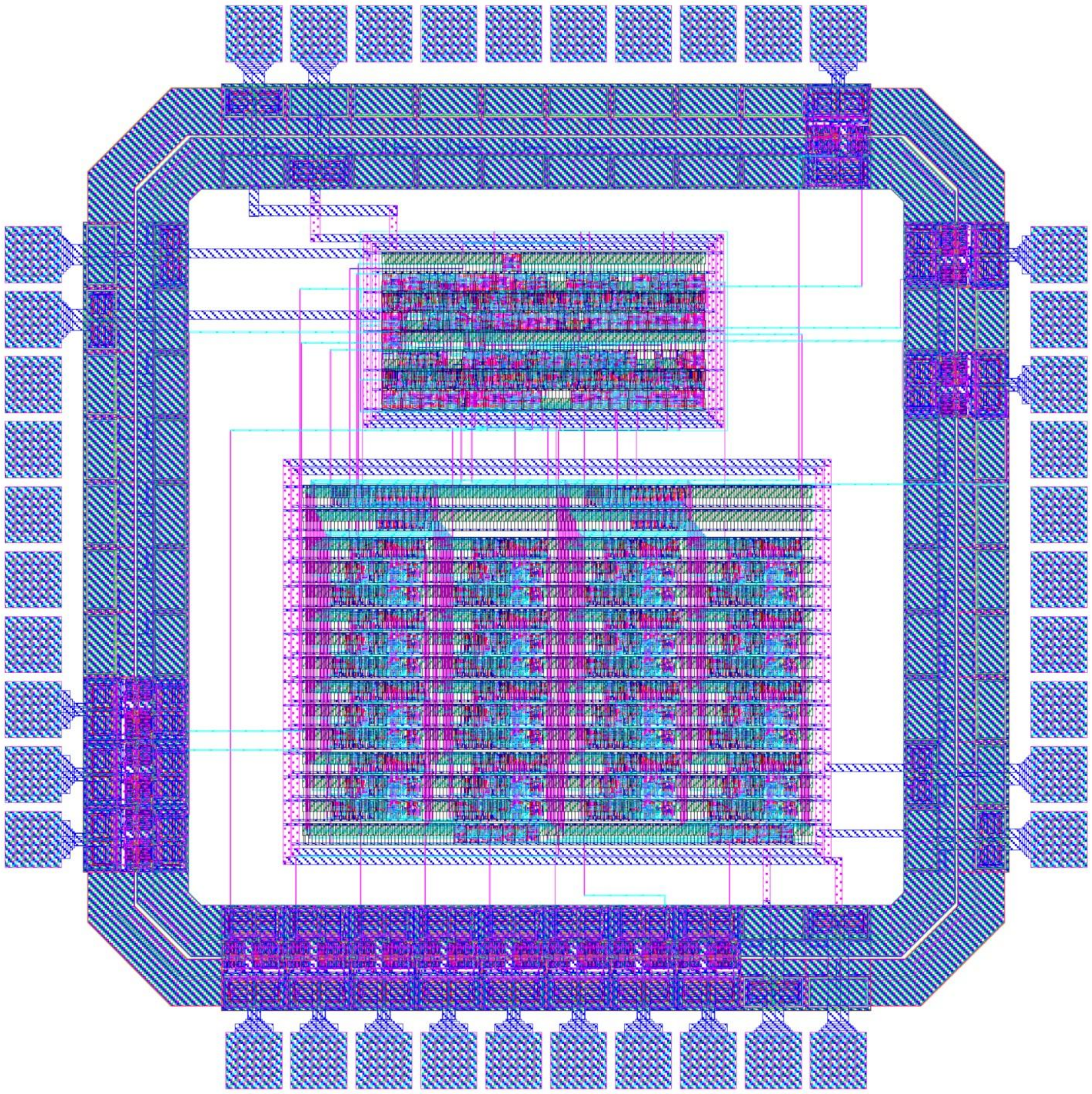
Simplified Checkers

Project Report



April 18, 2011

Max Korbel and Ian Jimenez

E158: Introduction to CMOS VLSI Design

Introduction

This project is a rule checker for “simplified checkers”. The rule checker was implemented in a .6 µm process on a 1.5 X 1.5 mm 40-pin MOSIS “TinyChip”. The goal of this project was to create a system which was able to recall the state of a game and then insure that any moves entered are correct based on the game state as it progresses. Simplified checkers rules were chosen over standard checkers rules due to concerns about design time and size constraints. We selected to not include forced jumps, double jumps, Kings, and checking the board state to see if a player has won. These modified rules ensured the design will fit in the “TinyChip.” The final result provides a resource for people seeking to learn to play checkers as well as for professional simplified checkers players (if our version ever catches on!) to confirm they are appropriately playing the game.

Functional Specifications

The chip is designed to take in two three bit binary numbers which are used to represent the row and column which a piece inhabits. The user then presses the enter button and enters the values for the desired final position. The numbering scheme of the game board is shown in Figure 1. Upon the next press of the enter button, the controller advances over a number of cycles through a finite state machine (Table 1) which is used to verify that the move the player selected is valid. The chip checks for single jumps and moves which are not jumps. As mentioned before, kings, forced available jumps, and double jumps are not legal in this game. If a move is invalid, the finite state machine enters an error state and an error light is activated and until the user pushes the enter button again. After pushing enter, the system is ready to accept a new move. If the clear button is pressed the turn will reset allow a user to pick a new piece. If the reset button is pressed, the entire game state is reset and the memory is cleared back to a new game set up. An LED array of lights displaying the current board state was considered, but it was decided that implementation would overcomplicate the design and make testing unnecessarily difficult. A list of ports and their intended uses is shown in Table 2.

State	**Logic**
0: Read Source	Read in the piece that the player wants to move. When the player presses enter, move on to Read Destination (1).
1: Read Destination	Read in the location that the player wants to move to.
2: Check Move	Check the move against rules of the game to make sure it is a valid move. If it’s not valid, go to Error (7). If it is valid, then we need to determine if it is a short move or a long move (jumping a piece). If it’s a short move, go to Write (4). Otherwise, if it’s a long move, we need to do some more logic and jump to Check Long Move (3).
3: Check Long Move	Checks if it is jumping the proper color piece and other small checks for the rules. If all is well, then continue to Clear Middle (6) to remove the piece we are jumping over from the board. Otherwise, go to Error (7).

4: Write	Writes the moved piece to its new location. Then continue to Clear (5).
5: Clear	Erase the piece from where it used to be before it moved. Then continue to Done (8).
6: Clear Middle	Remove the piece we are jumping over from the board, then continue to Write (4).
7: Error	Something bad happened. Maybe a broken rule. Light up the error light and wait for enter, then if enter is pressed go back to the beginning of the turn Read Source (0).
8: Done	We are now done with this turn. Change the turn to the opposite color and move back to Read Source (0) to get ready for the next move input from the opposite player.

Table 1: Controller finite state machine logic

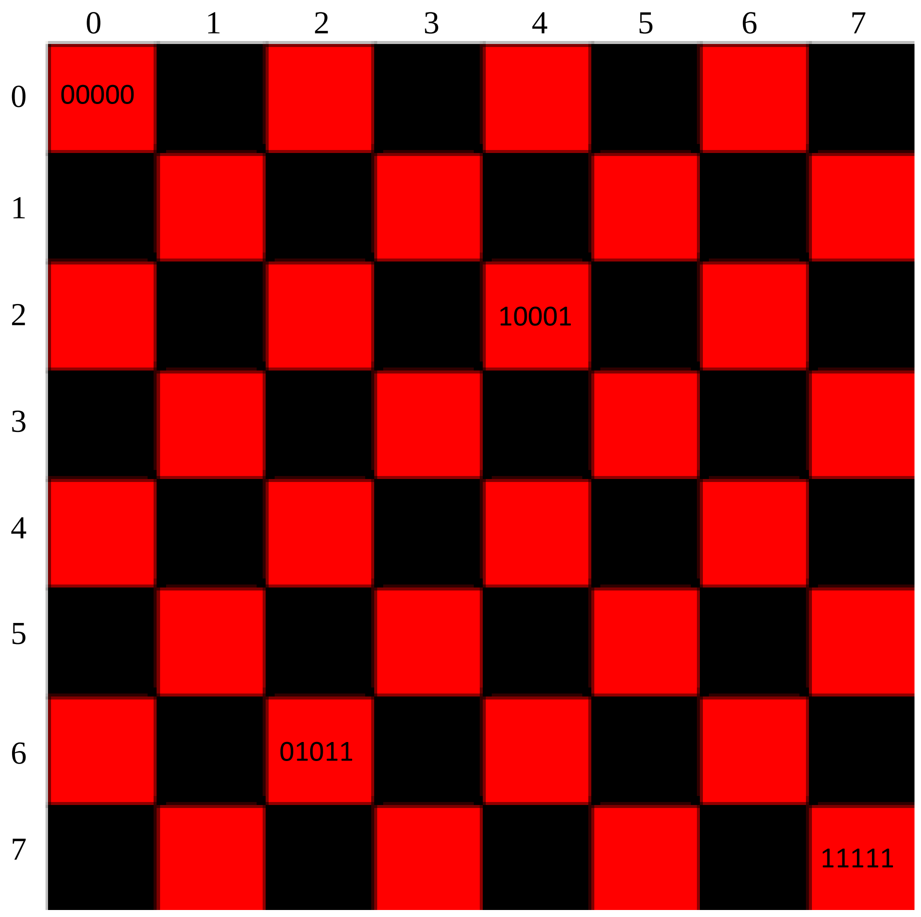


Figure 1: The game board.
Shows the number scheme for the board as well as a few examples of addresses in memory. The board is sideways, so rows go vertically and columns are horizontal. Red starts on the left three rows of the board while black starts on the right three rows.

Num Pins	**Name(Type)**	**Description**
3	3 Vdd (Input/Output)	
3	3 GND (Input/Output)	
3	3 Usr-Row (Input)	Accepts user's row input to move pieces
3	3 Usr-Col (Input)	Accepts user's column input to move pieces
1	Red turn indicator (Output)	Indicates it is red's turn
1	Black turn indicator (Output)	Indicates it is black's turn
1	Error (Output)	Indicates error with move placement
1	Reset(Input)	Restarts board state to a new game
1	Enter(Input)	Indicates that entry of position complete
1	Clear (Input)	Clears entry of active piece on turn
1	Phi 1 (Input)	Non-overlapping clock (Phase 1)
1	Phi 2 (Input)	Non-overlapping clock (Phase 2)
20 pins		

Table 2: Pin out. Locations for each pin on padframe shown in Figure 3.

Floor Plan

We have two major portions of the design. The first is mem which is the main memory used to store the positions of all pieces on the board. In order to know the three possible states of a position we needed 64 bits of information. The memory was therefore organized into two 32:1 SRAM arrays with 5 bit addresses for either red or black pieces on the board. Each of the two arrays was organized into eight 4-cell blocks to make layout simpler with an 8:1 multiplexer to give the correct output. Remember, half the board does not require a space in memory because you cannot move onto black squares. The controller was synthesized and placed above the memory cell (see Appendix 5 for actual layout).

Since the time of the proposal we have undergone some changes to the overall scheme incorporating the two 5:1 flip-flops in to the FSM and absorbing the 64:1 multiplexer into the memory array (turned into SRAM instead of a large array of flip-flops). Even taking these into account the memory array it is about 6 times the size we anticipated (see Figure 2). One reason is that while the basis for sizing was the MIPS processor, our memory incorporates logic specific to the simplified checkers rules. For example, some cells were made resettable and others settable to allow for initial game state to have pieces in appropriate starting positions. In addition, zipper logic is non-optimal to decrease design time to a make certain it would be possible to finish the layout in time. We could have reduced the size and increased the density of the blocks by more efficiently placing and routing since a large amount of space was taken with filler in order to maintain continuity of the power and ground rows through the blocks. Some issues were not reviewed since they would have required significant amounts of redesign and the chosen final design still easily fits within the bounds of the pad frame. A tradeoff was made between design time and space utilization.

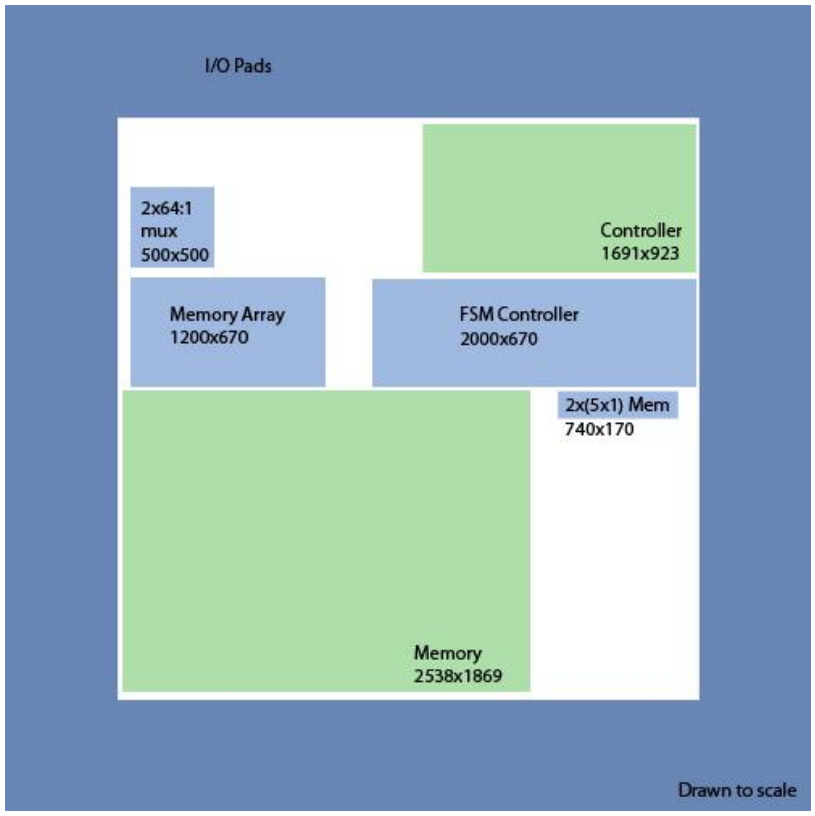


Figure 2: Layout sizing (drawn to scale).
Light blue is proposal approximations of sizes while light green is sizes for the final layout.

