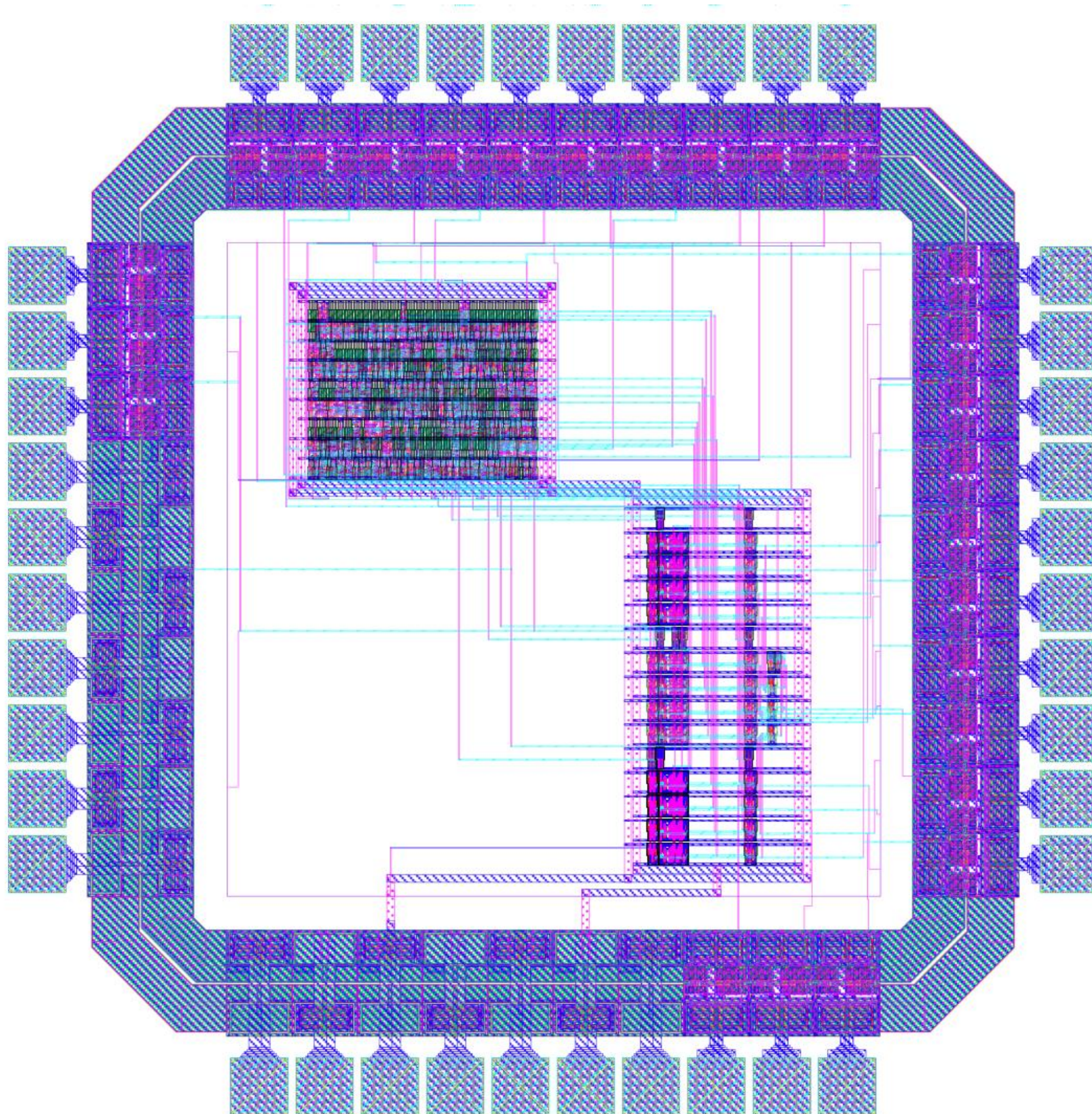
The Game of Nim



Tim Nguyen
Andrew Xue

E158: CMOS VLSI Design
Prof. David Money Harris
April 17, 2010

INTRODUCTION

Nim is a strategy game in which two players take turns removing items from three different stacks. The object of the game is to remove the last item from the last stack. The most interesting aspect of the game of Nim is that it has a mathematical solution; a player can guarantee that he will win by following a specific formula to determine each move.

This report documents the design process for a microchip that allows a human to play a game of Nim against an AI player that uses the mathematical solution to calculate each of its moves. The chip was designed to fit in a 1.5 mm by 1.5 mm 40-pin MOSIS "TinyChip" fabricated in a 0.6-μm process.

SPECIFICATIONS

Excluding the pins set aside for power and ground, the chip had a total of 12 inputs and outputs:

Name	Width	Direction	Description
ph1	1	Input	Two-phase clock
ph2	1	Input	Two-phase clock
reset	1	Input	Starts a new game
turn_enable	1	Input	Chip will only process user input when this is asserted
stack_choice_in	3	Input	Stack from which items will be removed
num_remove_in	4	Input	Number of items to be removed
invalid	1	Output	Indicates whether or not user input was a valid move
human_winner	1	Output	Indicates that the human player won
ai_winner	1	Output	Indicates that the AI won
a_out	4	Output	Size of stack A
b_out	4	Output	Size of stack B
c_out	4	Output	Size of stack C

During the human turn, the system will not update the stacks unless the human input (num_remove_in) is nonzero, and less than or equal to the chosen stack size. It also must wait for an enable signal in order to register a human move. If the number removed is greater than the number of items in the chosen stack, or if an invalid stack is chosen, invalid will be enabled.

During the AI turn, the system will take the flop sizes and determine the best course of action using the following algorithm:

```
Let A, B, and C represent the number of beads in each stack.
□= XOR
A □ B □ C = X    | The bitwise XOR of A, B, and C (called the nim-sum)
A □ X = A’       | The bitwise XOR of A, B, and C with the nim-sum
B □ X = B’
C □ X = C’
A’ < A → remove (A-A’) from A     | If A’ is less than A, reduce the size of A to A’.
B’ < B → remove (B-B’) from A     | Check A first, and if that inequality fails, check B
C’ < C → remove (C-C’) from A     | and then C.
```

The key to the winning theory to the game is the binary sums of the heap sizes. The AI performs the bitwise XOR of A, B, and C. this is called the nim-sum. Next, it performs three more bitwise XORs between nim-sum and each of the stacks. If it is possible to reduce one of the stacks to the result of its bitwise XOR with the nim-sum, then the new nim-sum is zero, which ensures an AI victory. If the nim-sum is already zero before the AI can make its move, then it will remove one item from the first non-zero stack in order to prolong the game.

FLOORPLAN

Our original floorplan called for a 650x480λ controller and a 1500x440λ datapath, but both components were significantly larger in the final design. The finalized datapath was measured to be 856x1801λ and the controller was measured to be 1222x982λ.

The datapath turned out to be taller than expected due to modifications in design. In the proposal, we assumed the datapath would have a height of four bits and a zipper, but the finalized datapath had a height of twelve bits with three zippers in order to separate hardware specific to each stack.

The controller also turned out to be much larger than expected. This, however, was due to the relocation of the AI and the human logic from the datapath to the controller. The reasoning behind this was that the human and the AI logic were prohibitively time-consuming and overtly complicated to draw schematics and layouts for them.

