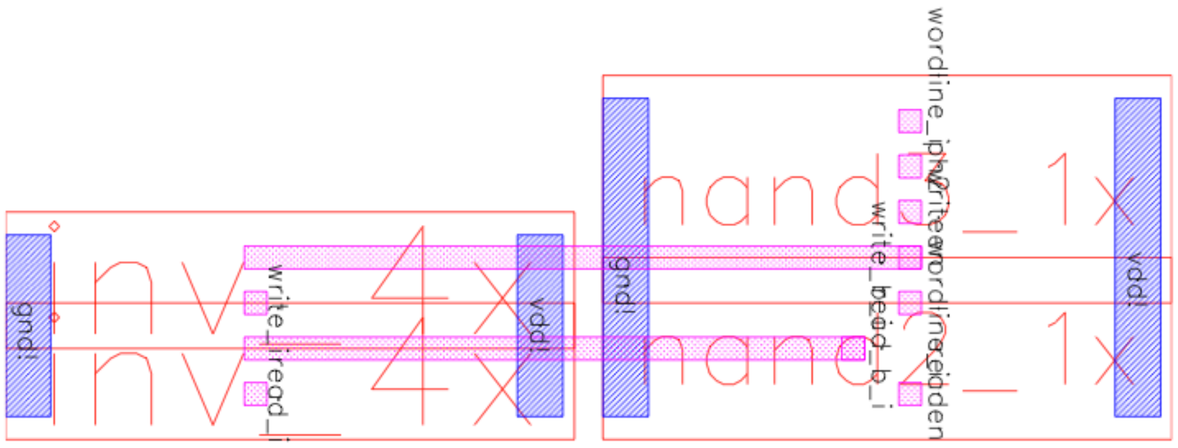


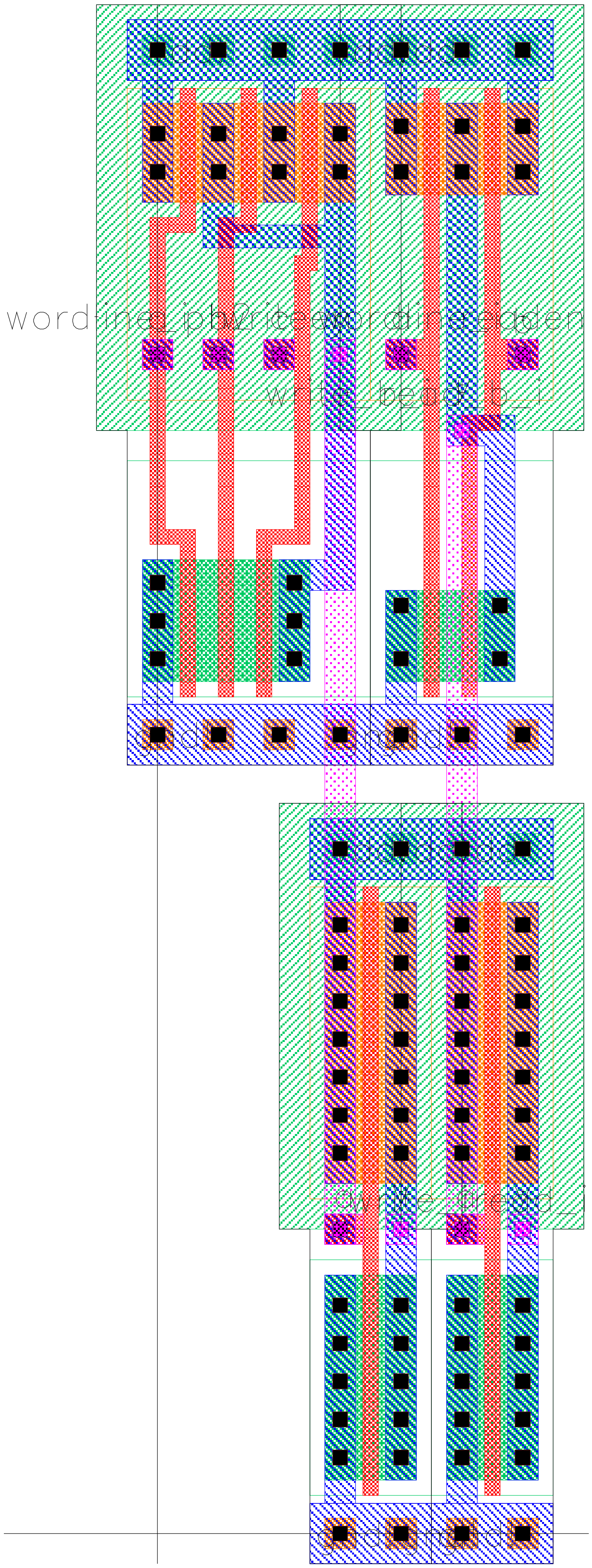


vdd!
gnd!
write
writeb
read
readb
w<8>
w<7>
w<6>
w<5>
w<4>
w<3>
w<2>
w<1>
w<0>
r<8>
r<7>
r<6>
r<5>
r<4>
r<3>
r<2>
r<1>
r<0>