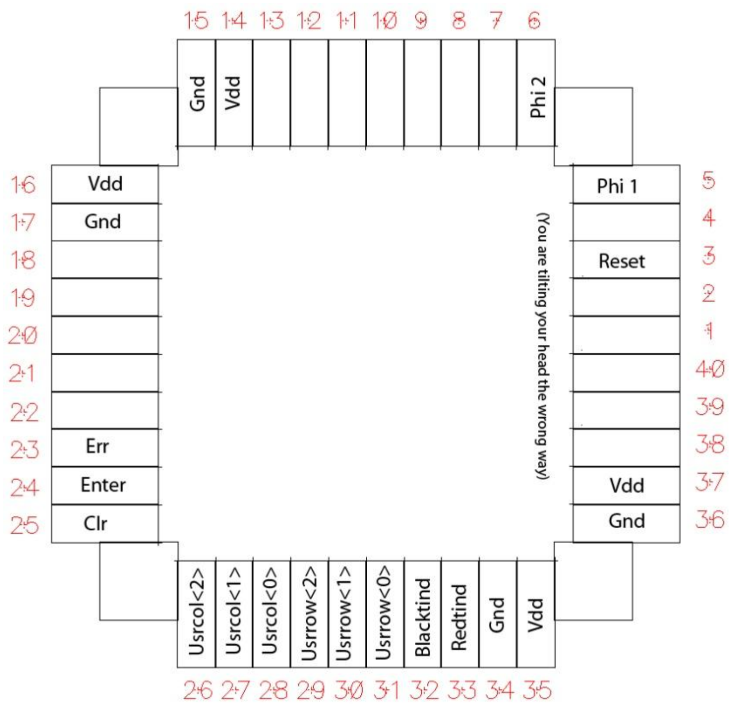


Figure 3: Pin – Pin Number correspondence pin out diagram

Verification

The Verilog code for our checkers chip passes our self checking testbench (Appendix 2) using our compelling test vectors (Appendix 3) which are explained in Table 4. Both the synthesized controller schematic and the handmade memory for the chip pass this same testbench on the same set of test vectors. The layout passes DRC, showing that our chip, if manufactured, would not have any physical spacing flaws that would cause definite malfunctions. The layout also passes LVS, which means that it matches our schematic and should therefore function the same way as the schematic does. There were no analog blocks so HSPICE simulations were not necessary.

Post-fabrication test plan

If this chip were to be fabricated, it would be quite easy to test. The only outputs are the turn indicators and the error light and the only inputs are the clocks, clear, error, enter, and the binary location selector (6 bits). One could use the same test vectors as used in the self checking testbench during verification for the chip. Another good test would be to play through a game of simplified checkers and make sure that the rule checks are valid at each step.

Test Vector	**Turn**	**Source**	**Dest**	**Description**	**Error Light**
0	Red	(2,0)	(3,1)	Move one square legally	OFF
1	Black	(5,3)	(4,2)	Move one square legally	OFF
2	Red	(2,6)	(3, 5)	Move one square legally	OFF
3	Black	(4,2)	(2,0)	Jump over first red piece moved and capture it	OFF
4	Red	(3,5)	(2,6)	Illegally try to move backwards to original location	ON
5	Red	(3,5)		Press "clear" mid entry	OFF
6	Red			Press "reset" before entry	OFF
7	Red	(0,6)	(1,5)	Illegally move into own piece 1 square	ON
8	Red	(1,5)	(3,7)	Illegally jump your own piece	ON
9	Red	(2,4)	(3,5)	Move one square legally	OFF
10	Black	(5,1)	(2,4)	Illegally move 3 squares (to empty one)	ON
11	Black	(5,5)	(4,4)	Move one square legally	OFF
12	Red	(2,0)	(3,1)	Move one square legally	OFF
13	Black	(4,4)	(3,5)	Illegally move into an enemy	ON

14	Black	(4,4)	(2,6)	Illegally jump over enemy into an enemy	ON
15	Black	(7,7)	(4,0)	Illegally move like a idiot (far away, non-diagonal, completely illegal)	ON
16	Black	(5,2)		Illegal src selection	ON
17	Black	(5,3)	(5,2)	Illegal dest selection	ON
18	Black	(4,4)	(5,5)	Illegally try to move backwards	ON
19	Black	(6,6)	(5,5)	Legally move where another piece has been and left	OFF
20	Red	(7,1)		Illegally try to select enemy piece	ON

Table 4: Test vectors explanation. Actual test vectors in Appendix 3.

Design Time

A summary of design time for different components of the project is shown in Table 5.

Project Proposal	2.5 hours
Verilog	35 hours
Schematic	25 hours
Memory redesign	10 hours
Layout	40 hours
Total	**~112.5 hours**

Table 5: Time spent on this project.

File Locations

All files are located on chips.eng.hmc.edu in the locations listed in Table 6. Many of these files can also be found in the appendices.

Item	**Location**
Verilog code (memory)	/home/mkorbel/VLSI/Final_Project/verilog/checkersmem.sv
Verilog code (controller)	/home/mkorbel/VLSI/Final_Project/verilog/controller.sv
Verilog code (main)	/home/mkorbel/VLSI/Final_Project/verilog/checkers.sv
Testbench	/home/mkorbel/VLSI/Final_Project/chip.template
Test vectors	/home/mkorbel/VLSI/Final_Project/checkers.tv
Synthesis results	/home/mkorbel/VLSI/Final_Project/synthesis
All Cadence libraries	/home/mkorbel/VLSI/IC_CAD/cadence/checkers
CIF	/home/mkorbel/VLSI/IC_CAD/cadence/checkers_cifin
PDF of chip plot	/home/mkorbel/VLSI/Final_Project/chip.pdf
PDF of this report	/home/mkorbel/VLSI/Final_Project/checkers_korbel_jimenez.pdf

Table 6: List of files and their locations

Appendices

Appendix 1: Verilog code

```
//-----------------------------------------------
// checkers.sv
// Max Korbel and Ian Jimenez
// April 18, 2011
// VLSI Final Project: Simplified Checkers
// The main file for checkers
//-----------------------------------------------

`include "checkersmem.sv"
`include "controller.sv"

module checkers(input  logic            phi1, phi2, reset,
                           input  logic        enter, clr,
                           input  logic [2:0]  usrrow, usrcol,
                           output logic        blacktind, redtind,
                           output logic        err);


       logic         memwrite;
       logic [4:0]   addr;
       logic [1:0]   datain;
       logic         turn;


       //Creates the controller for the system
       controller c(phi1,phi2,reset,usrrow,usrcol,
                  enter,clr,r,b,err,memwrite,addr,datain,turn);

       //Creates memory block
       mem gamemem(phi2,reset, memwrite,addr,datain,r,b);

endmodule
```

```
//-----------------------------------------------
// checkersmem.sv
// Max Korbel and Ian Jimenez
// April 18, 2011
// VLSI Final Project: Simplified Checkers
// This file is the memory for checkers.sv
//-----------------------------------------------

//datain:
//     00 -> nothing in square
//     01 -> black in square
//     10 -> red in square

module mem (input   logic        clk, reset,
                    input  logic        memwrite,
                    input  logic [4:0]  addr,
                    input  logic [1:0] datain,
                    output logic        r, b);

       // instantiate the memory arrays
       logic  [31:0]       RAMblack, RAMred;

       always_ff @(posedge clk or posedge reset)
              begin
                     if (reset) begin
                            // set the two memories to start with initial piece
locations
                            RAMblack<=32'b11111111111100000000000000000000;
                            RAMred  <=32'b00000000000000000000111111111111;
                     end
                     else if (memwrite) begin // writing
                            RAMblack[addr[4:0]] <= datain[0];
                            RAMred[addr[4:0]] <= datain[1];
                     end
                     else begin // reading
                            r <= RAMred[addr[4:0]];
                            b <= RAMblack[addr[4:0]];
                     end
              end

endmodule
```

```
//-----------------------------------------------
// controller.sv
// Max Korbel and Ian Jimenez
// April 18, 2011
// VLSI Final Project: Simplified Checkers
// Serves as controller for checkersmem and  as rule checker
//-----------------------------------------------

module controller(  input  logic               phi1,phi2, reset,
                    input  logic [2:0]         usrrow, usrcol,
                    input  logic               enter, clr,r,b,
                    output logic               err,
                    output logic               memwrite,
                    output logic [4:0]         addr,
                    output logic [1:0]         datain,
                    output logic               turn);

       //Declarations
       logic [5:0]  src, dest;          //Signal from Source and Destination
       logic [1:0]  distance, adrctrl;  //represents distance of move
       logic        storesrc, storedst; //controls when info is stored
       logic [5:0]  mid;                //represents middle peice
       logic [3:0]  state, nextstate;   //the current state and the next state
       logic [6:0]  controls;           //bus holding a bunch of control signals
       logic [5:0]  longloc;            //concatenated row and column
       logic [5:0]  decinp;             //a source or destination to be passed
       logic [5:0]  regnum;             //values from src or dest
       logic        turnen;             //indicates end or turn changing turn
       logic        fsmreset;           //combins clr and reset

       //For Turn
       // 0 red
       // 1 black
       assign blacktind = turn;
       assign redtind   = ~turn;

       assign longloc = {usrrow, usrcol}; //concatenation of the row and column
       assign fsmreset = (reset | clr);  //if the FSM should reset



       //STATES
       parameter    RDSOURCE     = 4'b0000; // Read the source
       parameter    RDDEST       = 4'b0001; // Read the dest, check the piece at loc
       parameter    CHKMOVE      = 4'b0010; // Checks if move is open and gets dist
       parameter    CHKLONG      = 4'b0011; // If its skipping, check more stuff
       parameter    WRITE        = 4'b0100; // Done, write everything
       parameter    CLEAR        = 4'b0101; // Done, now clear old
       parameter    CLEARMID     = 4'b0110; // Removes middle piece during jump
       parameter    ERR          = 4'b0111; // If there is an error anywhere
       parameter    DONE         = 4'b1000; // Change the turn and go back to 0

       //DISTS
       parameter    ZERODIST     = 2'b00;     // There was an error checking the move
       parameter    ONEDIST      = 2'b01;     // Moving one square
       parameter    TWODIST      = 2'b10;     // Jumping
```

```
always_comb
       case(state)
              RDSOURCE:    begin
              // Checks if a vaild source on board (odd odd or even even)
                     if(enter & ~clr) begin
                            if (~(usrrow[0] ^ usrcol[0]))
                                   nextstate <= RDDEST;
                            else
                                   nextstate <= ERR;
                     end
                     else
                            nextstate<=RDSOURCE;
              end
              RDDEST: //checks jump is over enemy
                     if (b == turn & r == ~turn) begin
                     // Checks if a vaild dest on board (odd odd or even even)
                            if(enter) begin
                                   if (~(usrrow[0] ^ usrcol[0]))
                                          nextstate <= CHKMOVE;
                                   else
                                          nextstate <= ERR;
                            end
                            else
                                   nextstate<=RDDEST;
                     end
                     else nextstate<= ERR;
              CHKMOVE:
                     if (b == 0 & r == 0)begin // Checks moving own piece
                       case(distance)
                         ZERODIST:nextstate <= ERR;     // some kind of error
                         ONEDIST: nextstate <= WRITE;   // single space move
                         TWODIST: nextstate <= CHKLONG; // jump attempt
                         default: nextstate <= ERR;
                       endcase
                     end
                     else nextstate <= ERR;
              CLEARMID:
                     nextstate <= WRITE; //erase the piece we're jumping over
              CHKLONG:
                     begin
                            if(r == turn & b == ~turn) nextstate <= CLEARMID;
                            else nextstate <= ERR;
                     end
              WRITE:        nextstate <= CLEAR;
              CLEAR:        nextstate <= DONE;
              DONE:         nextstate <= RDSOURCE;
              ERR:          begin
                                   if(enter) nextstate <= RDSOURCE;
                                   else nextstate <= ERR;
                            end
              default:      nextstate <= RDSOURCE;
       endcase

// set up the control signals
assign {storesrc, storedst, memwrite, adrctrl, turnen, err} = controls;

getdistandmid check(src, dest, turn, distance, mid);

// set appropriate control signals
always_comb
       case(state)
              RDSOURCE:     controls <= 7'b1_0_0_00_0_0;
              RDDEST:       controls <= 7'b0_1_0_00_0_0;
```

```
                CHKMOVE:      controls <= 7'b0_0_0_01_0_0;
                CHKLONG:      controls <= 7'b0_0_0_10_0_0;
                WRITE:        begin
                                    controls <= 7'b0_0_1_01_0_0;
                                    datain <= {~turn,turn};
                              end
                CLEAR:        begin
                                    controls <= 7'b0_0_1_00_0_0;
                                    datain <= 2'b00;
                              end
                CLEARMID:     begin
                                    controls <= 7'b0_0_1_10_0_0;
                                    datain <=2'b00;
                              end
                DONE:         controls <= 7'b0_0_0_00_1_0;
                ERR:          controls <= 7'b0_0_0_00_0_1;
                default:      begin
                                    controls <= 7'bx_x_x_xx_x_x;
                                    datain <= 2'bxx;
                              end
          endcase

    flopenr #(6)  srcreg(phi1, phi2 , reset, storesrc, longloc, src);
    flopenr #(6)  destreg(phi1, phi2, reset, storedst, longloc, dest);

    // Selects if dest or src will be sent to
    mux_2_1 #(6)  selmemmux(dest, src, adrctrl[0], regnum);

    // address for memory gets set
    assign addr = {decinp[5:3], decinp[2:1]};

    // Selects if selmemmux or ctrlloc will be sent to mem
    mux_2_1 #(6)  contrmemmux(mid,regnum, adrctrl[1], decinp);

    // keeps track of turn stuff
    flopenr #(1)  turnflop(phi1, phi2, reset, turnen, ~turn, turn);

    //stores nextstate logic
    flopenr #(4) nextstateflop(phi1, phi2, fsmreset,1'b1,nextstate, state);


endmodule


module getdistandmid (input logic [5:0] src, dest,// dont forget, these go (row, col)
                      input       turn,  // dont forget, turn is 0 for red, 1 for black
                      output      logic [1:0]  distance,
                      output      logic [5:0] mid);
    logic [5:0]         longdist;
    always_comb begin
          //if row or col didnt change
          //if row or col changed more than 2
          //if row didnt change exactly as much as col
          if(src[2:0] == dest[2:0] | src[5:3] == dest[5:3]|
                longdist[2:0] > 2'b10 | longdist[5:3] > 2'b10 |
                ~(longdist[2:0] == longdist[5:3]))
                distance <= 0;      //if row didnt change exactly as much as col
          else distance <= longdist[5:3];

          if(src[2:0] < dest[2:0]) begin // if src col < dest col (OK for either)
                mid[2:0] = src[2:0] + 1;
                longdist[2:0] <= dest[2:0] - src[2:0];
          end
```

```
            else if(turn) begin // if src col > dest col (OK for either)
                  mid[2:0] = src[2:0] - 1;
                  longdist[2:0] <= src[2:0] - dest[2:0];
            end

            if(src[5:3]<dest[5:3] & ~turn) begin// if src row < dest row (OK for red)
                  mid[5:3] = src[5:3] + 1;
                  longdist[5:3] <= dest[5:3] - src[5:3];
            end
            else if(turn) begin       // if src row > dest row (OK for black)
                  mid[5:3] = src[5:3] - 1;
                  longdist[5:3] <= src[5:3] - dest[5:3];
            end
            else   longdist[5:3] <= 0; // this will cause distance = 0, aka an error
      end
endmodule

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

  always_latch
    if (ph) q <= d;
endmodule

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

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

  logic [WIDTH-1:0] mid;

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

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

  logic [WIDTH-1:0] d2;

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

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

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

module mux_2_1 #(parameter width=1)
                           (input  logic [width-1:0] A,
                            input  logic [width-1:0] B,
                            input  logic ctrl,
                            output logic [width-1:0] out);
       always_comb
             if (ctrl) out <= A;
             else out <= B;
endmodule

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

  assign y = s ? d1 : d0;
endmodule
```