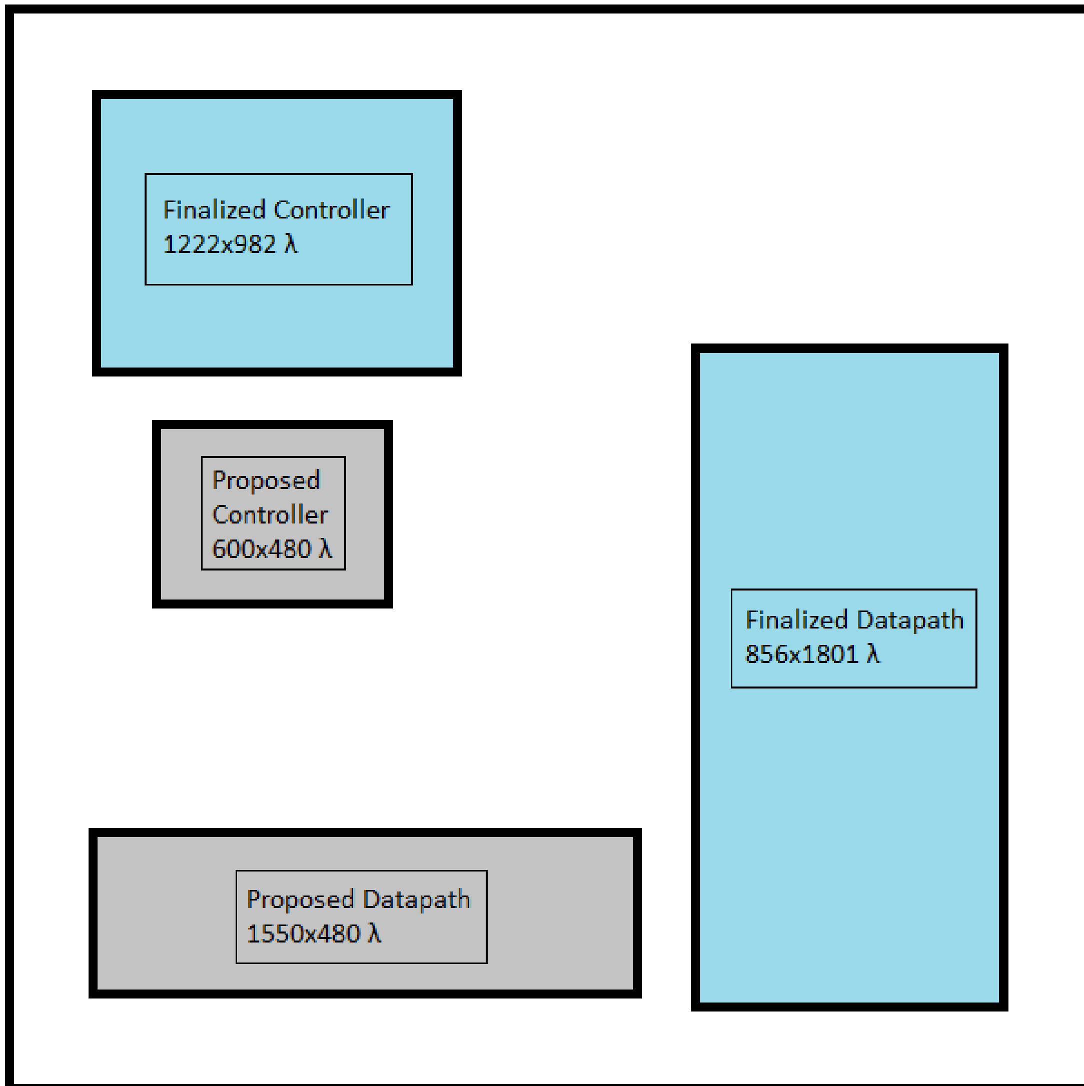


Figure 1: A comparison of the proposed and finalized controller and datapath

The slice plan for the datapath consisted of three four-bit sections stacked on top of each other, resulting in a total height of twelve bits and three zippers. This configuration was chosen because the same hardware was used for each of the three stacks. On the left was the stack memory, which consisted of an array of multiplexers feeding into the inputs of four-bit flip-flops. The outputs of the flip-flops were sent to the controller, which determined the next move, and the make_move cell, which sent updated values to the flip-flops. The check_winner cell on the right takes in the new stack values and performs a four-bit nor to determine whether or not all of the stacks are empty. If so, then the game is over and the datapath sends a high winner signal to the controller, which triggers the controller's winner logic and determines which player won the game.

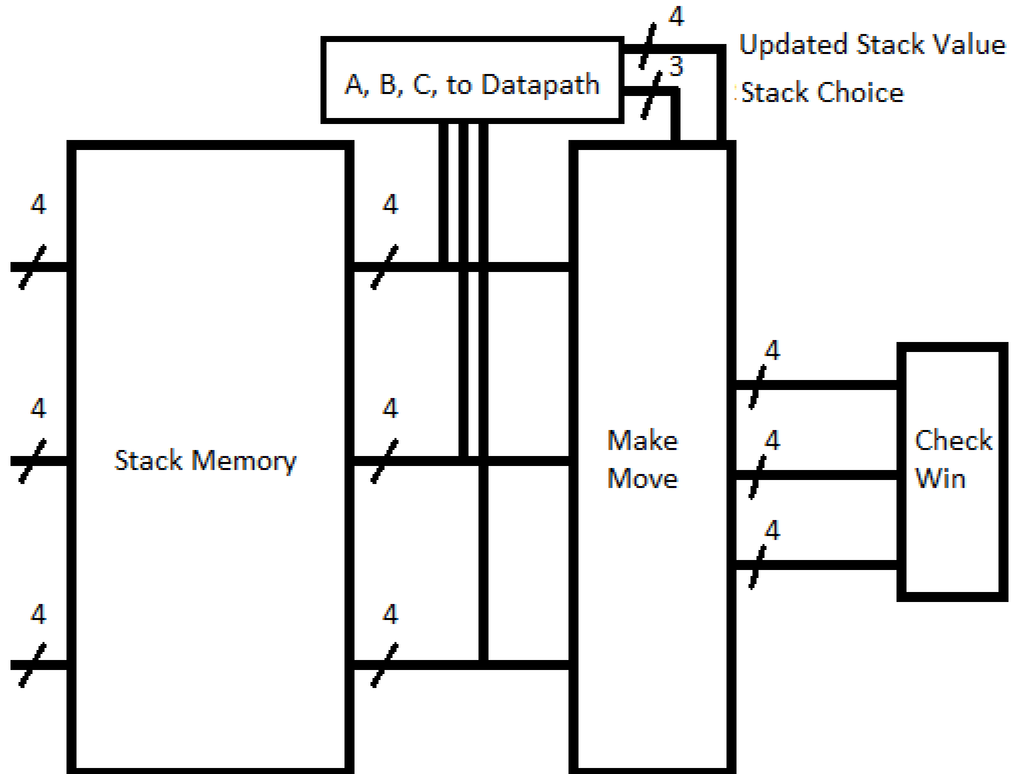


Figure 2: A slice plan of the datapath

VERIFICATION

- Does the Verilog pass testbench? Yes*
- Do schematics pass testbench? Yes*
- Does the layout pass DRC and LVS? Yes
- Does CIF load correctly and pass DRC and LVS? Yes

Discrepancies:
- The Verilog and the schematics do not fully pass the testbench (there is one error out of the 17 checks), but this error arises from the way test vectors are read into the system. Otherwise the system passes the testbench and simulates correctly.

POSTFABRICATION TEST PLAN

If the chip were fabricated, a board would need to be built around it. The input signals would be wired to switches or buttons to indicate low/high values. The outputs would be wired to LEDs in order to indicate values. A function generator would have to apply a two-phase clock input to the system in order for it to function correctly. The same test vectors could be applied to ensure consistency and functionality.

Design Time

The following table contains a list of how much time was spent on each stage of the design process. All of the time was spent with both designers working together, so the total number of man-hours spent on each stage can be found by multiplying the provided number by two.

Stage	**Time Spent (hours)**
Project Proposal	3
Verilog	16
Schematics	16
Layout	20
Final Report	5
Total	**60**

File Locations

All of the files for this project are saved on the chips server in the Parsons VLSI Lab at Harvey Mudd College. The specific files for each stage of the design process can be found in the following directories:

Files	**Path and Filename**
Verilog Code	/home/zxue/IC_CAD/cadence/proj2/nim_verilog.sv
Test Vectors	/home/zxue/IC_CAD/cadence/proj2/nim_testvectors.tv
Synthesis Results	/home/tnguyen/IC_CAD/synth/proj2_1/
All Cadence Directories	/home/zxue/IC_CAD/cadence/proj2/
CIF	/home/zxue/IC_CAD/cadence/proj2_cifin/
PDF Chip Plot	/home/zxue/IC_CAD/cadence/proj2/nim_chip_plot.pdf
PDF of this Report	/home/zxue/IC_CAD/cadence/proj2/finalreport.pdf