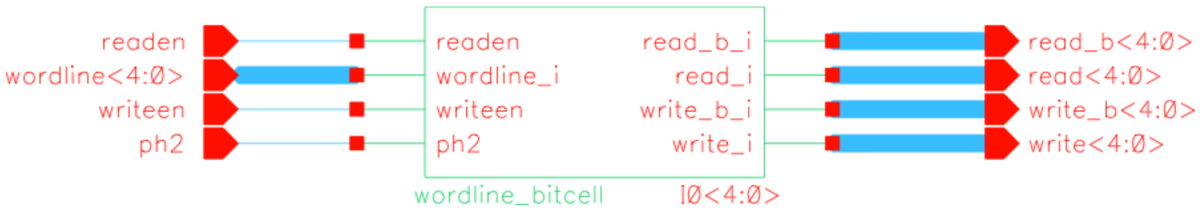
3.4. Wordline logic

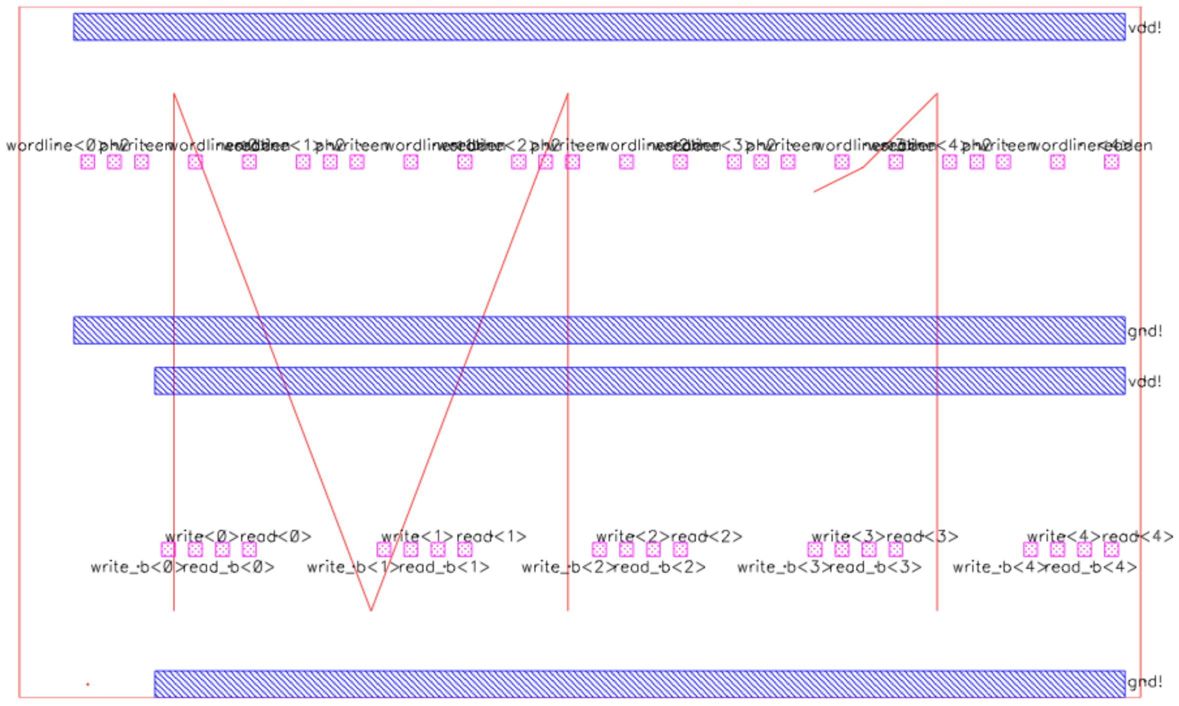


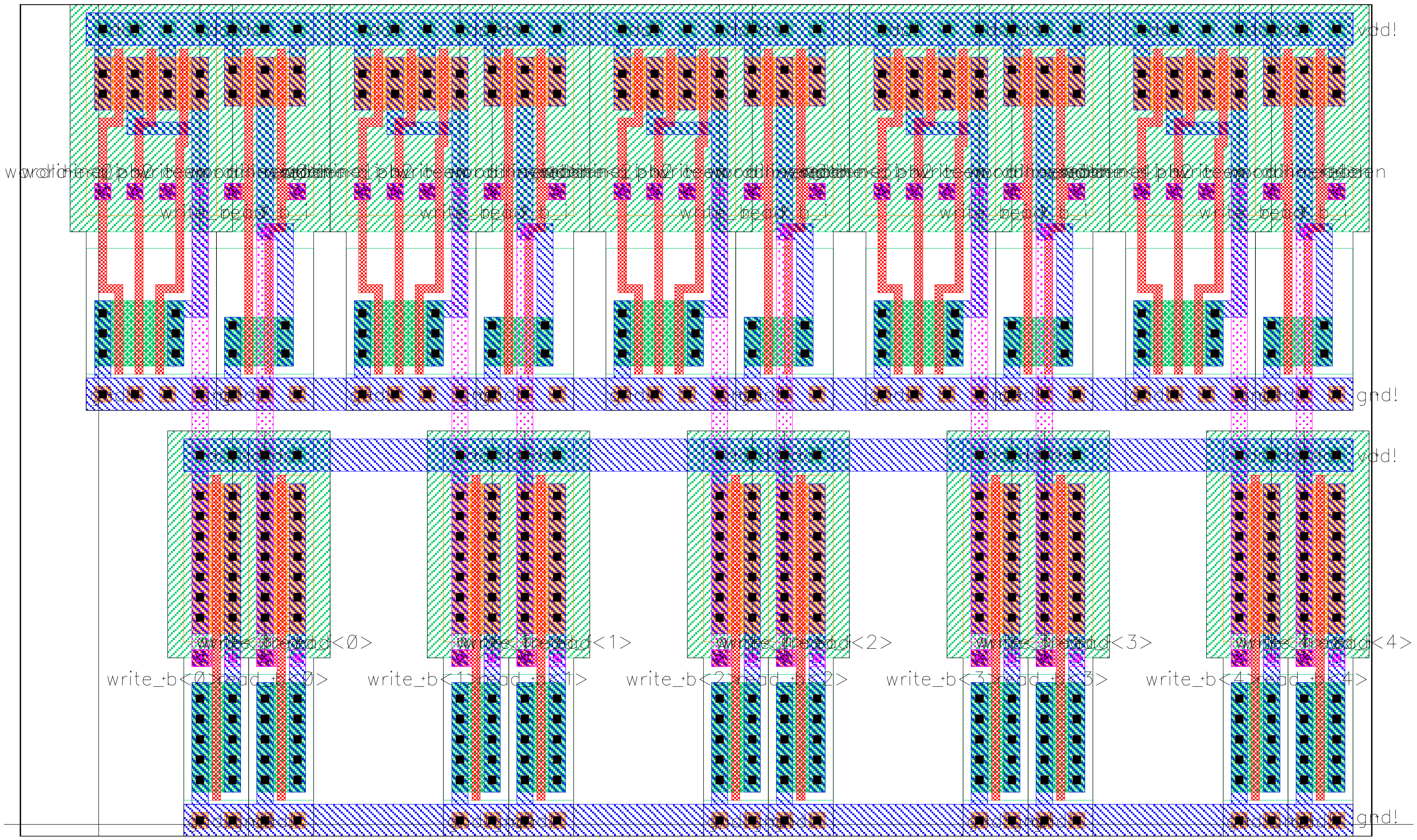