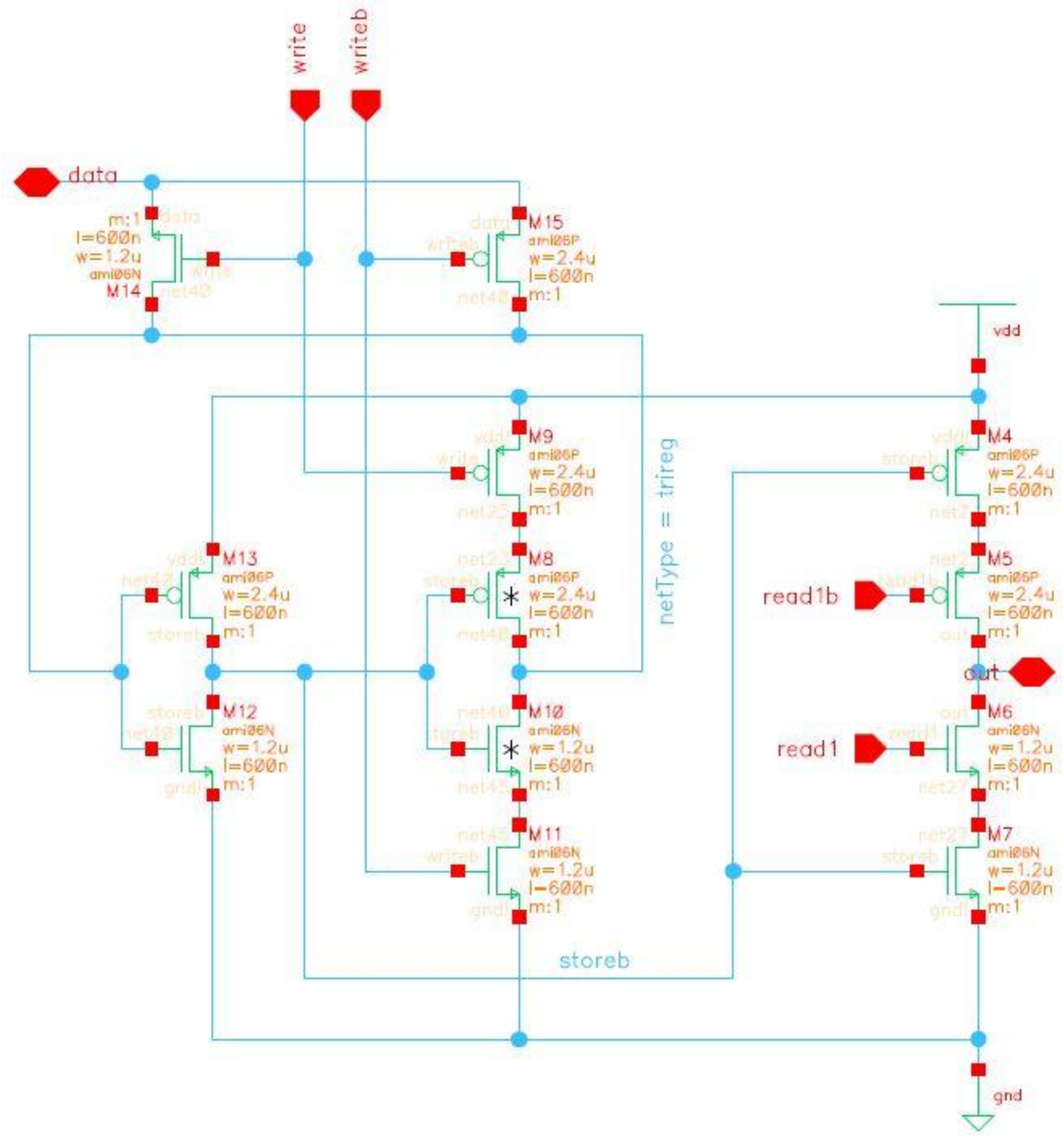
Appendix 2: Self-checking Testbench for the chip

```
//------------------------------------------------
// chip.template
// Max Korbel and Ian Jimenez
// April 18, 2011
// VLSI Final Project: Simplified Checkers
// Testbench for the chip
//------------------------------------------------


`timescale 1ns / 100ps

module test;



// 9 bits of input
logic [2:0]  usrrow,usrcol;
logic reset, enter, clr;
logic phi1,phi2;

// 3 bits of output that matter
reg [7:0]  rowcontrol;
reg blacktind, redtind, err;


chip game ( blacktind, err, redtind, clr, enter, phi1, phi2, reset,
     usrcol, usrrow );


    logic [11:0] vectors[200:0], currentvec;
    logic [12:0] vectornum, errors;

    // read test vector file and initialize test
    initial begin
       $readmemb("checkers.tv", vectors);
       vectornum = 0; errors = 0;
    end
    // generate a clock to sequence tests
    always begin
      phi1 = 0;  phi2 = 0; #5;
      phi1 = 1;  phi2 = 0; #5;
      phi1 = 0;  phi2 = 0; #5;
      phi1 = 0;  phi2 = 1; #5;
    end
    // apply test
    always @(posedge phi1) begin
       currentvec = vectors[vectornum];
       usrrow = currentvec[8:6];
       usrcol = currentvec[5:3];
       enter  = currentvec [2] ;
       clr  = currentvec [1] ;
       reset  = currentvec [0] ;
       if (currentvec[0] === 1'bx) begin
        $display("Test completed with %d errors", errors);
      $stop;

       end
    end
    // check if test was sucessful and apply next one
    always @(negedge phi1) begin
       if (err !== currentvec[11] | blacktind !== currentvec[9] |
           redtind !==currentvec[10]) begin
              errors = errors + 1;
          $display("Error: Vectornum =%d ", vectornum);
          $display("       output mismatches at err: %b,turn: %b   (%b, %b expected)",
                 err, {redtind, blacktind},currentvec[11], currentvec[10:9]);
      end
       vectornum = vectornum + 1;

    end

endmodule
```

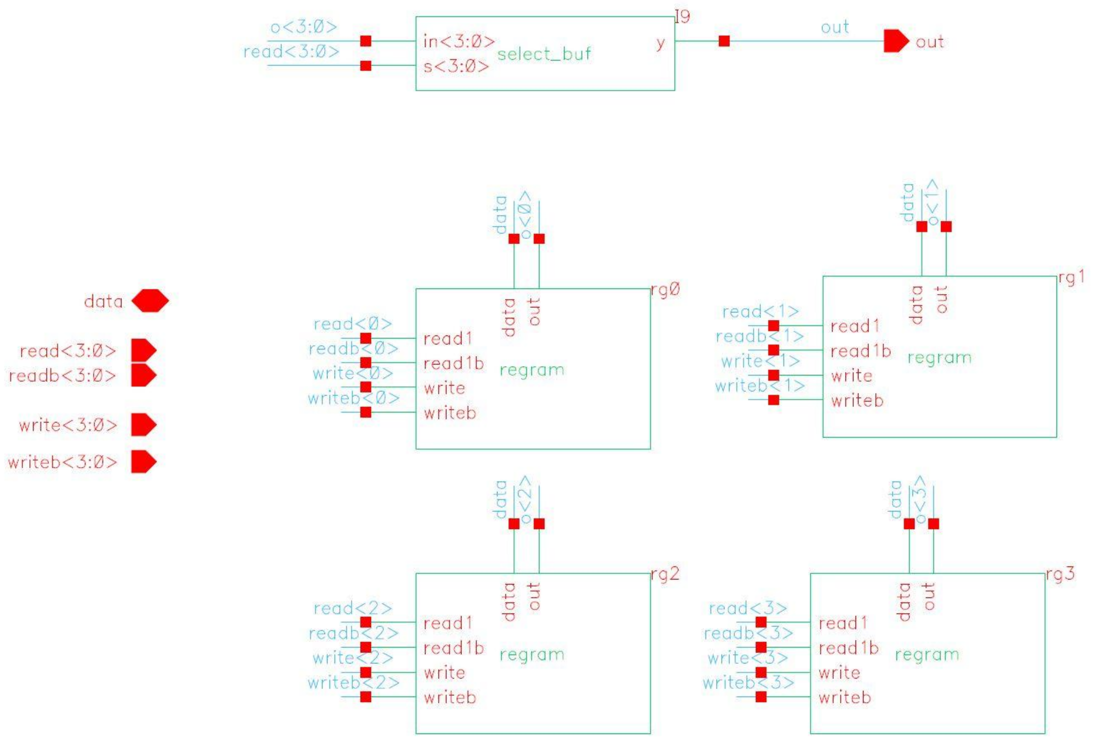
Appendix 3: Test vectors (arranged in 4 columns)

```
x_xx_xxx_xxx_0_0_1    0_10_xxx_xxx_0_1_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_010_000_1_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_1_0_0
0_10_011_001_1_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_1_0_0    0_01_xxx_xxx_0_1_0
0_10_xxx_xxx_1_0_0    0_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    0_01_101_010_1_0_0
0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_101_101_1_0_0    1_01_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_100_100_1_0_0    1_01_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_1    0_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_101_011_1_0_0    0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_1_0_0
0_01_100_010_1_0_0    0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_1_0
0_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_101_011_1_0_0
0_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_10_010_000_1_0_0    0_01_101_010_1_0_0
0_01_xxx_xxx_0_0_0    0_10_000_110_1_0_0    0_10_011_001_1_0_0    1_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    0_10_001_101_1_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    1_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    1_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_010_110_1_0_0    1_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_011_101_1_0_0    1_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    1_01_xxx_xxx_1_0_0
0_10_xxx_xxx_0_0_0    1_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    0_01_xxx_xxx_0_1_0
0_10_xxx_xxx_0_0_0    1_10_xxx_xxx_1_0_0    0_01_100_100_1_0_0    0_01_100_100_1_0_0
0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_011_101_1_0_0    0_01_101_101_1_0_0
0_10_xxx_xxx_0_0_0    0_10_001_101_1_0_0    0_01_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    0_10_011_111_1_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_100_010_1_0_0    1_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_010_000_1_0_0    1_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    1_10_xxx_xxx_0_0_0    1_01_xxx_xxx_1_0_0    1_01_xxx_xxx_1_0_0
0_01_xxx_xxx_0_0_0    1_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    1_10_xxx_xxx_1_0_0    0_01_100_100_1_0_0    0_01_110_110_1_0_0
0_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    0_01_010_100_1_0_0    0_01_101_101_1_0_0
0_01_xxx_xxx_0_0_0    0_10_010_100_1_0_0    0_01_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0
0_01_xxx_xxx_0_0_0    0_10_011_101_1_0_0    1_01_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0
0_10_011_101_1_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0
0_10_010_110_1_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    0_10_xxx_xxx_0_0_0
1_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    1_01_xxx_xxx_1_0_0    0_10_xxx_xxx_0_0_0
1_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    0_01_xxx_xxx_0_1_0    0_10_111_001_1_0_0
1_10_xxx_xxx_0_0_0    0_01_xxx_xxx_0_0_0    0_01_111_111_1_0_0    0_10_xxx_xxx_0_0_0
1_10_xxx_xxx_0_0_0    0_01_101_001_1_0_0    0_01_100_000_1_0_0    1_10_xxx_xxx_0_0_0
1_10_xxx_xxx_0_0_0    0_01_010_100_1_0_0    0_01_xxx_xxx_0_0_0
1_10_xxx_xxx_1_0_0    0_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
0_10_011_101_1_0_0    1_01_xxx_xxx_0_0_0    1_01_xxx_xxx_0_0_0
```