APPENDICES

Appendix 1: Verilog Code

```
`timescale 1ns / 100ps

module testbench();

  logic         ph1, ph2;
  logic         reset;

  logic turn_enable;
  logic [2:0]   stack_choice_in;
  logic [3:0]   num_remove_in;
  logic         invalid, human_winner, ai_winner;
  logic [3:0]   a_out, b_out, c_out;
  logic         invalid_exp, human_winner_exp, ai_winner_exp;
  logic [3:0]   a_exp, b_exp, c_exp;

  logic [31:0] vectornum, errors;

  logic [29:0] testvectors[1000:0];

  // instantiate device to be tested
  core sexycore( ph1, ph2, reset, turn_enable,
                        stack_choice_in, num_remove_in,
                        invalid, human_winner, ai_winner,
                        a_out, b_out, c_out);

/*
  // The following should be used when testing the chip.
  chip testcore( a_out, ai_winner, b_out, c_out, human_winner, invalid,
     num_remove_in, ph1, ph2, reset, stack_choice_in, turn_enable );

*/

  // initialize test and load vectors
        initial begin
                reset <= 1; # 84; reset <= 0;

                // where to dump the results
                $dumpfile("nim_core_test.vcd");

                // dump the variables
                $dumpvars(1, ph1, ph2, invalid, human_winner, ai_winner, a_out, b_out, c_out);
                // load test vectors
                $readmemb("nim_testvectors.tv", testvectors);

                vectornum = 0; errors = 0;
        end

  // generate clock to sequence tests
  always
    begin
     ph1 <= 0; ph2 <= 0; #8;
     ph1 <= 1; #12;
     ph1 <= 0; #8;
     ph2 <= 1; #12;
    end


  always @(posedge ph2)

    if (!reset) begin // skip during reset

      if ((a_out !== a_exp) || (b_out !== b_exp) || (c_out !== c_exp)) begin
```

```
        $display("Error: turn_enable = %h, stack_choice_in = %h, num_remove_in = %h,",
                turn_enable, stack_choice_in, num_remove_in);
        $display("invalid = %h, human_winner = %h, ai_winner = %h,",
                invalid, human_winner, ai_winner);
        $display("(a,b,c) = (%h, %h, %h)",
                a_out, b_out, c_out);
        $display("expected invalid = %h, human_winner = %h, ai_winner = %h,",
                invalid_exp, human_winner_exp, ai_winner_exp);
        $display("expected (a,b,c) = (%h, %h, %h)\n",
                a_exp, b_exp, c_exp);
        errors = errors + 1;

      end
      else begin
        $display("Error-free move: (a,b,c) = (%h, %h, %h)\n",
                a_out, b_out, c_out);
      end

    vectornum = vectornum + 1;


    if (testvectors[vectornum] === 30'bx) begin

      $display("%d tests completed with %d errors",

        vectornum, errors);

      $dumpflush;

      $finish;

    end

        #2;     {turn_enable, stack_choice_in, num_remove_in,
                invalid_exp, human_winner_exp, ai_winner_exp,
                a_exp, b_exp, c_exp} = testvectors[vectornum];

  end
endmodule

// Top level
module core(
        input  logic            ph1, ph2, reset, turn_enable,
        input  logic [2:0]      stack_choice_in,
        input  logic [3:0]      num_remove_in,
        output logic            invalid, human_winner, ai_winner,
        output logic [3:0]      a_out, b_out, c_out);

        // Wire connections not otherwise defined
        // (just things passed from controller to datapath and vice versa)
        logic           a_enable, b_enable, c_enable, winner;
        logic [2:0]     stack_choice;
        logic [3:0]     num_remove_out;

        // Instantiate controller in core level
        controller nim_controller(      reset, ph1, ph2, turn_enable, winner, a_out, b_out, c_out,
                                        stack_choice_in, num_remove_in, stack_choice, num_remove_out,
                                        a_enable, b_enable, c_enable,
                                        human_winner, ai_winner, invalid);

        // Instantiate datapath in core level
        datapath nim_datapath(          ph1, ph2, reset, a_enable, b_enable, c_enable, stack_choice,
                                        num_remove_out, winner, a_out, b_out, c_out);

endmodule

// Simple controller to enable and control turn changes
module controller(
```

```
        input                    reset, ph1, ph2,
        input  logic             turn_enable, winner,
        input  logic [3:0]      a_out, b_out, c_out,
        input  logic [2:0]       stack_choice_in,
        input  logic [3:0]       num_remove_in,
        output logic [2:0]      stack_choice_out,
        output logic [3:0]      num_remove_out,
        output logic             a_enable, b_enable, c_enable,
        output logic             human_win, ai_win, invalid);

        // Simple truth table logic to control turn processes

        // Define non-input wires
        logic turn, zero, not_validity, validity, turn_en, turn_old;
        logic [2:0] stack_choice_nim, stack_choice_human;
        logic [3:0] stack_out_nim, stack_out_human;

        // Store turn somewhere...
        flopenr #(1) turn_flop(ph1, ph2, reset, turn_en, turn_old, turn);

        always @ (*) begin
                if(reset == 1) begin                    // if reset is asserted,
                        a_enable = 1;                   // enable all three flip-flops
                        b_enable = 1;                   // (the datapath will send their initial values)
                        c_enable = 1;
                        turn_en = 0;                    // and make the AI go first
                end
                else if (invalid == 1) begin            // If invalid, turn stays the same
                        a_enable = 0;                   // and make sure all the flip-flops are disabled
                        b_enable = 0;
                        c_enable = 0;
                        turn_en = 0;
                end
                else begin                              // If not invalid and reset is not asserted,
                        a_enable = stack_choice_out[0]; // enable the correct flip-flop
                        b_enable = stack_choice_out[1];
                        c_enable = stack_choice_out[2];
                        turn_en = 1;                    // and change turn
                end
                turn_old = ~turn;
        end
        always @ (*) begin
                if (winner == 1) begin                  // if there was a winner...
                        human_win = turn;               // if turn = 1, then human wins (human_win = 1)
                        ai_win = ~turn;                 // if turn = 0, then AI wins (ai_win = 1)
                end
                else begin                              // if there was no winner, then neither side wins
                        human_win = 0;
                        ai_win = 0;
                end
        end

        // ai or human logic pathways
        nim_logic   l_nim(a_out,b_out,c_out,stack_choice_nim,stack_out_nim);

        human_logic l_human(    turn_enable,stack_choice_in,num_remove_in,
                                a_out,b_out,c_out,stack_choice_human,stack_out_human,validity);

        // set invalid output
        assign zero = 0;
        assign not_validity = ~validity;
        mux2 #(1) valid_mux(zero, not_validity, turn, invalid);


        // choose between human or ai turn inputs
        mux2 #(3) stack_choice_mux(stack_choice_nim, stack_choice_human, turn, stack_choice_out);
        mux2      stack_out_mux(stack_out_nim, stack_out_human, turn, num_remove_out);
endmodule
```

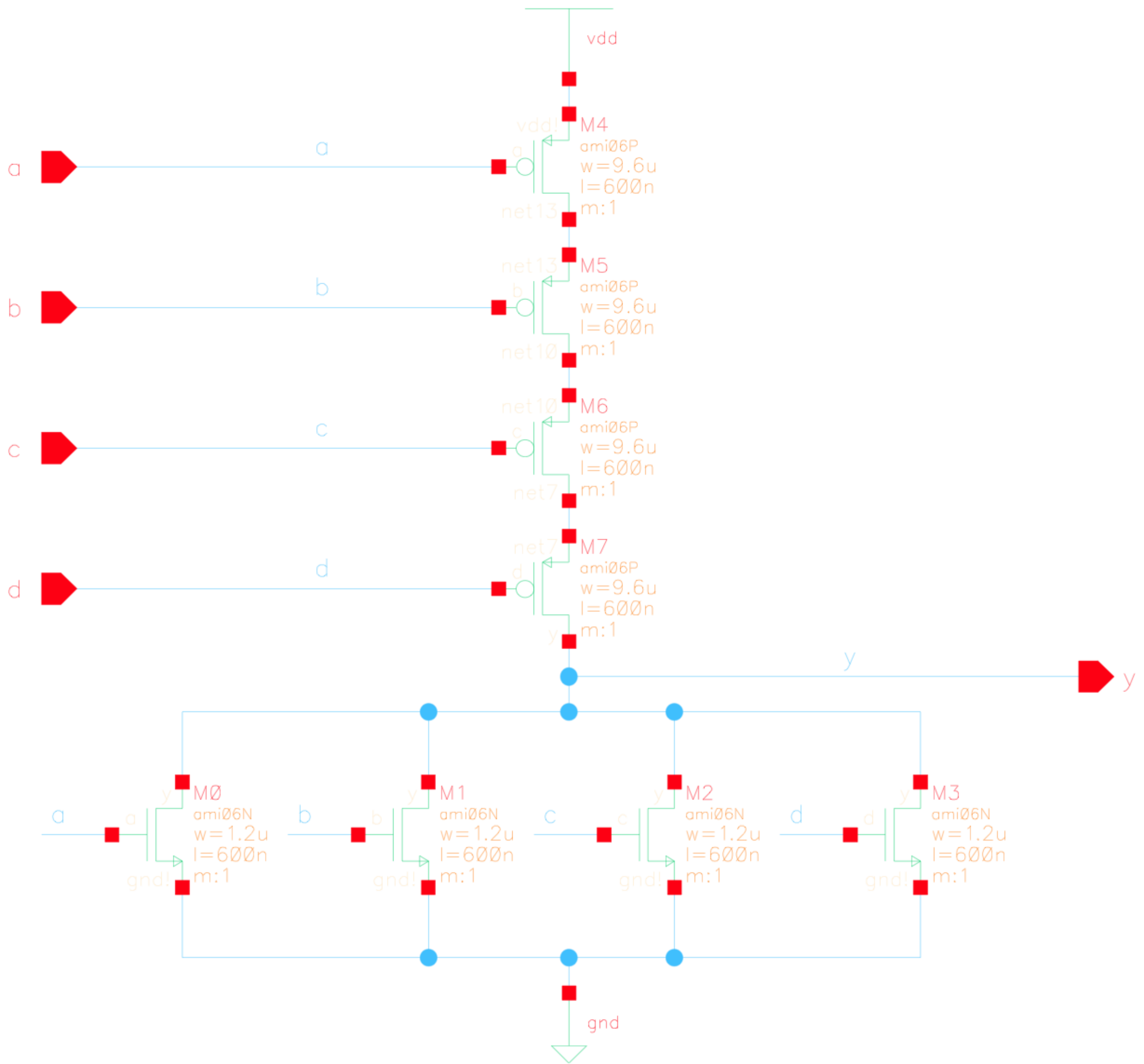
```
// Datapath handles turn logic and winner logic
module datapath(
        input  logic            ph1, ph2, reset,
        input  logic            a_enable, b_enable, c_enable,
        input logic [2:0]       stack_choice,
        input  logic [3:0]      stack_out,
        output logic            winner,
        output logic [3:0]      a_ct_out, b_ct_out, c_ct_out,
        output logic [3:0]      a_out, b_out, c_out);

        logic [3:0] a_in, b_in, c_in;
        logic [3:0] a_pass, b_pass, c_pass;
        logic [3:0] a_default, b_default, c_default;

        // initial values for the three stacks
        assign a_default = 4'b0011;
        assign b_default = 4'b0100;
        assign c_default = 4'b0101;

        // reset muxes
        mux2 resetmux_a(a_out, a_default, reset, a_pass);
        mux2 resetmux_b(b_out, b_default, reset, b_pass);
        mux2 resetmux_c(c_out, c_default, reset, c_pass);

        // flippy flops
        flopen stacka(ph1, ph2, a_enable, a_pass, a_ct_out);
        flopen stackb(ph1, ph2, b_enable, b_pass, b_ct_out);
        flopen stackc(ph1, ph2, c_enable, c_pass, c_ct_out);


        // make the move, update, and check for winners!
        make_move gen_move(stack_choice, stack_out, a_ct_out, b_ct_out, c_ct_out, a_out, b_out, c_out);
        check_winner chk_winner(reset, a_out, b_out, c_out, winner);

endmodule

// After legality is covered, update values
module make_move(
        input  logic [2:0]      stack_choice,
        input  logic [3:0]      stack_out,
        input  logic [3:0]      stacka, stackb, stackc,
        output logic [3:0]      a_new, b_new, c_new);

        // sets the chosen stack to the new value
        always @(*) begin
                case(stack_choice)
                        3'b001: begin a_new = stack_out; b_new = stackb; c_new = stackc; end
                        3'b010: begin a_new = stacka; b_new = stack_out; c_new = stackc; end
                        3'b100: begin a_new = stacka; b_new = stackb; c_new = stack_out; end
                        default: begin a_new = stacka; b_new = stackb; c_new = stackc; end
                endcase
        end
endmodule

module mux2 #(parameter WIDTH = 4)
      ( input  logic [WIDTH-1:0] d0, d1,
        input  logic                     s,
        output logic [WIDTH-1:0] y);

  always_comb
    case (s)
      0: y = d0;
      1: y = d1;
    endcase
endmodule

// check winner logic
module check_winner(
        reset,
```

```
        stacka,
        stackb,
        stackc,
        winner);

        //inputs and outputs

        input reg reset;
        input reg [3:0] stacka, stackb, stackc;
        output reg winner;



// if reset is being asserted, then there is no winner
// otherwise, if all stacks are empty, then someone has won the game.
        always @(*) begin
                if( reset == 1 ) begin,
                        winner = 0;
                end
                else if( (stacka == 0) && (stackb == 0) && (stackc == 0)) begin
        winner = 1;
                end
                else begin
                        winner = 0;
                end
        end
endmodule

// ai logic
module nim_logic(stacka, stackb, stackc, stack_choice, stack_out);
        // input-output ports
        input reg [3:0] stacka, stackb, stackc;
        output reg [2:0] stack_choice;          // the stack that will be changed
        output reg [3:0] stack_out;             // new value of stack that will be changed

        reg [3:0] x, a_prime, b_prime, c_prime;

        // XOR logic

        always @ (*) begin

                x = (stacka^stackb)^stackc;     // take bitwise xor of the three stacks to get nim sum
                a_prime = x^stacka;             // next, take the bitwise xor of the nim sum
                b_prime = x^stackb;             // and each of the three stacks
                c_prime = x^stackc;             // resulting numbers give target size for each stack,
                                                // but not each one will correspond to a valid move
                if(a_prime < stacka) begin      // if the target size for stack a is less than
                                                // the current size of stack a,
                        stack_out = a_prime;    // reduce stack a to its target size
                        stack_choice = 3'b001;
                end

                else if(b_prime < stackb) begin  // likewise
                        stack_out = b_prime;
                        stack_choice = 3'b010;
                end

                else if(c_prime < stackc) begin  // likewise
                        stack_out = c_prime;
                        stack_choice = 3'b100;
                end

                // If none of those three moves were valid, we remove one bead from the first non-empty
                // stack and hope the human makes a mistake

                else if(stacka != 0) begin
                        stack_out = stacka - 1;
                        stack_choice = 3'b001;
```