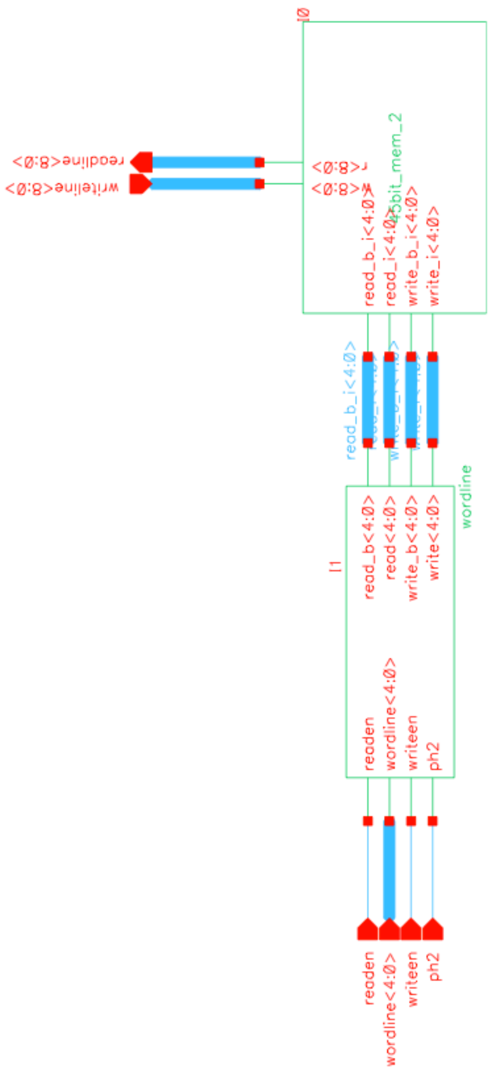
3.5. 5 bits Wordline logic

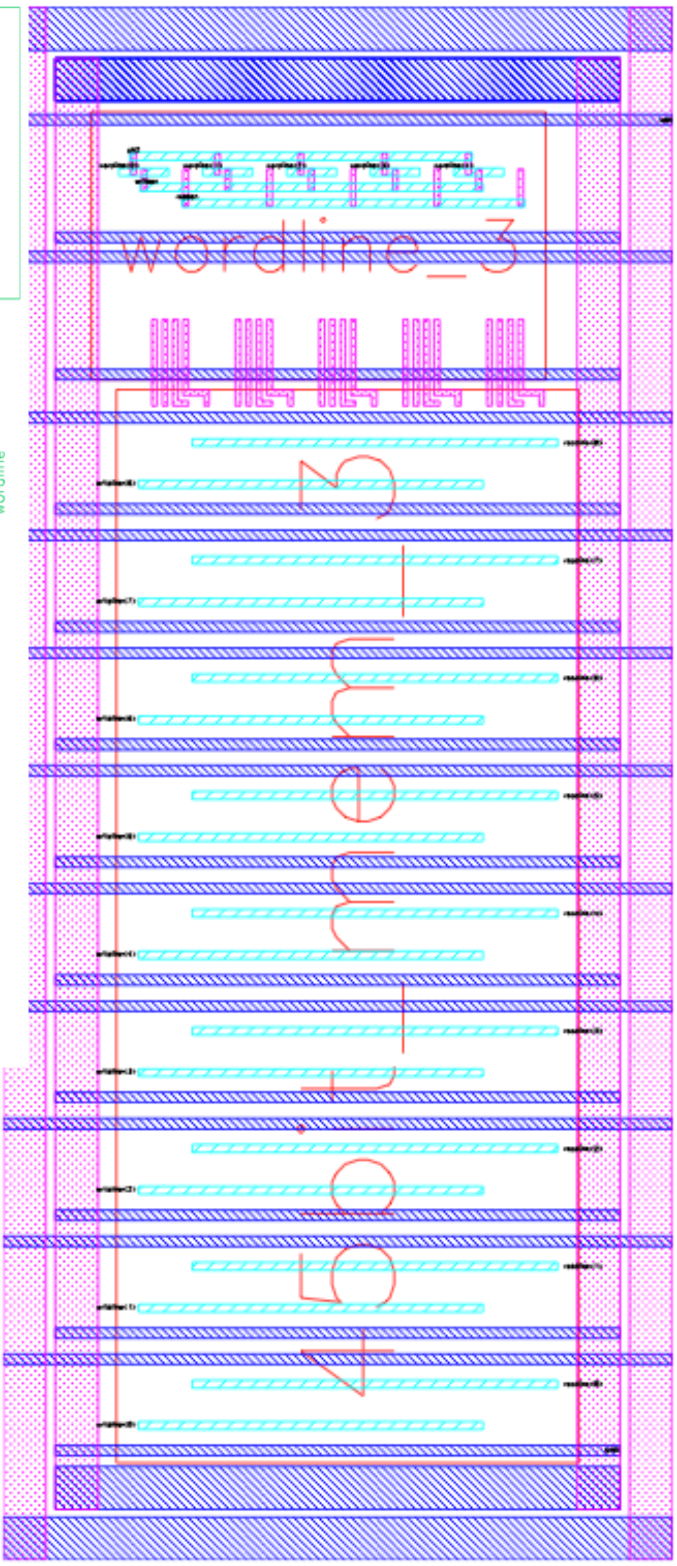