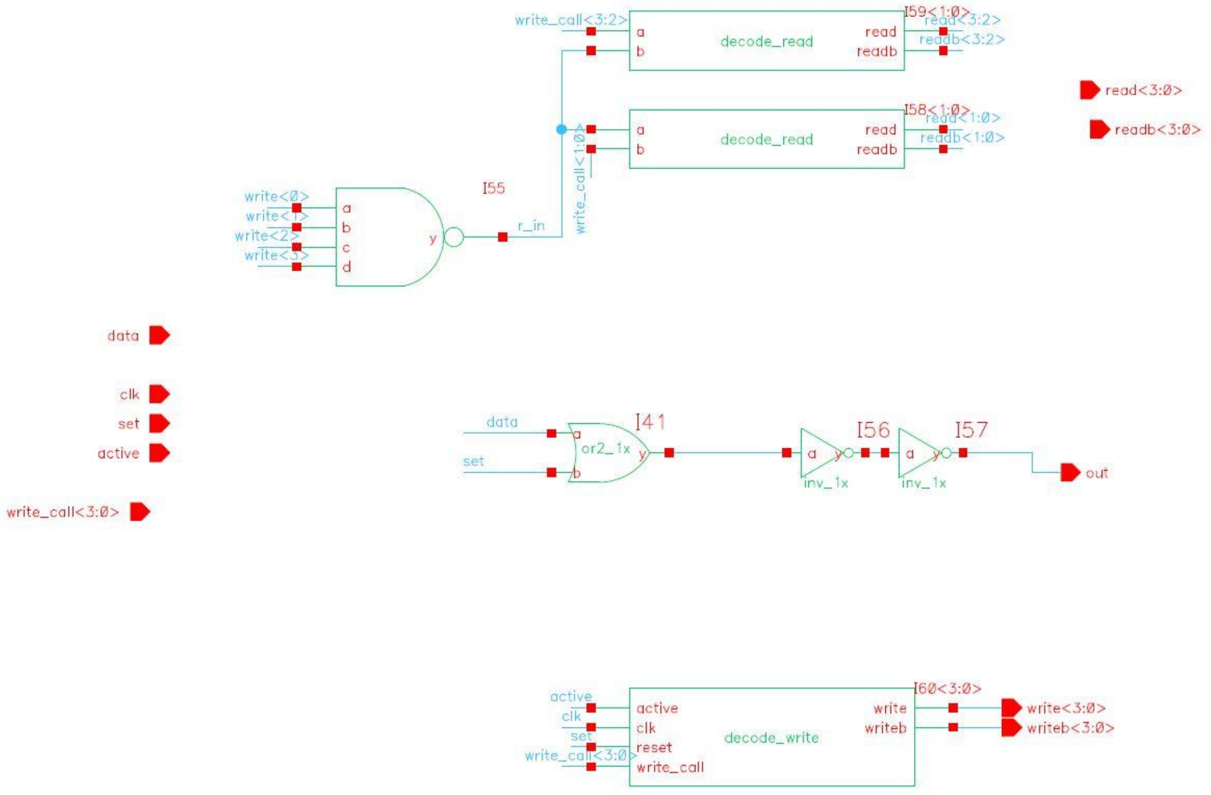
Appendix 4: Schematics



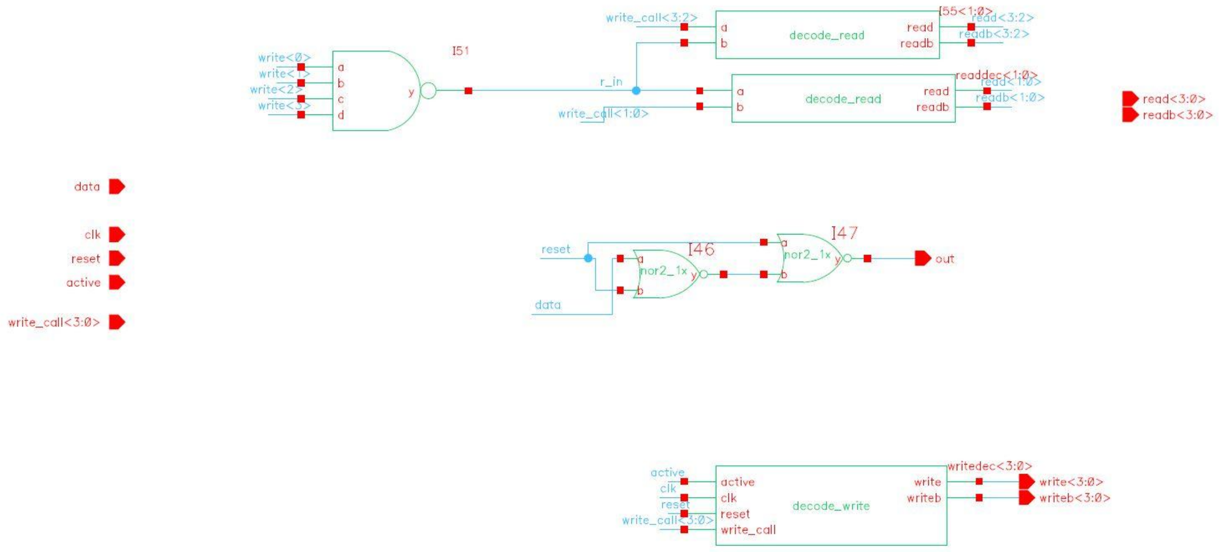
Schematic for regram



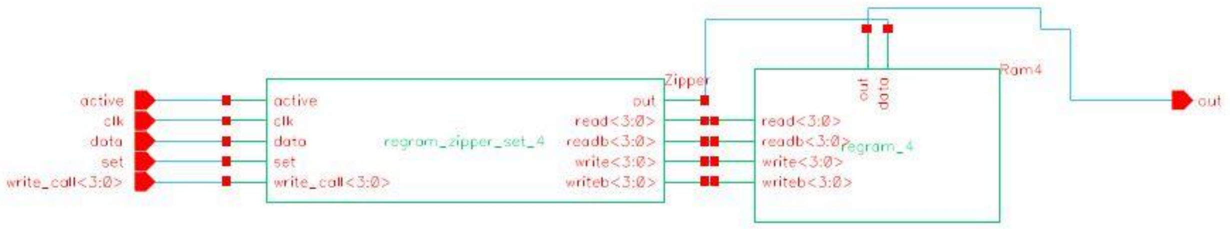
Schematic for regram_4



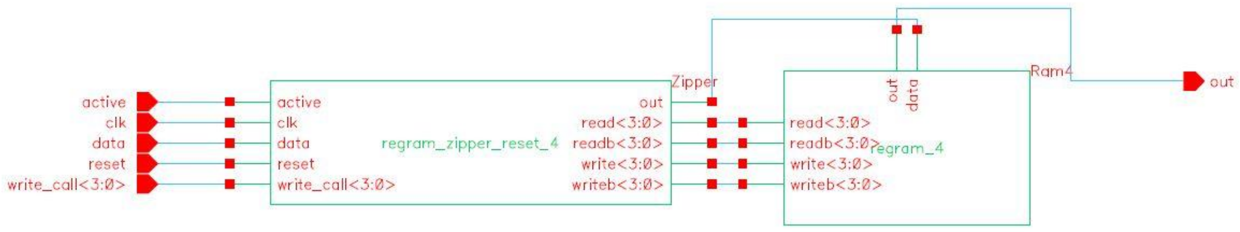
Schematic for regram_zipper_set_4



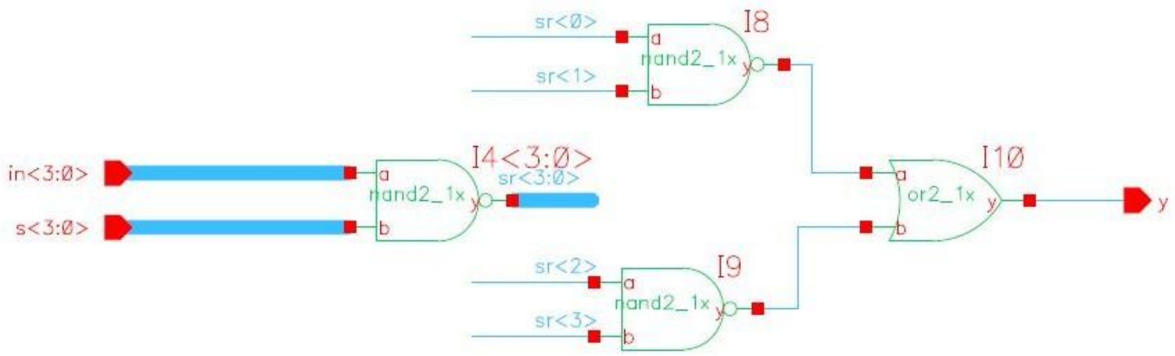
Schematic for regram_zipper_reset_4