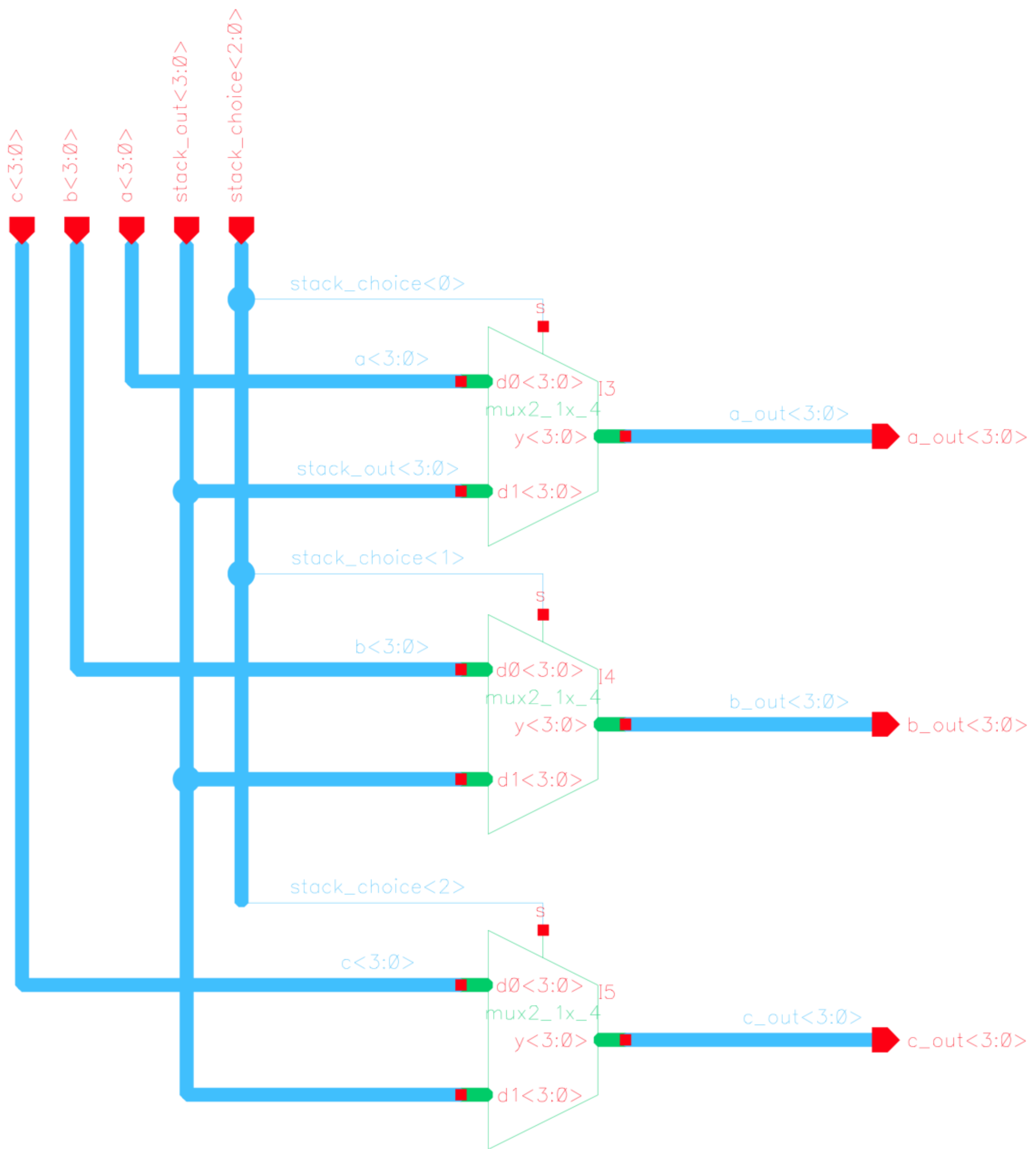
```
		end

		else if(stackb != 0) begin
			stack_out = stackb - 1;
			stack_choice = 3'b010;

		end

		else if(stackc != 0) begin
			stack_out = stackc - 1;
			stack_choice = 3'b100;

		end
		else begin	// should never happen (this module will only be called when at least one
				// of the stacks is still non-empty)

			stack_out = stacka;
			stack_choice = 3'b001;
		end
	end
endmodule


// Takes in human input, checks to see if it is valid
// If invalid, passes through original stack value
// If valid, passes through new stack value
module human_logic(
	turn_enable,
	stack_choice_in,
	num_remove_in,
	stacka,
	stackb,
	stackc,
	stack_choice,
	stack_out,
	validity);

	input reg turn_enable;
	input reg [2:0] stack_choice_in;
	input reg [3:0] num_remove_in, stacka, stackb, stackc;
	output reg [2:0] stack_choice;
	output reg [3:0] stack_out;
	output reg validity;

	reg [3:0] stack_size;

	always @(*) begin
		case (stack_choice_in) // take size of the user's chosen stack and wire it to stack_size
			3'b001: stack_size = stacka;
			3'b010: stack_size = stackb;
			3'b100: stack_size = stackc;
			default: validity = 0; // if the user is stupid and asks for an invalid stack,
					       // the move is automatically invalid.
		Endcase

		// if the user's chosen stack contains less beads than the user is trying to remove
		// or if the user tries to remove 0 beads, then the move isn’t valid.

		if( (stack_size < num_remove_in) || (num_remove_in == 0) ) begin
			validity = 0;
		end
		else begin
			validity = 1;	// otherwise, the move is valid.
		end
		// if the user input was valid, pass the user's chosen stack to the output stack_choice
		// and subtract to find the new value of the stack that was chosen.
		if ((turn_enable == 1 ) && (validity == 1)) begin
			stack_choice = stack_choice_in;
			stack_out = stack_size - num_remove_in;
```

```
                end
                else begin                      // if move was invalid or if enable was not asserted,
                        stack_choice = 3'b000;  // don't choose any stack
                        stack_out = 0;          // and this shouldn't matter
                end
        end
endmodule

// *************************** flip flop stuff ***************************
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 flopen #(parameter WIDTH = 4)
               (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 flop #(parameter WIDTH = 4)
             (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 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 latch #(parameter WIDTH = 4)
              (input  logic             ph,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_latch
    if (ph) q <= d;
endmodule
// ******************** end of flip flop stuff *********************************
```