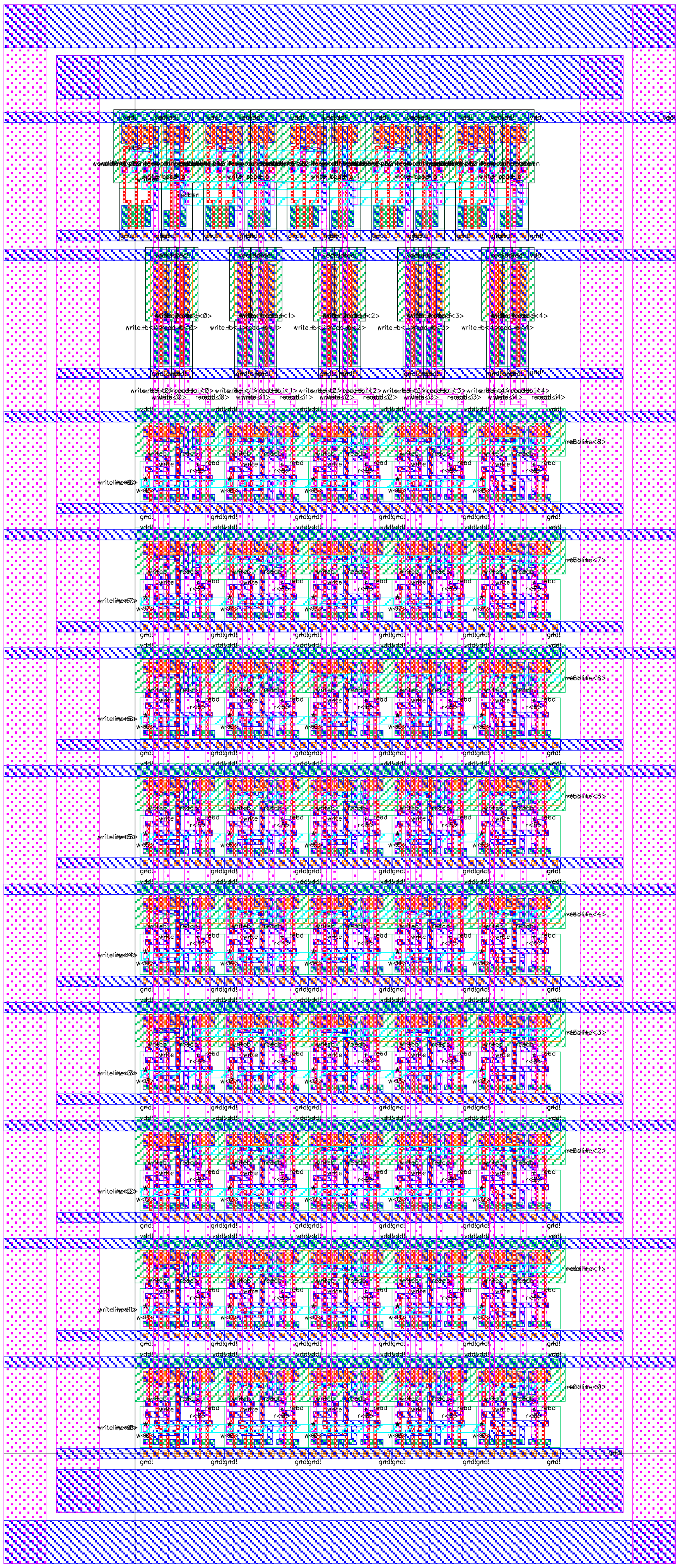



vdd!
gnd!
vdd!
gnd!
write_b<0> read_b<0>
write_b<1> read_b<1>
write_b<2> read_b<2>
write_b<3> read_b<3>
write_b<4> read_b<4>
write<0> read<0>
write<1> read<1>
write<2> read<2>
write<3> read<3>
write<4> read<4>
wordline_ph2 writeen readen

