

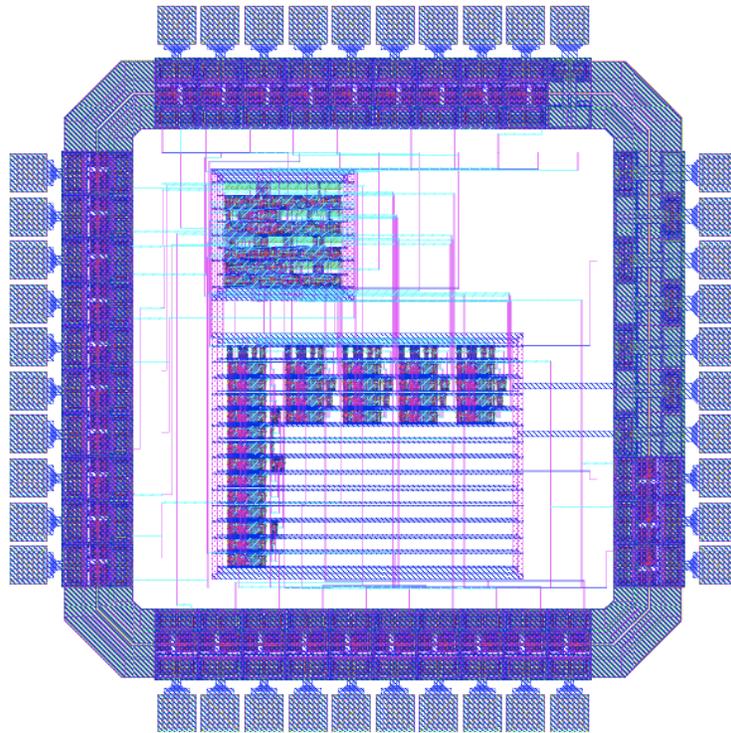
# VLSI Project: Stopwatch Chip

Final Project Report

April 19th, 2010

E158

Chris Ferguson and Matthew Keeter



## Introduction

Our final project for VLSI was a chip to control a stopwatch-style timer. The chip requires a clock of a known frequency to control the timing of the chip. The chip shows its output on four common-cathode 7-segment displays, displaying a time between 00.00 and 99.99 seconds. This layout for this chip consists of a custom layout to count the number of clock ticks, hundredths, tenths, seconds, and tens of seconds; furthermore, a synthesized controller converts counter values into 7-segment display outputs and tracks the mode of the system. This chip was designed in the AMI 0.5 micron process ( $\lambda = 0.3$ ) for a 40-pin DIP package. It was developed using the Cadence software suite, and NCSim was used for logic verification.

## Specifications

The chips has the following pins:

Name	Type	Number of pins	Description
Vdd	Bidirectional	3	Global power
Gnd	Bidirectional	3	Global ground
$\phi_1$	Input	1	First clock
$\phi_2$	Input	1	Second clock
reset	Input	1	System reset
startstop	Input	1	Timer start/stop
tens[6:0]	Output	7	Control for 10s digit of timer. (Each bit represents 1 segment of the display)
secs[6:0]	Output	7	Control for 1's digit of timer
tenths[6:0]	Output	7	Control for 0.1's digit of timer
hundredths[6:0]	Output	7	Control for 0.01's digit of timer

In order to function properly, the input clock frequency needs to be at 500 kHz. The number of clock ticks in each 0.01 seconds is hardwired into the system assuming an input clock of 500 kHz. Given this input clock, the chip should function as follows: Whenever the reset button is pressed, the chip reads 00.00 and stays at that value. When the Start/Pause button is pressed, if the timer is paused, it will begin displaying numbers increasing by 0.01 on the output display. If the timer is currently counting, pressing the Start/Pause button will cause it to pause and continue displaying whatever time it had most recently reached. The outputs of the chip consist of four 7-bit busses, each designed to control a common-cathode seven-segment display of the type shown in Figure 1.

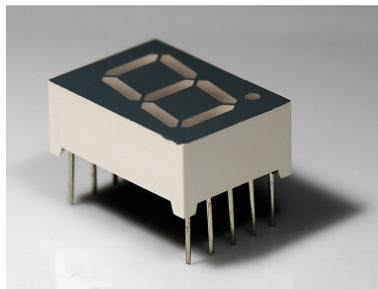


Figure 1: Seven-segment display (picture by Peter Halasz)

## Floorplan

The chip has two main components: the controller, which is synthesized logic, and the datapath, which was laid out by hand. The proposed floorplan (left) is compared with the actual floorplan (right) in Figure 2.