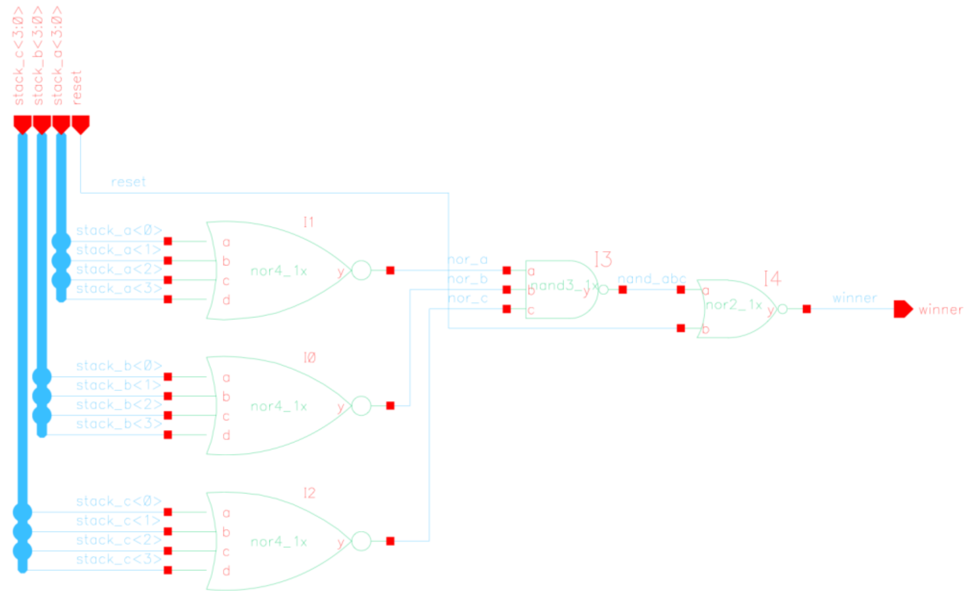
Appendix 2: Schematics



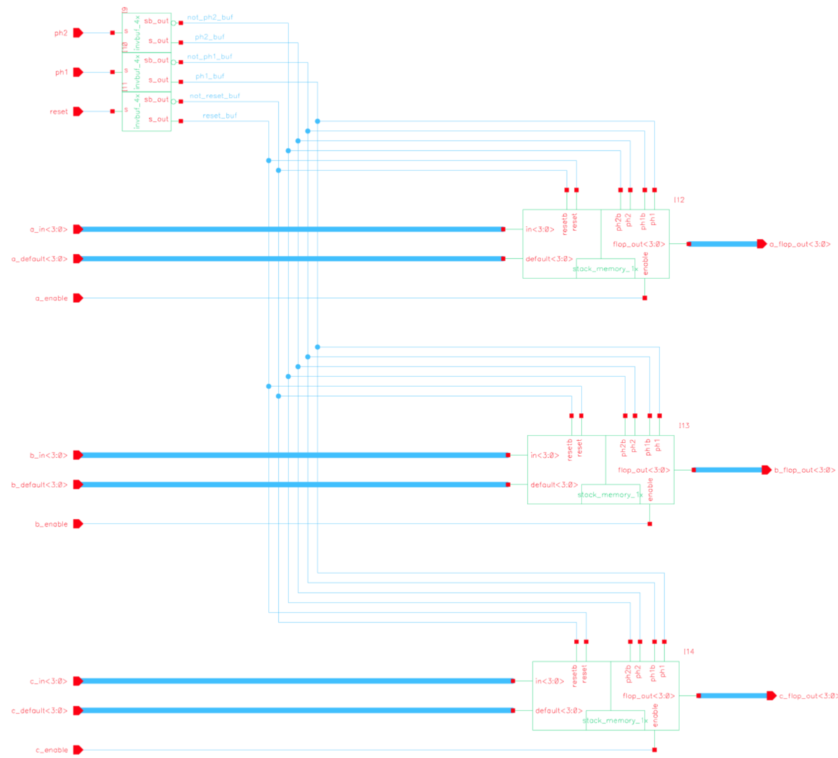
Schematic 1: nor4 leaf cell



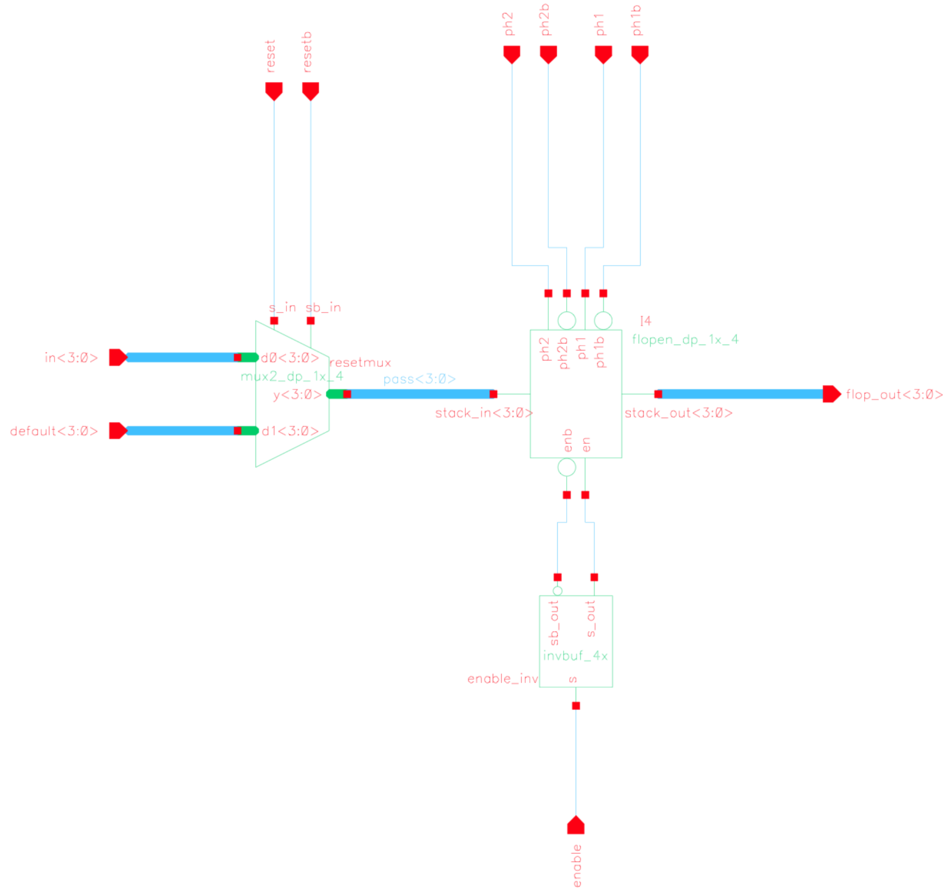
Schematic 2: make_move



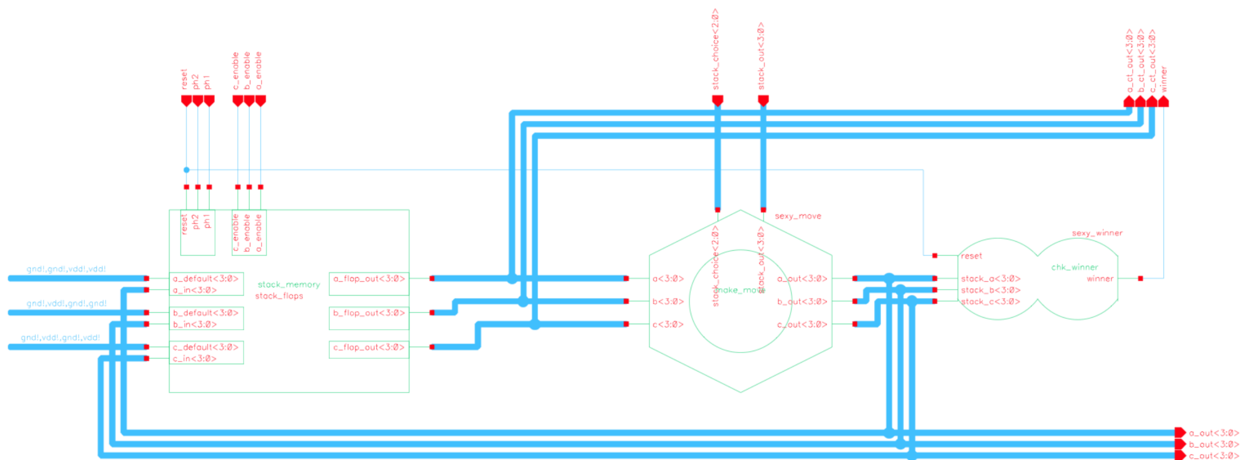