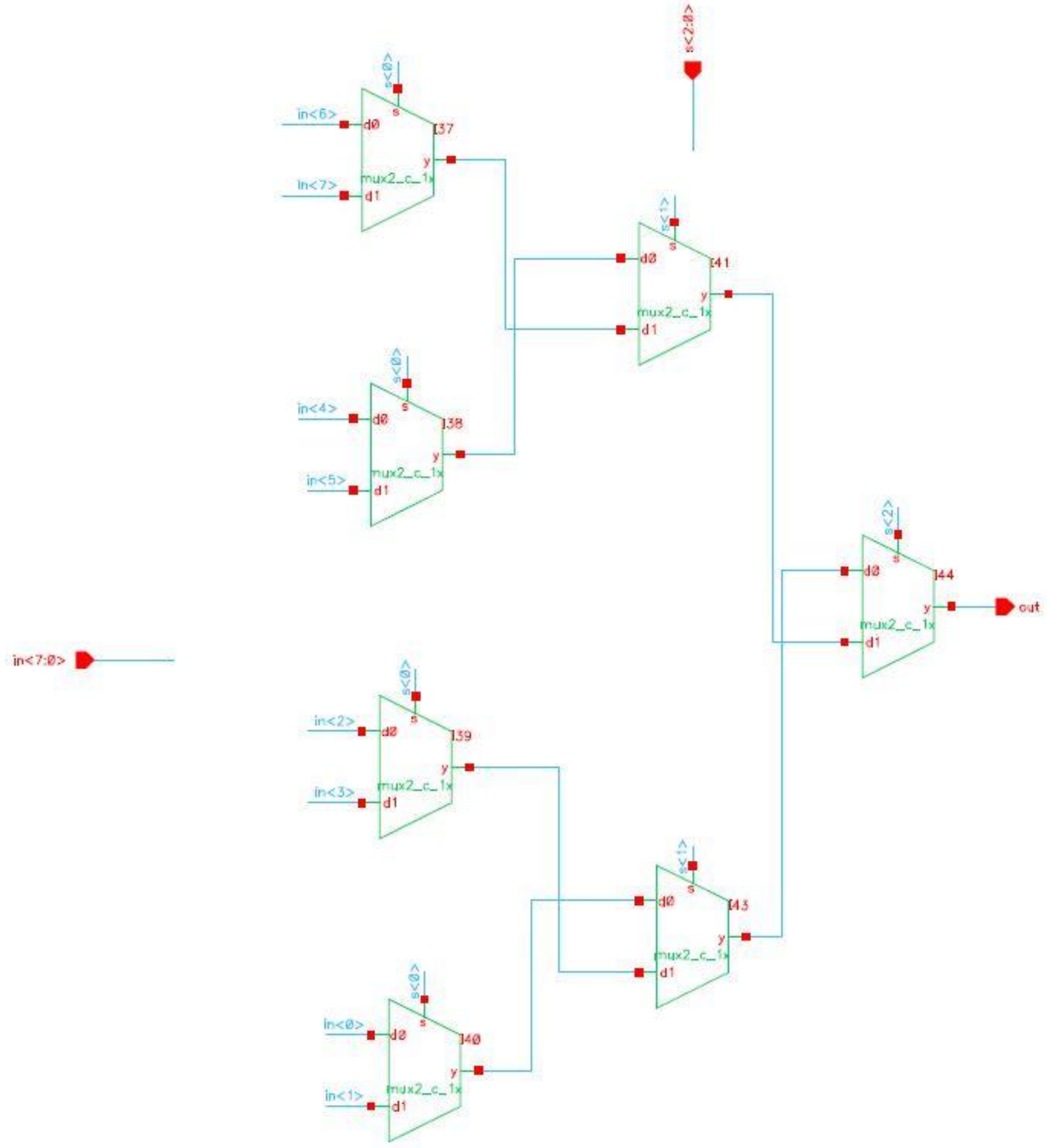
Schematic for regram_vector_set_4



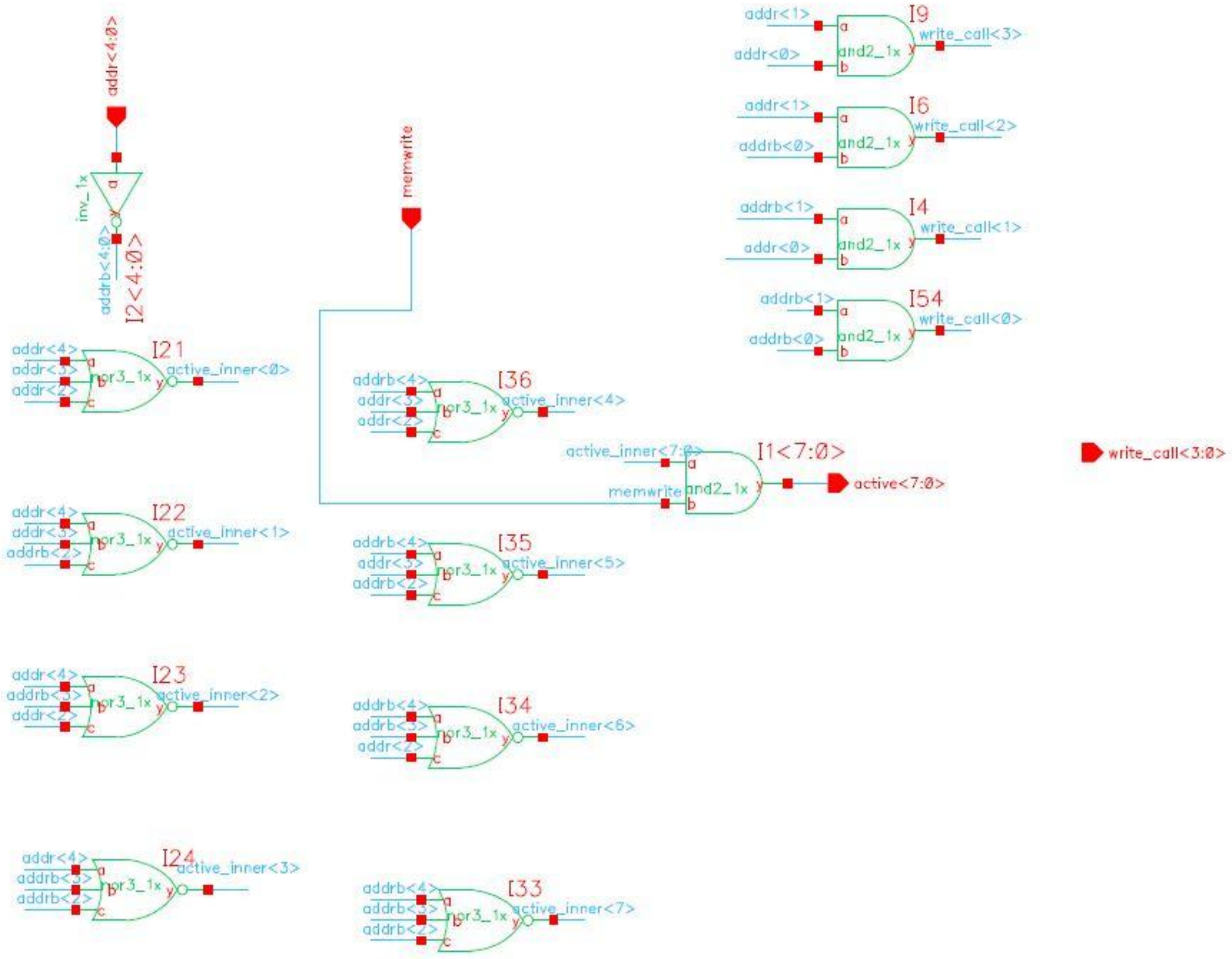
Schematic for regram_vector_reset_4



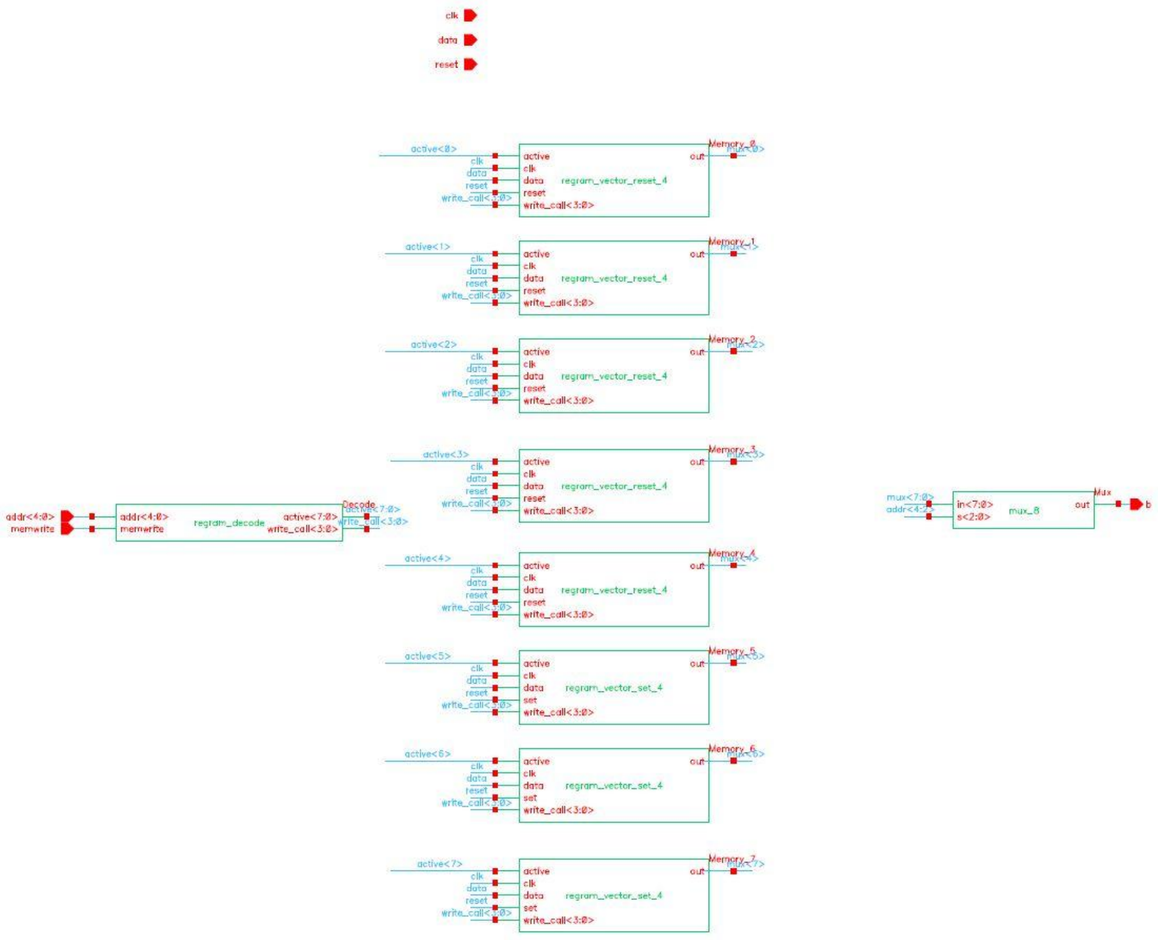
Schematic for select_buf



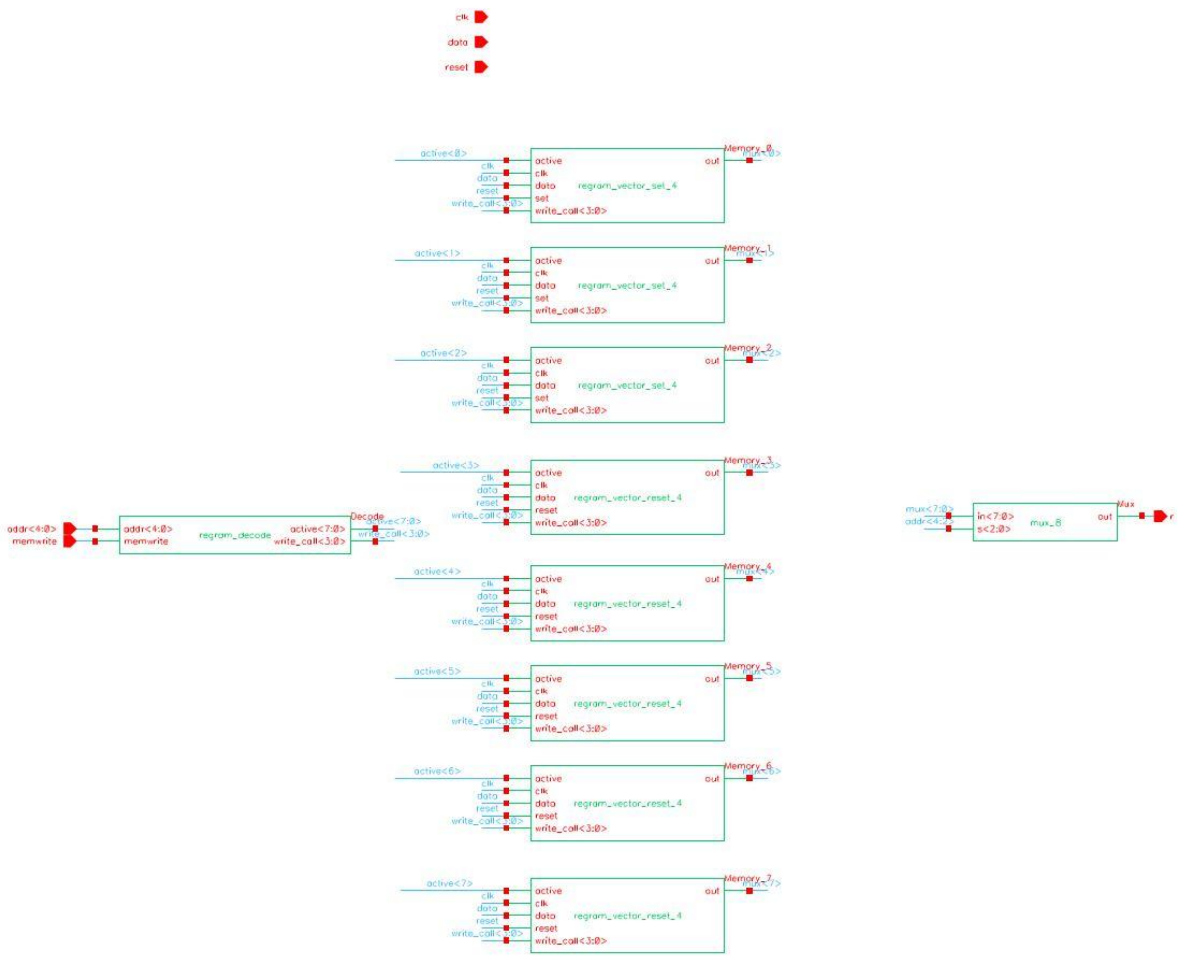
Schematic for mux_8



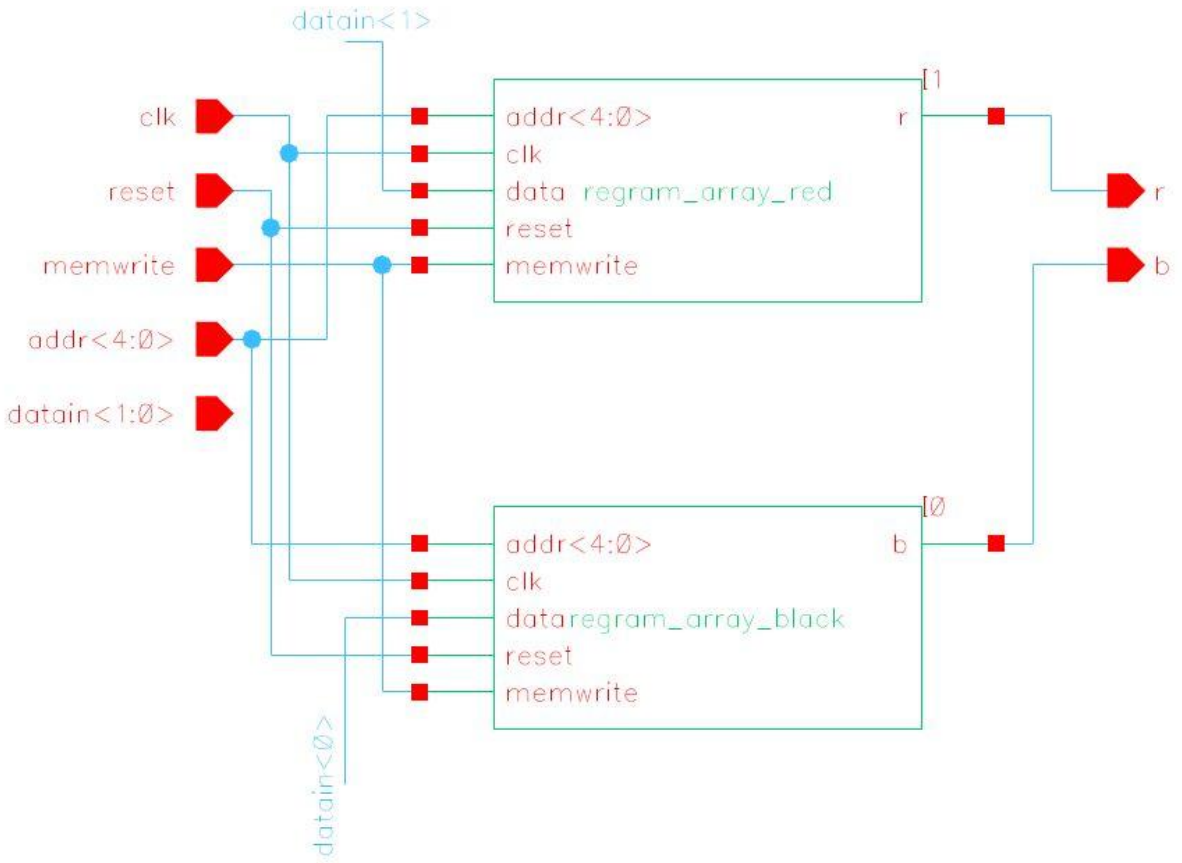