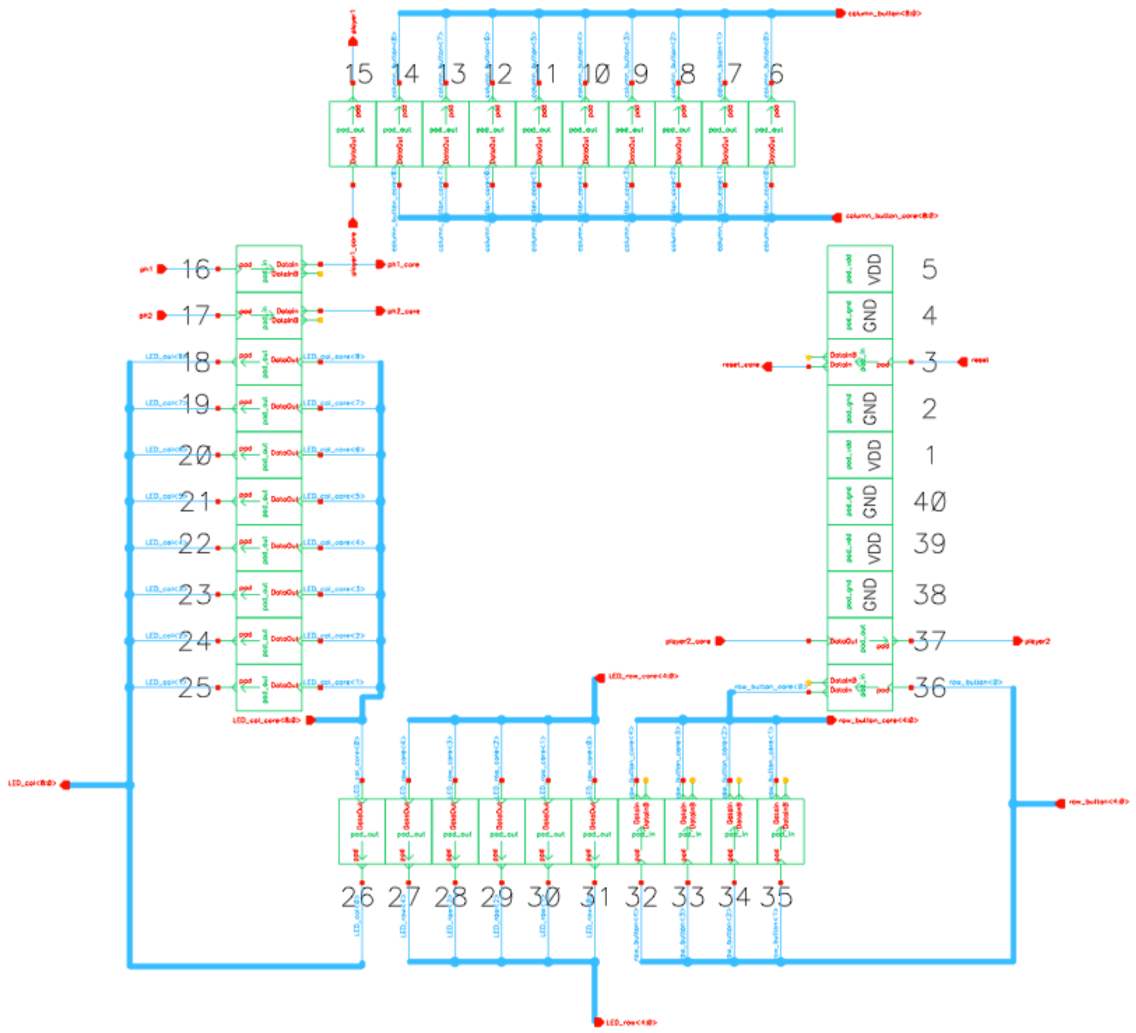
3.6. Full memory



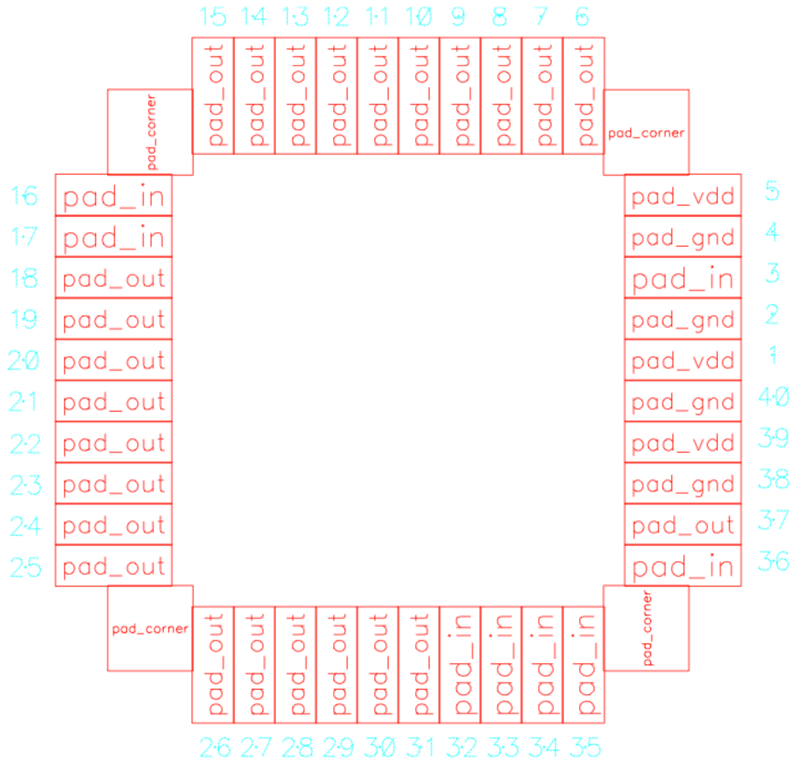



vdd!
gnd!
writeline<8>
writeline<7>
writeline<6>
writeline<5>
writeline<4>
writeline<3>
writeline<2>
writeline<1>
writeline<0>
readline<8>
readline<7>
readline<6>
readline<5>
readline<4>
readline<3>
readline<2>
readline<1>
readline<0>