Schematic 3: check_winner



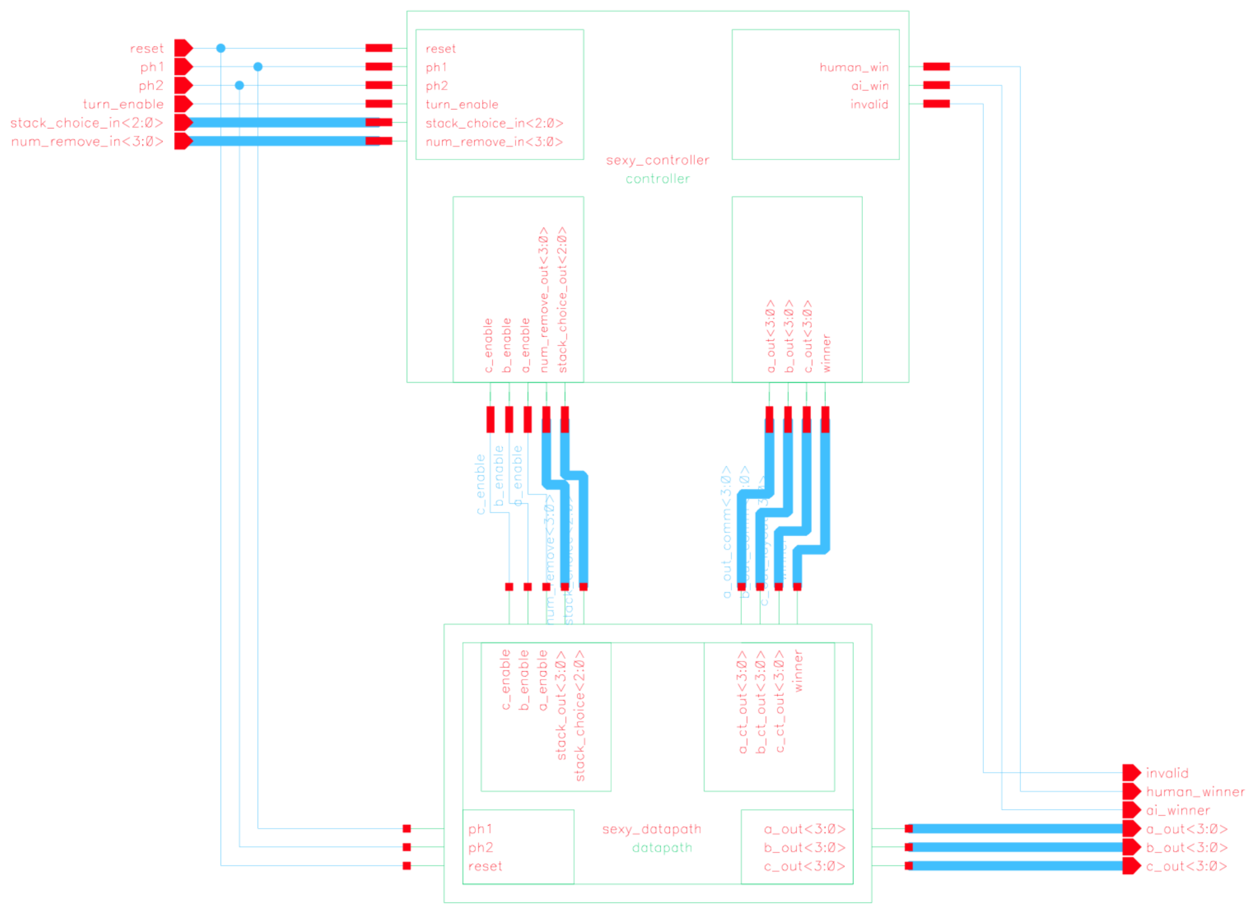
Schematic 4: stack_memory



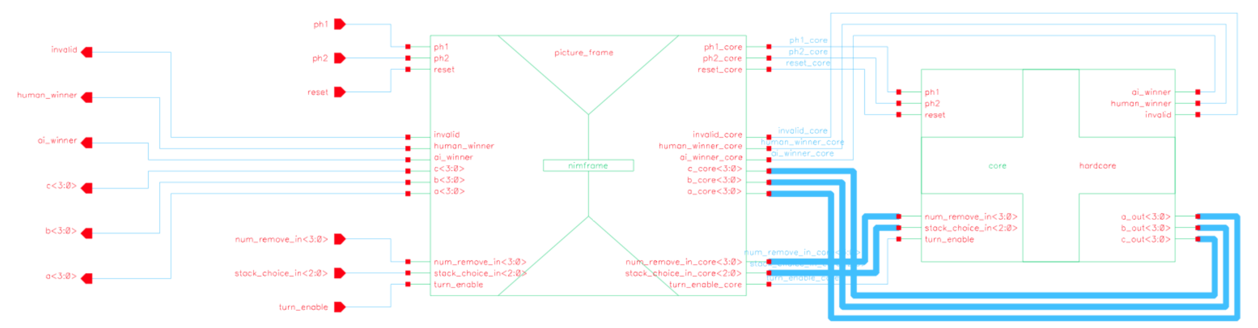
Schematic 5: stack_memory_1x



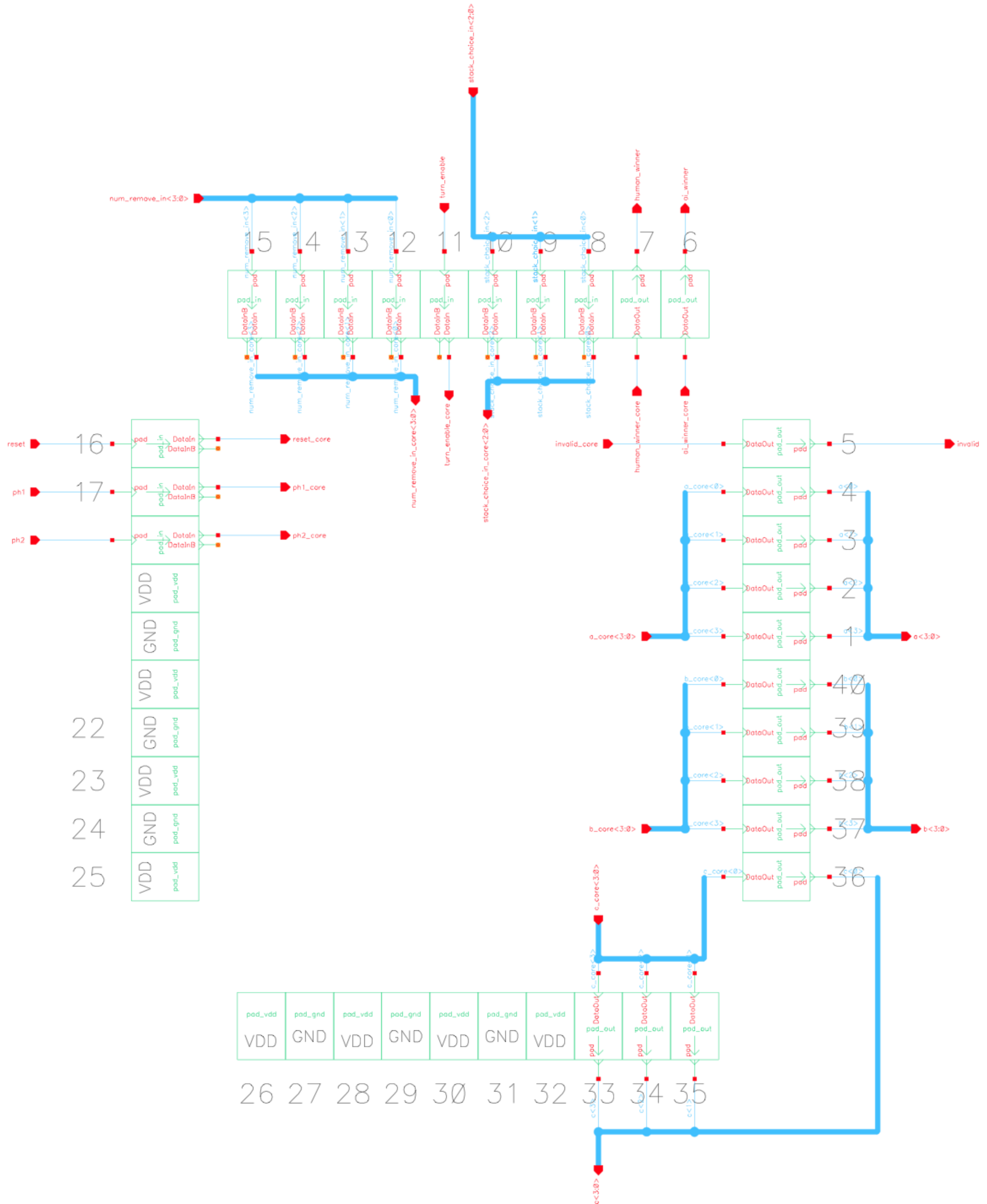
Schematic 6: datapath



Schematic 7: core