Schematic for regram_decode



Schematic for regram_array_black

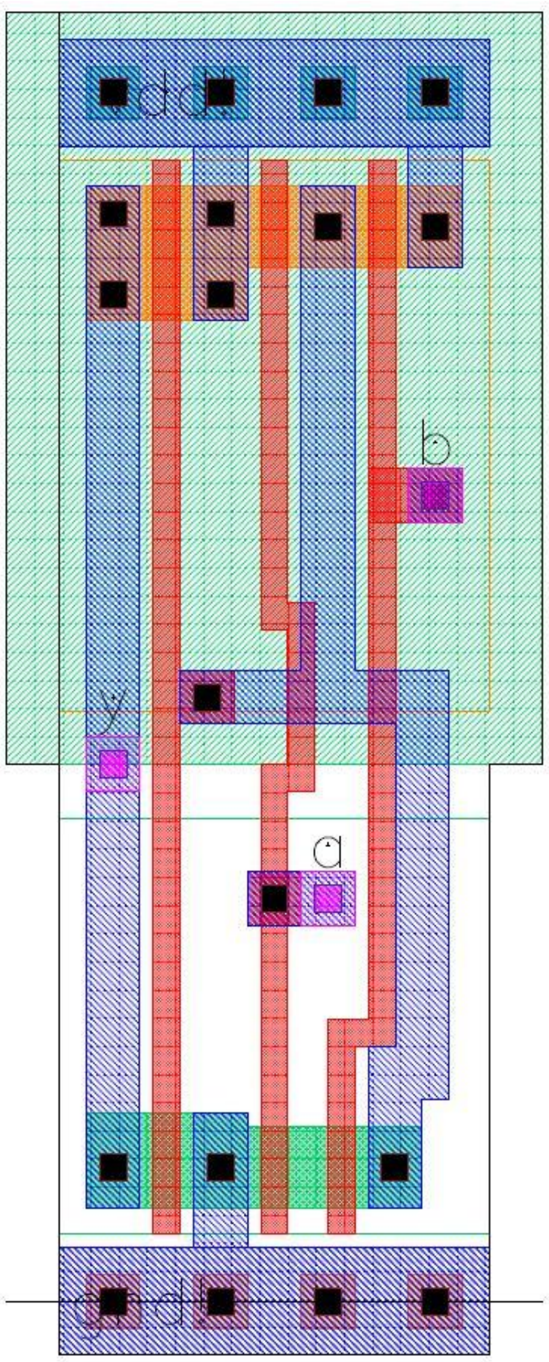


Schematic for regram_array_red

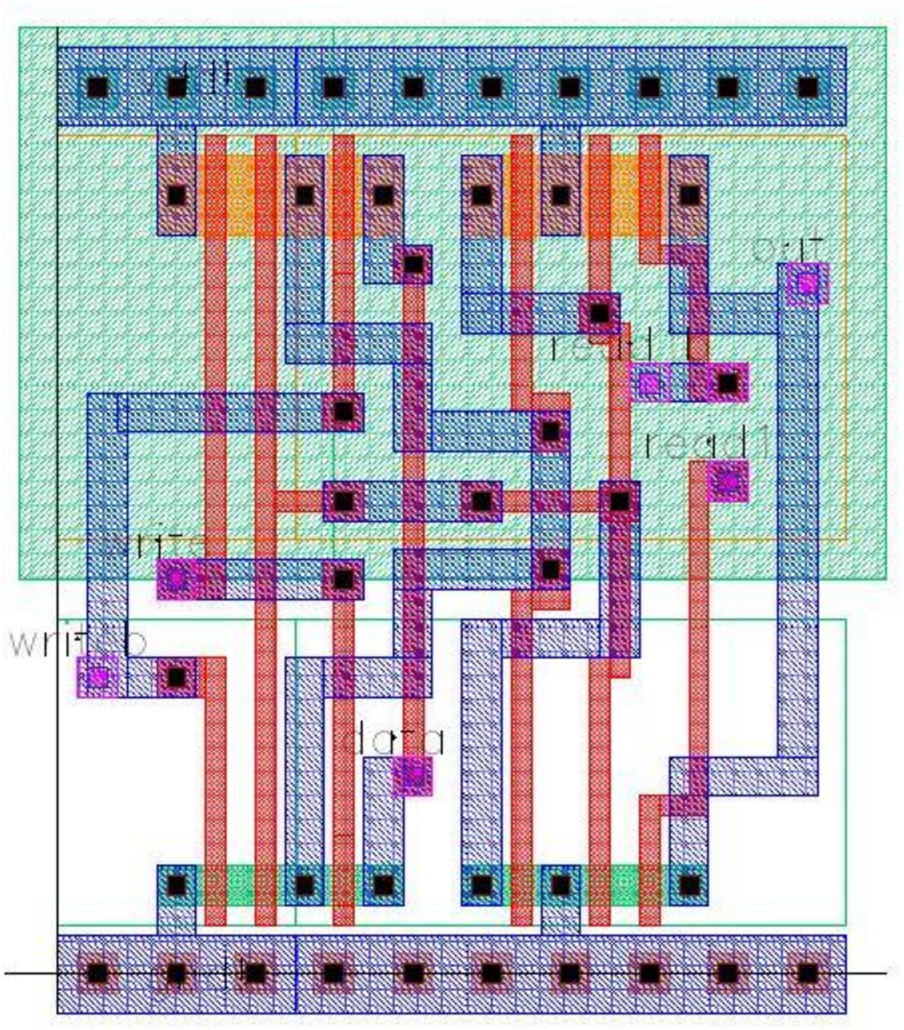


Schematic for mem

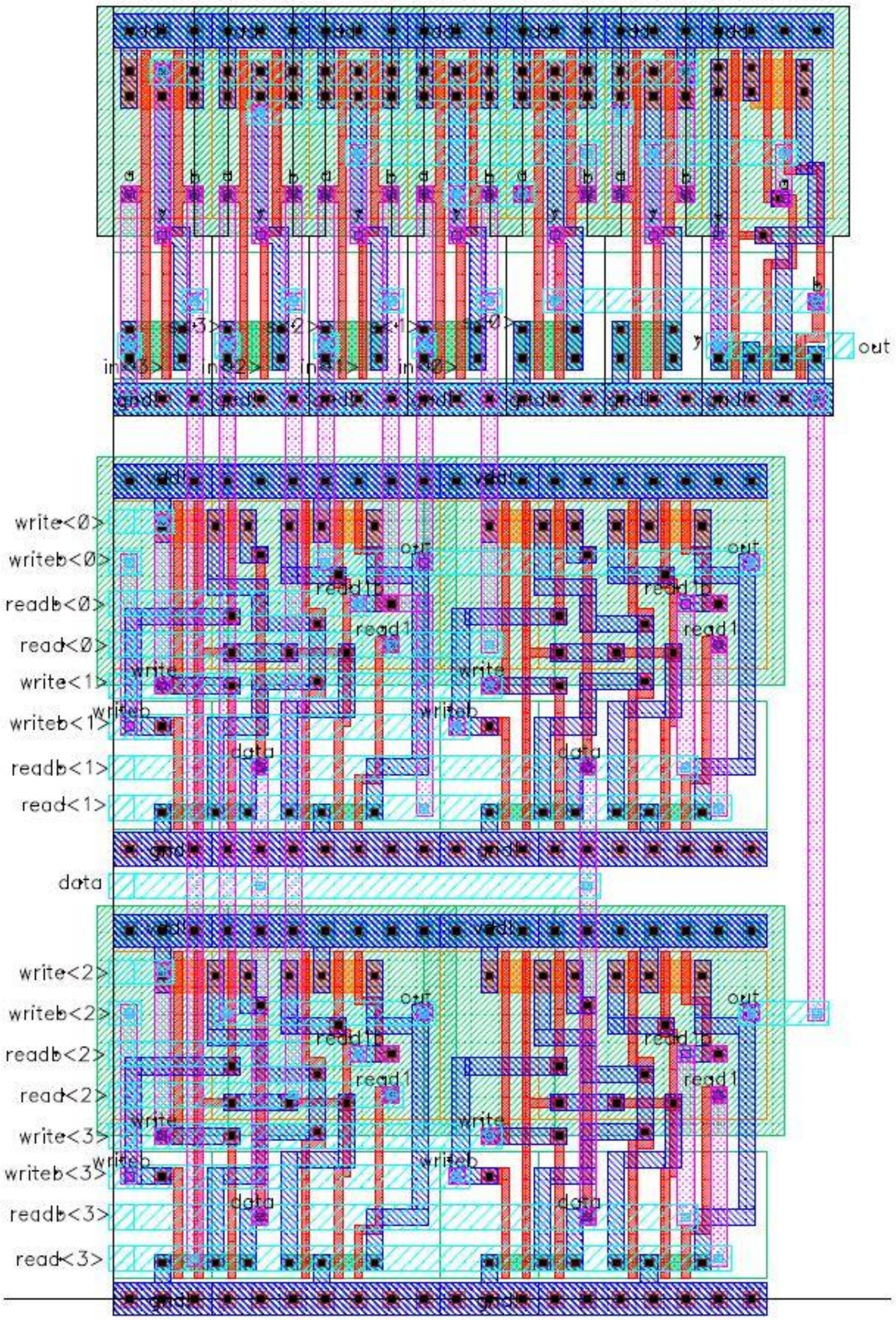
Appendix 5: Layouts



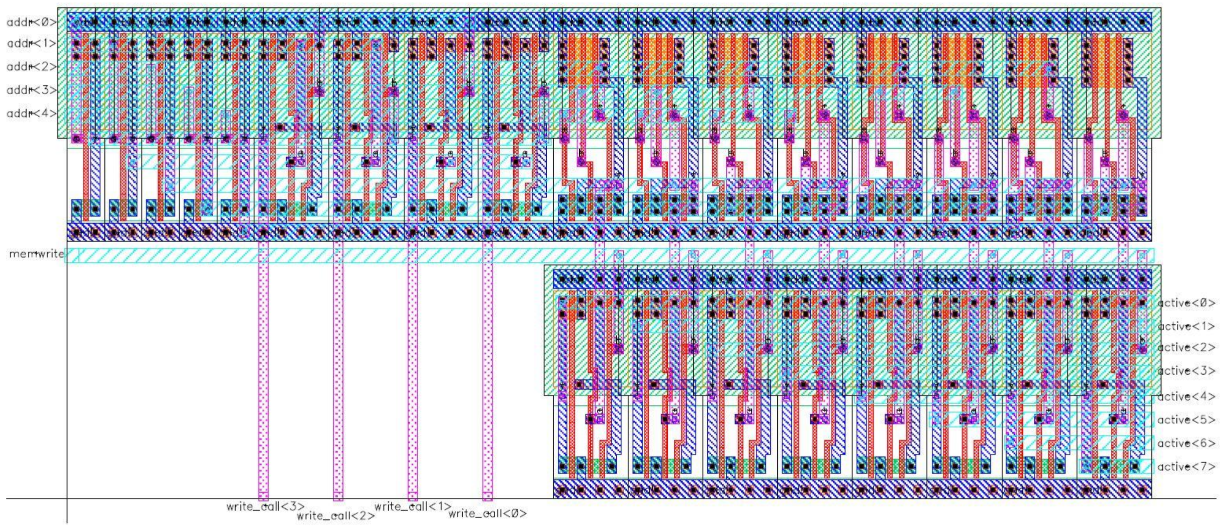
Layout for and2_1x (custom, moved pins)