3.7. Padframe

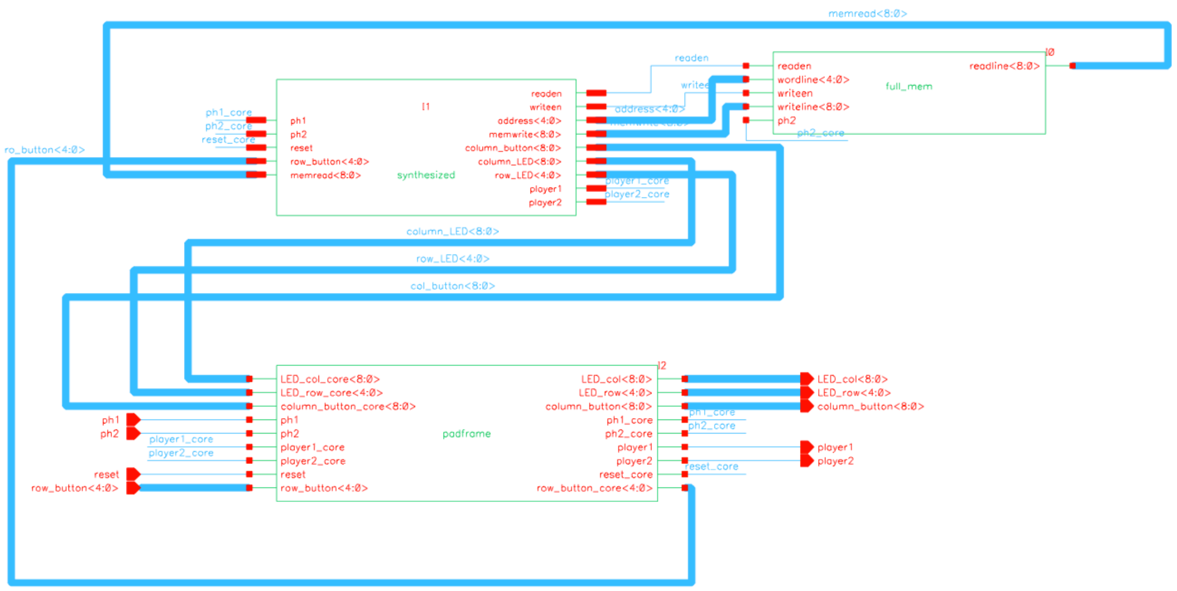


15 14 13 12 11 10 9 8 7 6
16
17
18
19
20
21
22
23
24
25
5 VDD
4 GND
3
2 GND
1 VDD
40 GND
39 VDD
38 GND
37
36
26 27 28 29 30 31 32 33 34 35



15 14 13 12 11 10 9 8 7 6
pad_out
pad_corner
pad_in
pad_vdd
pad_gnd
16 17 18 19 20 21 22 23 24 25
5 4 3 2 1 40 39 38 37 36
26 27 28 29 30 31 32 33 34 35

3.8. Chip


memread<8:0>
readen
write
address<4:0>
I0
readen
wordline<4:0>
writeen
writeline<8:0>
ph2
readline<8:0>
full_mem
ph2_core
I1
ph1_core
ph2_core
reset_core
ro_button<4:0>
ph1
ph2
reset
row_button<4:0>
memread<8:0>
synthesized
readen
writeen
address<4:0>
memwrite<8:0>
column_button<8:0>
column_LED<8:0>
row_LED<4:0>
player1
player2
player1_core
player2_core
column_LED<8:0>
row_LED<4:0>
col_button<8:0>
I2
padframe
LED_col_core<8:0>
LED_row_core<4:0>
column_button_core<8:0>
ph1
ph2
player1_core
player2_core
reset
row_button<4:0>
LED_col<8:0>
LED_row<4:0>
column_button<8:0>
ph1_core
ph2_core
player1
player2
reset_core
row_button_core<4:0>
LED_col<8:0>
LED_row<4:0>
column_button<8:0>
player1
player2
ph1
ph2
player1_core
player2_core
reset
row_button<4:0>
ph1_core
ph2_core
reset_core