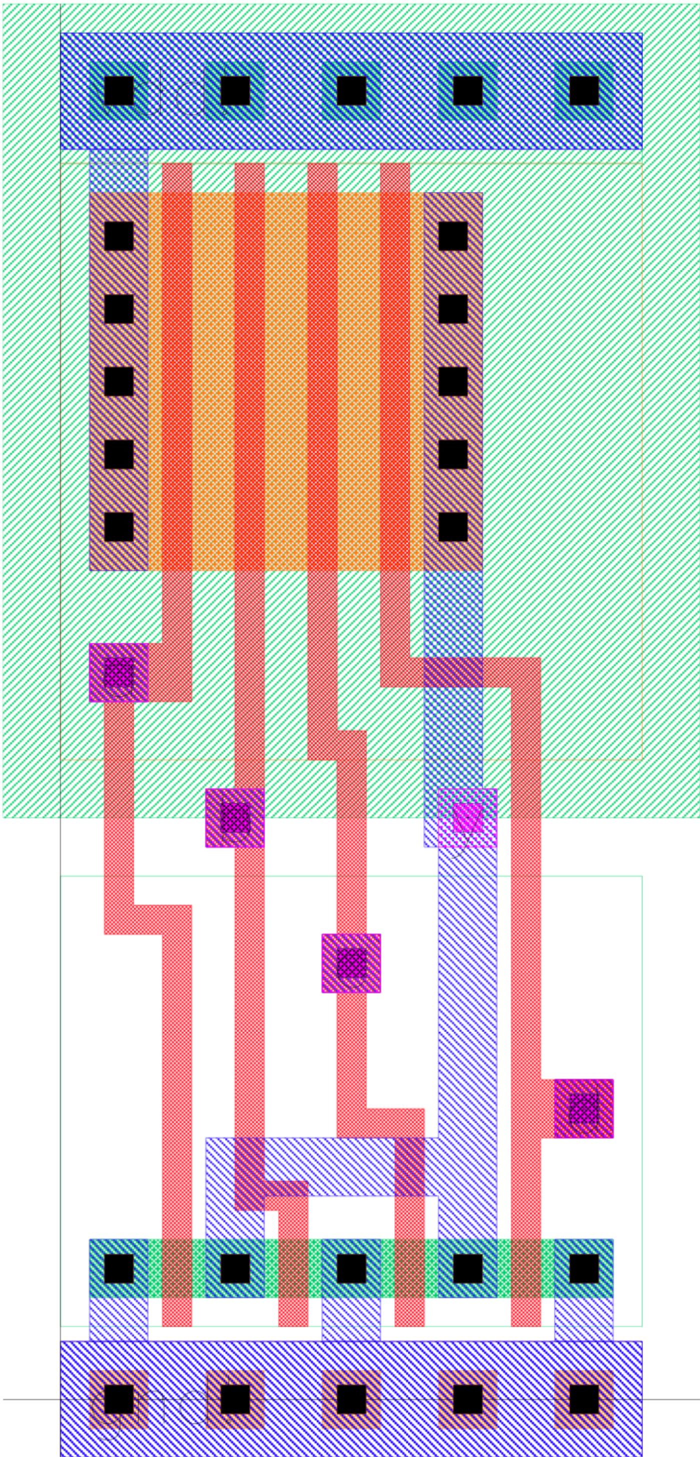


Schematic 8: chip

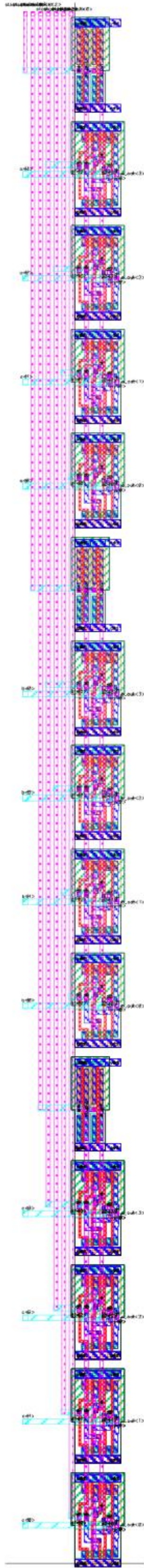


Schematic 9: nimframe

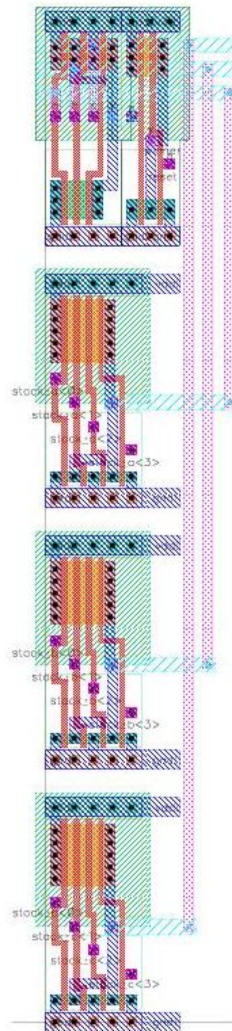
Appendix 3: Layouts



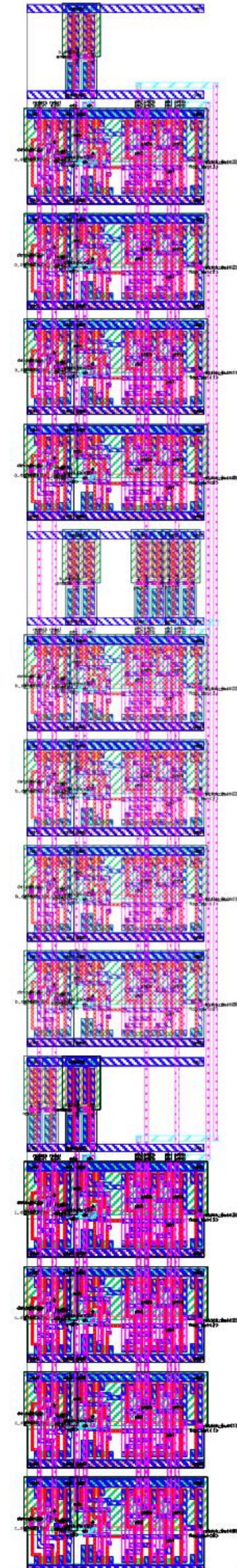
Layout 1: nor4