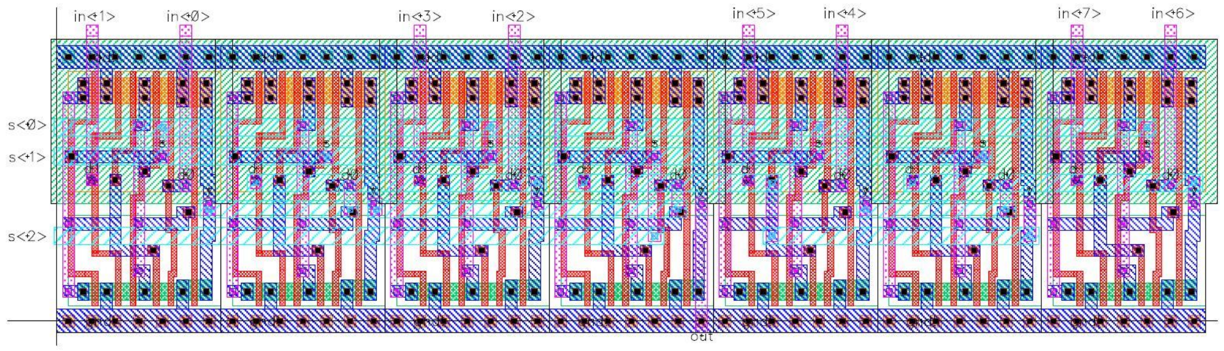
Layout for regram



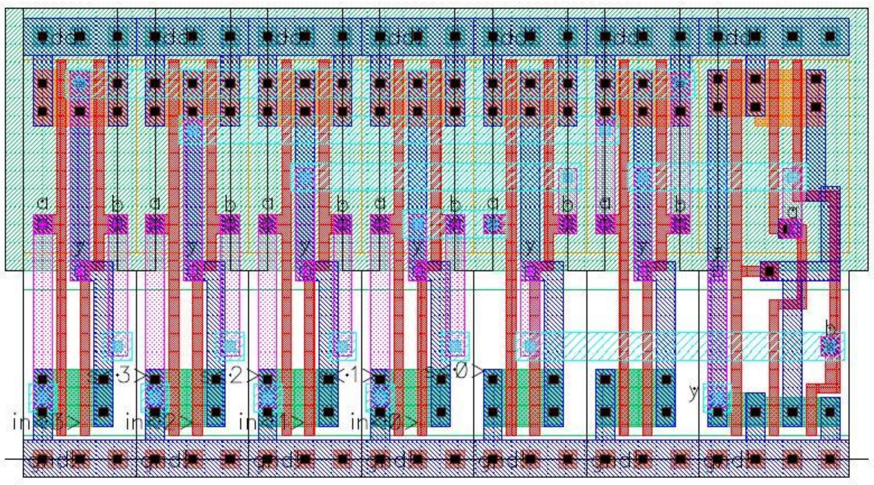
Layout for regram_4



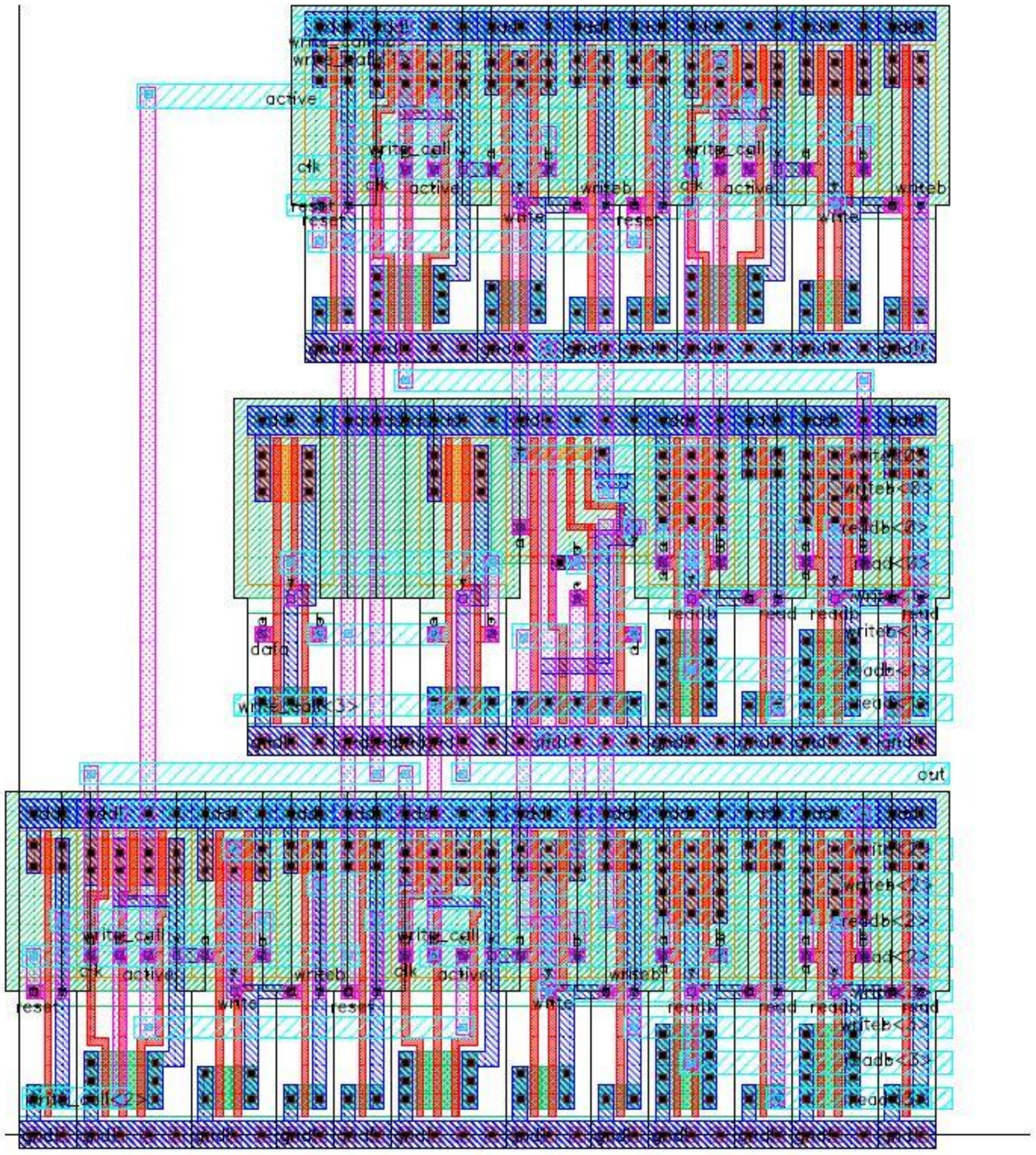
Layout for regram_decode

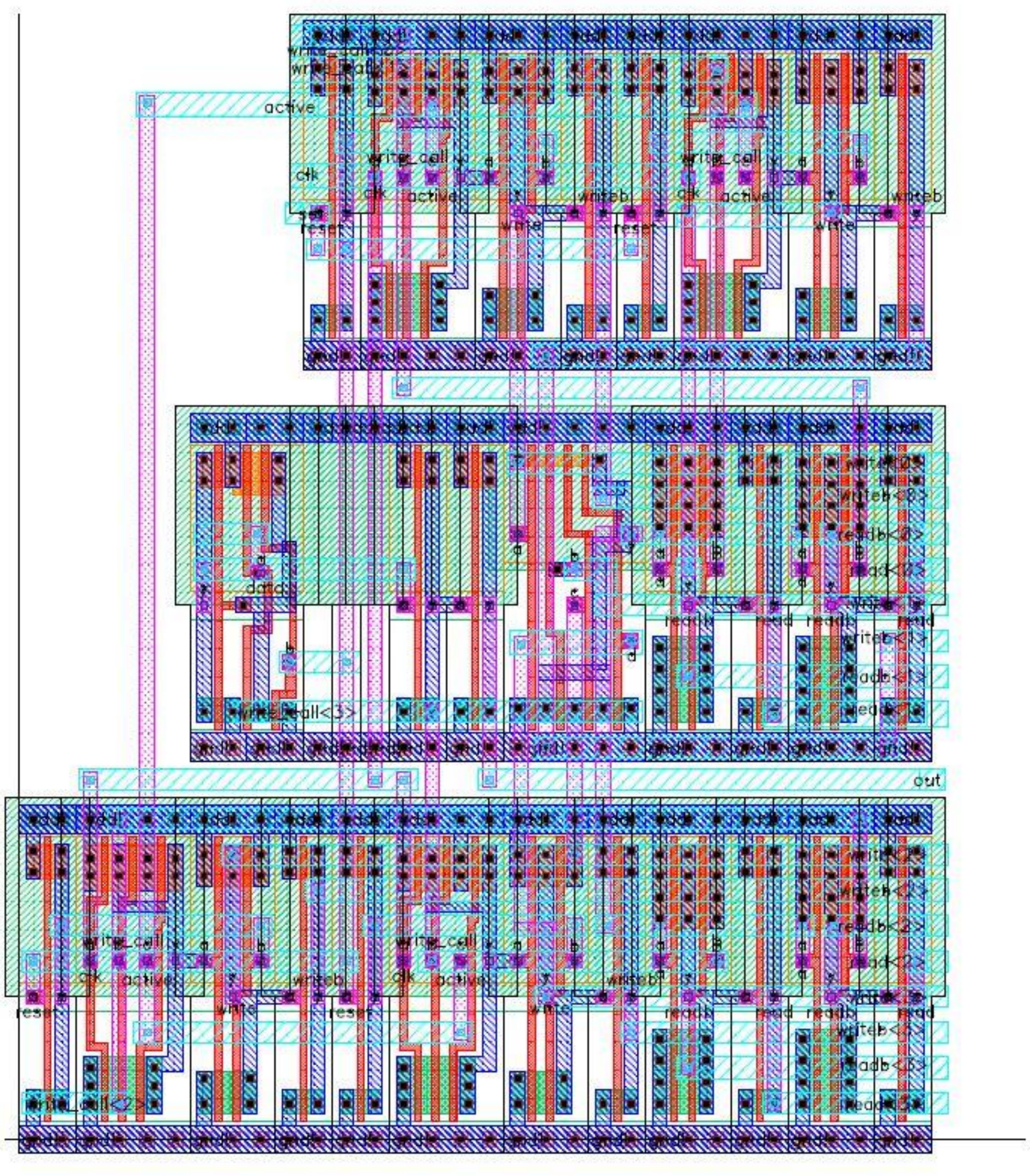


Layout for mux_8

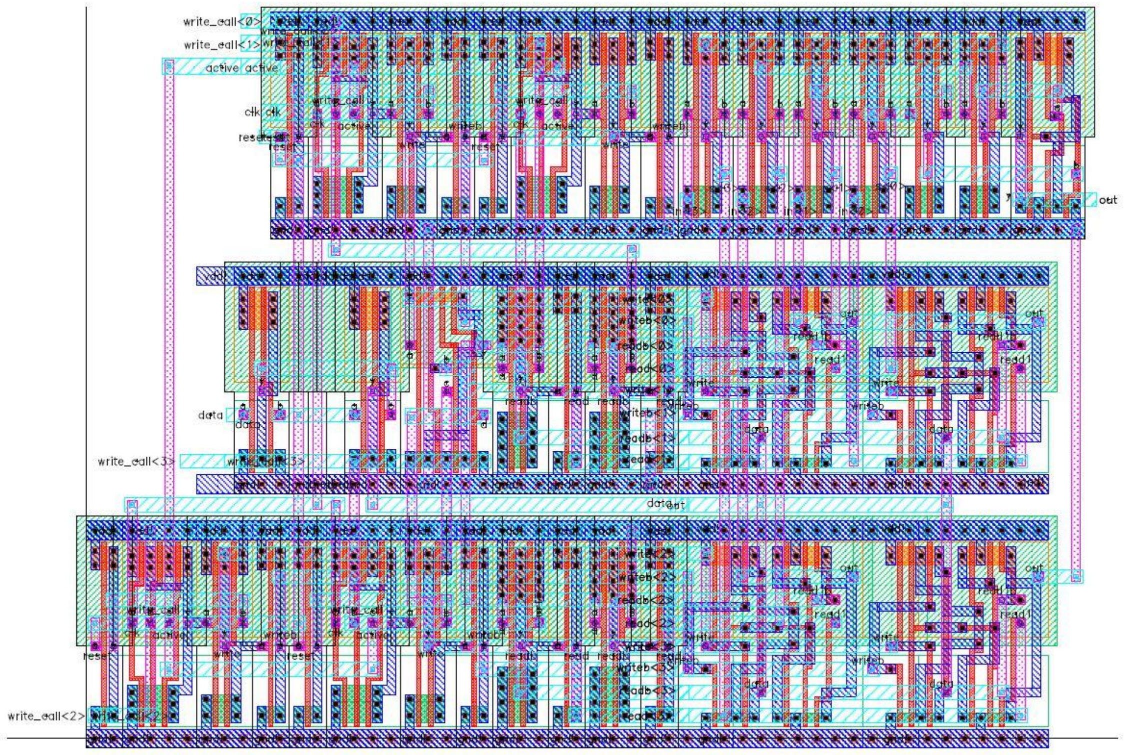


Layout for select_buf

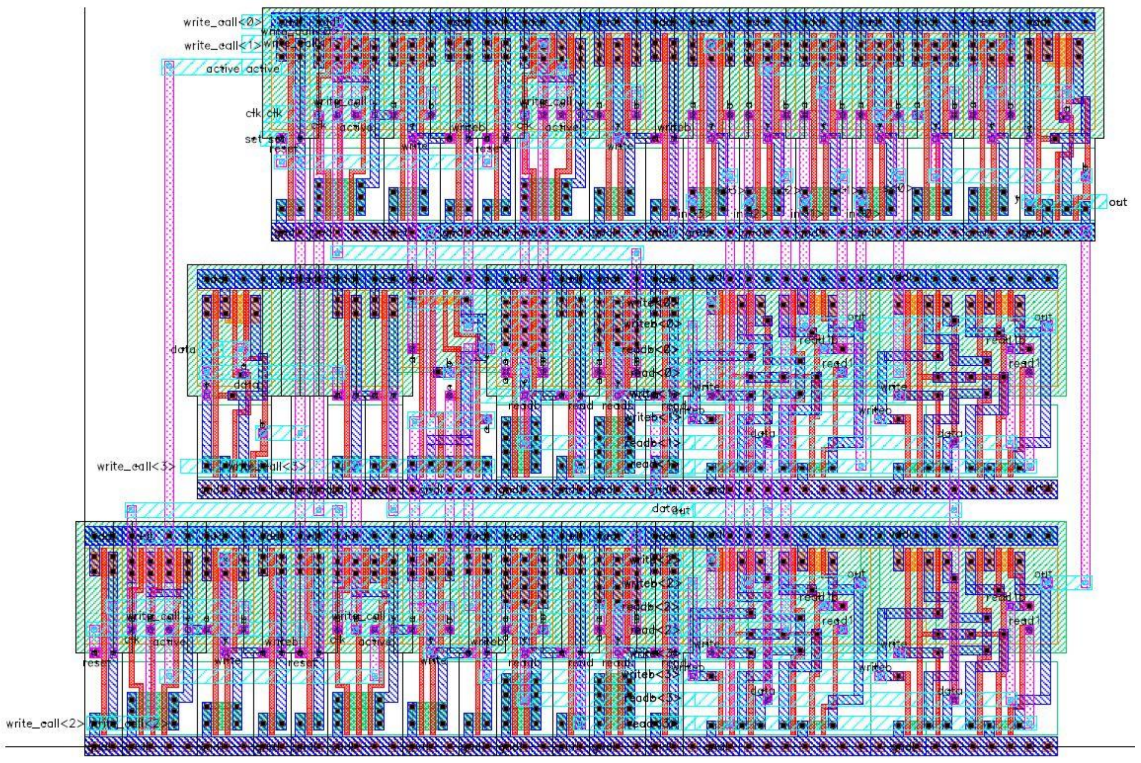

Layout for regram_zipper_reset_4



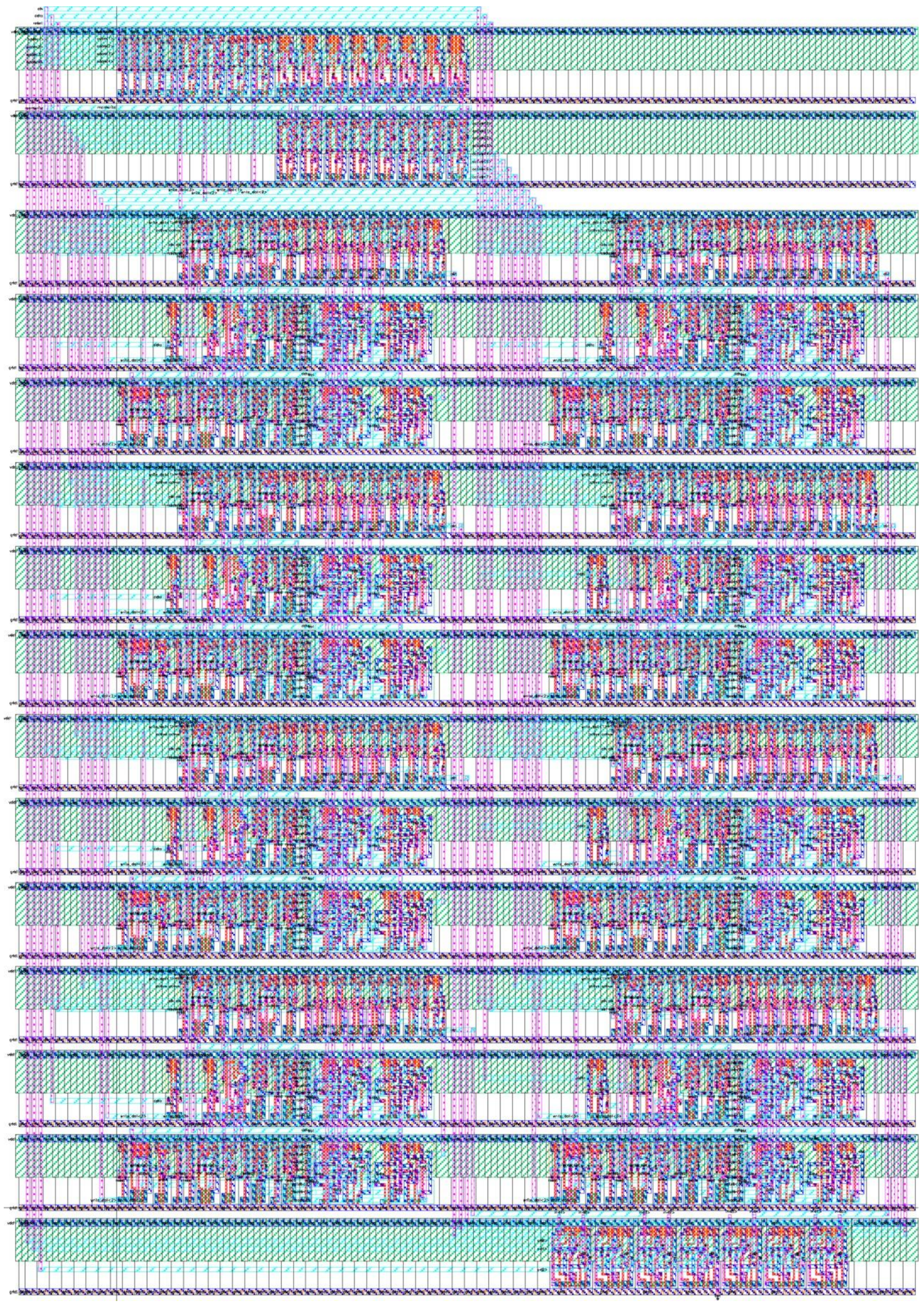
Layout for regram_zipper_set_4



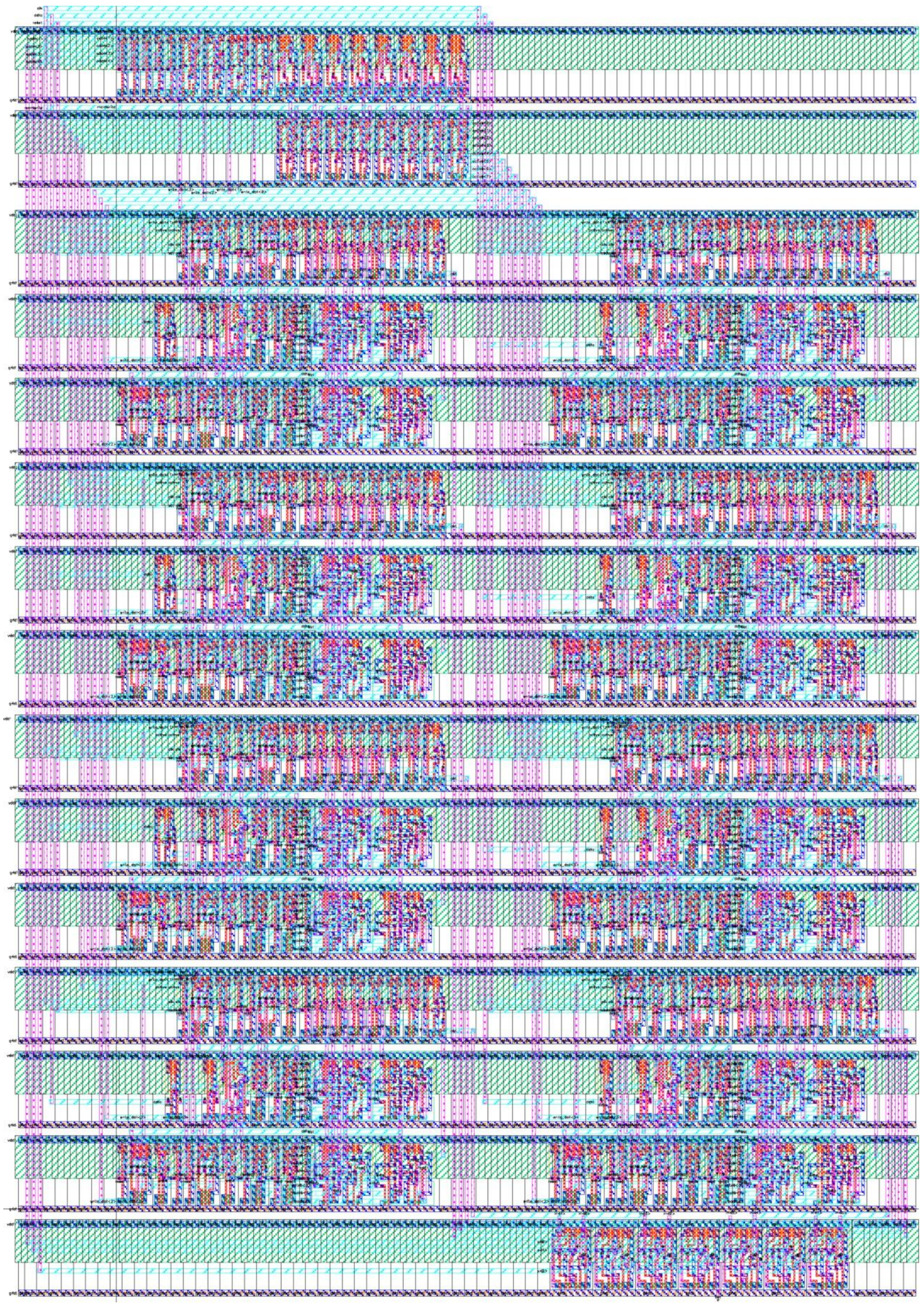
Layout for regram_vector_reset



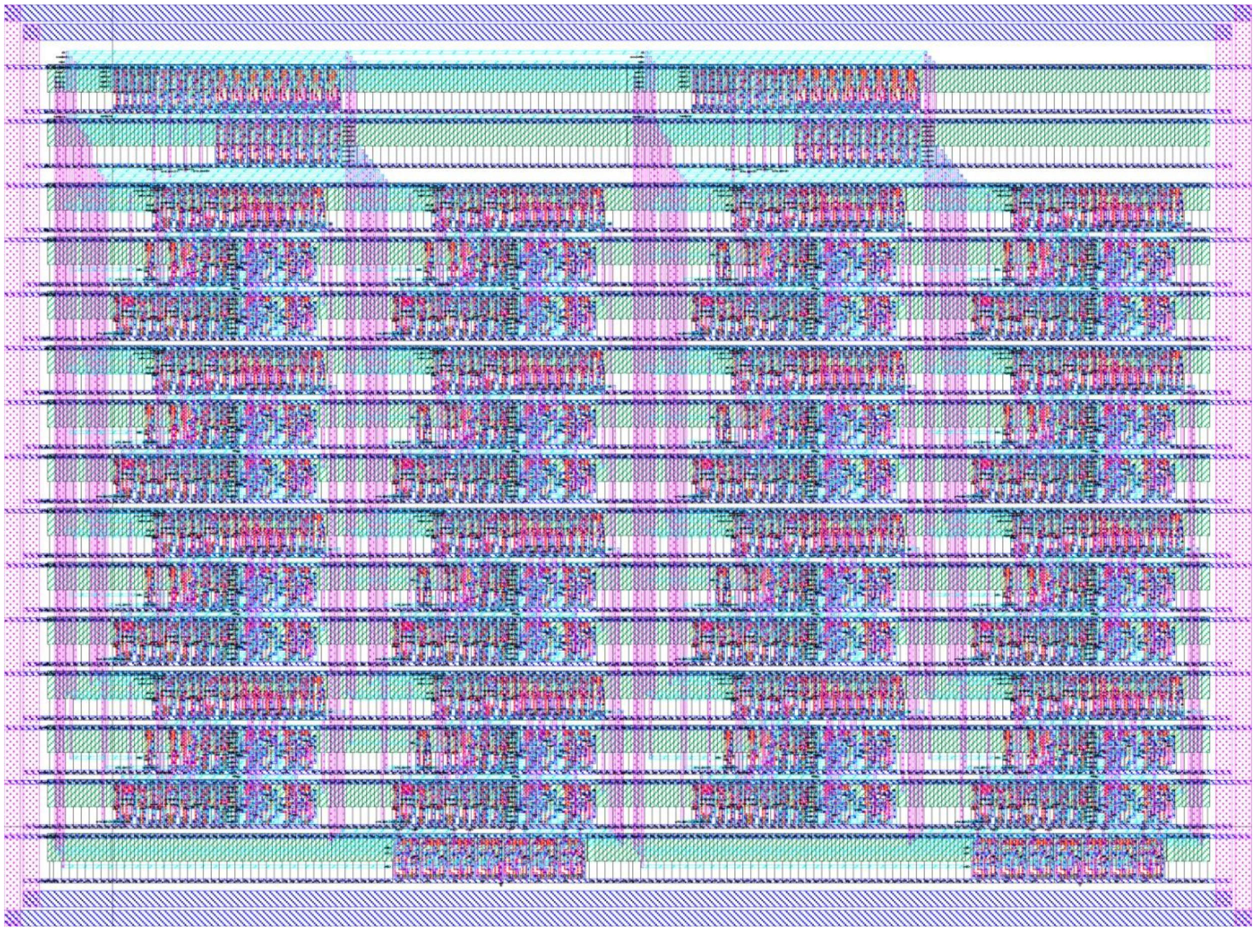
Layout for regram_vector_set



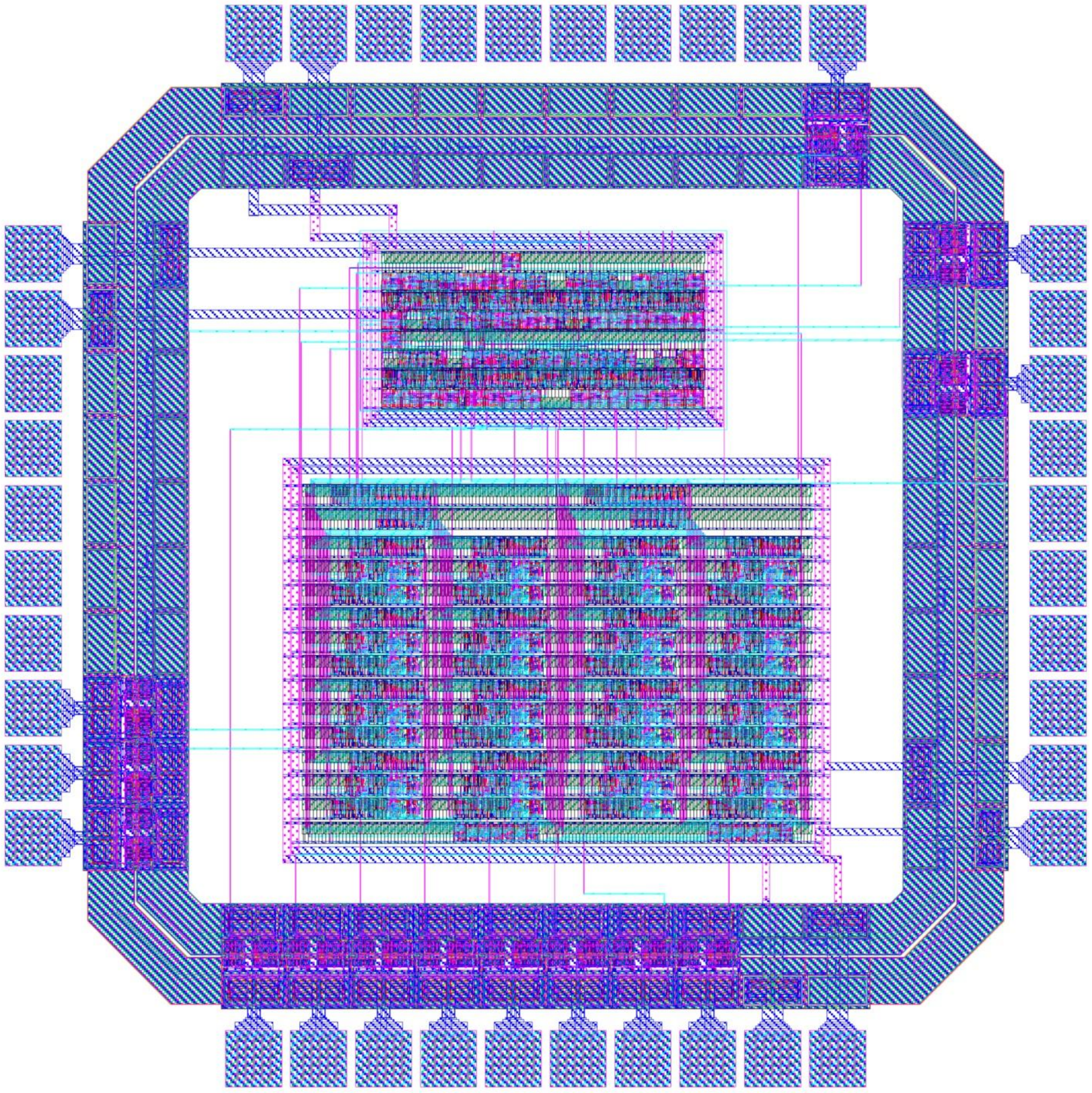
Layout for regram_array_black



Layout for regram_array_red



Layout for mem



Layout for chip