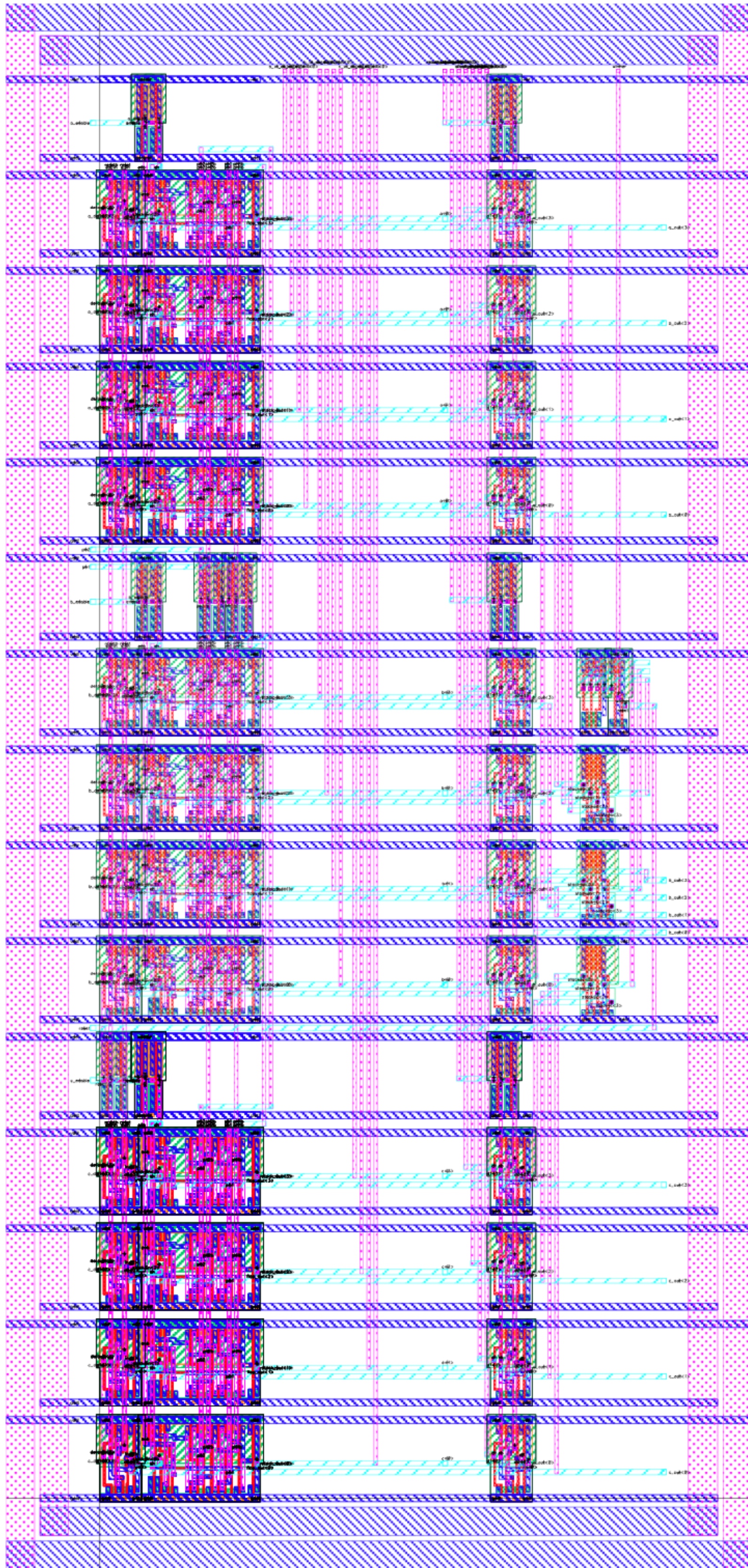
Layout 2: make_move



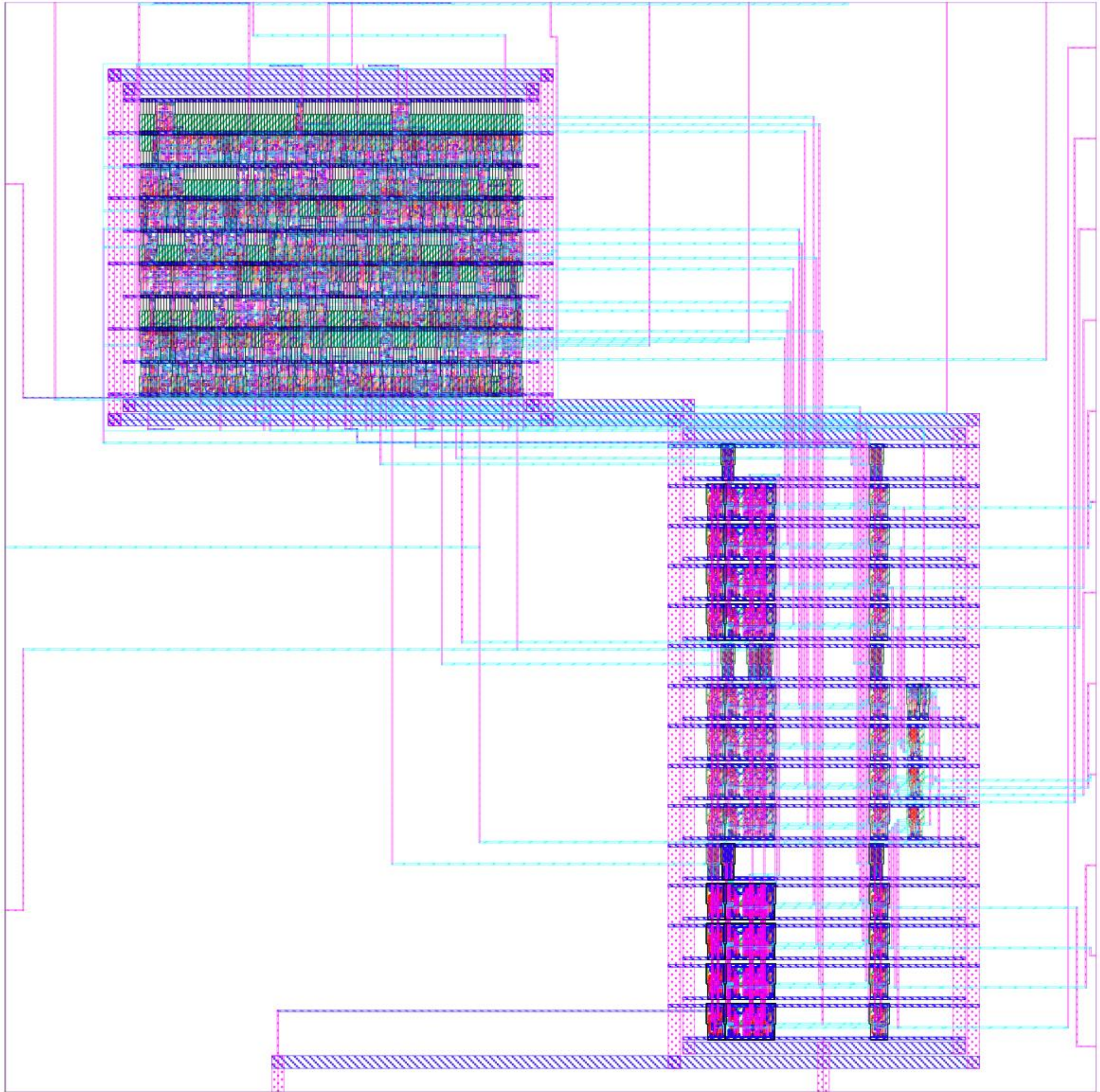
Layout 3: check_winner



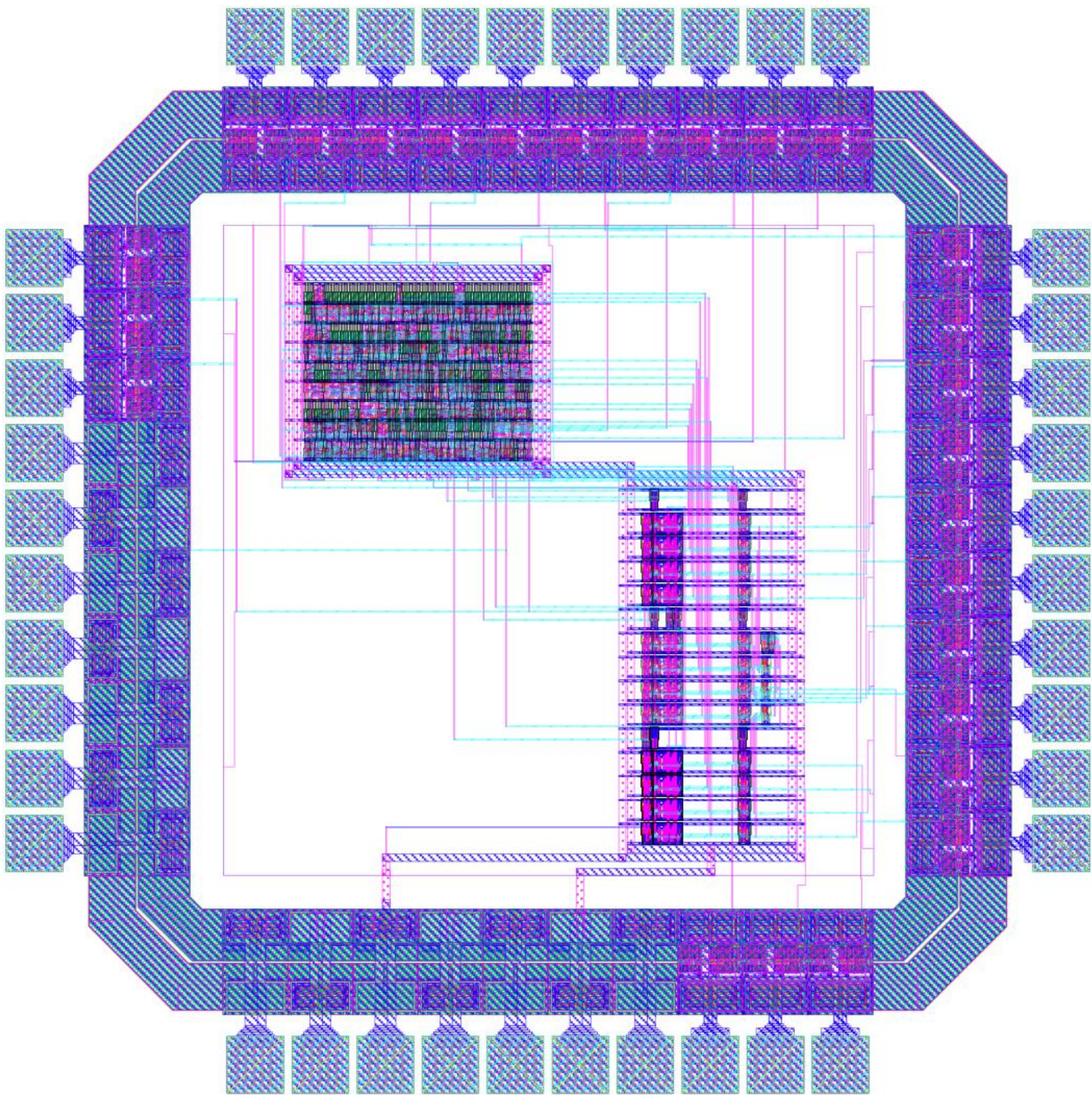
Layout 4: stack_memory



Layout 5: datapath



Layout 6: core



Layout 7: chip