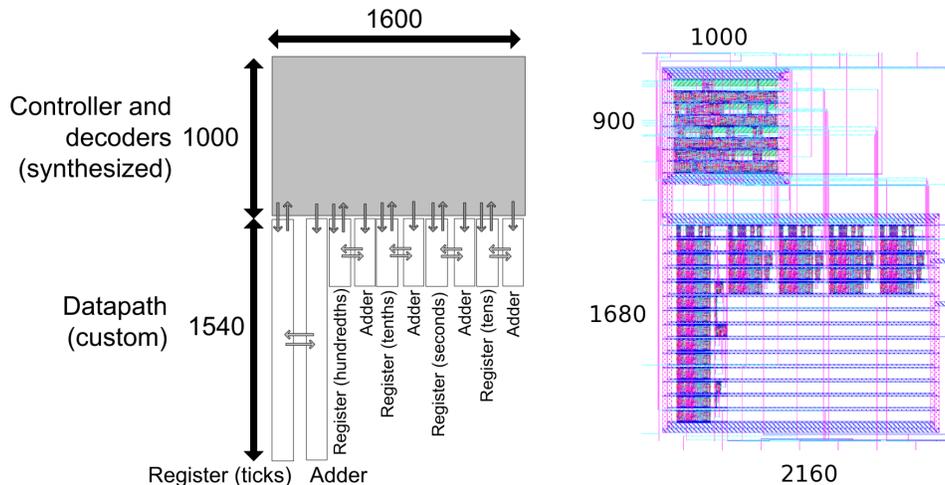


Figure 2: Comparison of floorplans (units of lambda, scales are different)

There are several discrepancies between the two floorplans. The stopwatch requires comparison logic that checks the value of each register, to detect if the register needs to roll over and increment the next register. In the proposal, this logic was included in the synthesized controller. However, this logic is highly regular and reliant on datapath register values, so the final design includes these comparators as part of the datapath. These comparators increased the width of the datapath, and a small amount of zipper logic increased the height. These changes also produced a much smaller controller than originally planned.

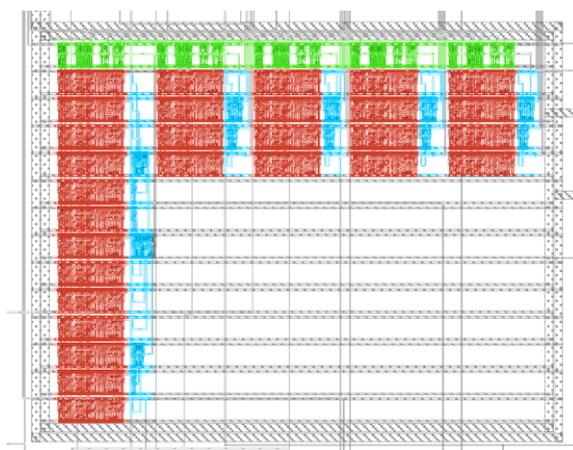


Figure 3: Datapath (red = counter, blue = comparator, green = zipper logic)

The datapath (slice plan shown in Figure 3) is a series of counters and comparators. The first counter (far left red block) is 13 bits tall and counts ticks of the 500 kHz clock. The first comparator

(left-most blue block) triggers when this counter reaches 4999, which means that the hundredths counter needs to roll over on the next clock cycle. The next four counter/comparator blocks are identical: a 4-bit counter, and a comparator that triggers when the counter reaches 9. The four identical counter-comparator blocks count hundredths, tenths, seconds, and tens of seconds. Each comparator both resets its counter and increments the next counter when it triggers.

The system has the pinout shown in Figure 4.

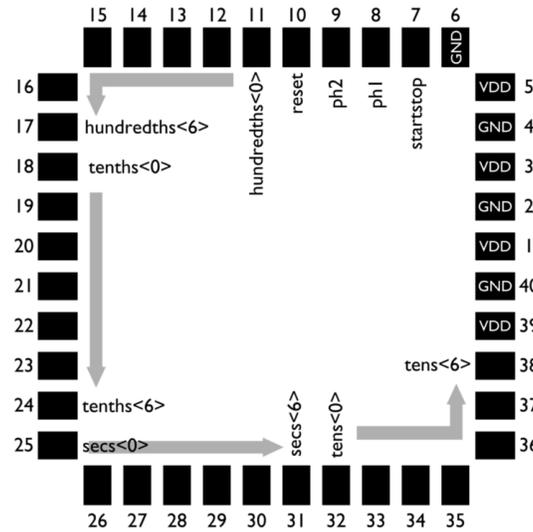


Figure 4: Pinout diagram (grey arrows indicate increasing bits of a bus)

## Verification

The system was tested on several levels: the Verilog model was simulated, as was the schematic; and the layout was compared to the schematic.

The Verilog describing the system (see Appendix A) simulates properly in the self-checking testbench, using `nc-sim`. The schematics (see Appendix B) pass the same testbench. The layout (see Appendix C) passes DRC and LVS, with “Compare FET Parameters” turned off. This option must be turned off due to quirks in padframe extraction. The core passes LVS with “Compare FET Parameters” turned on. The CIF imports correctly, and passes DRC and LVS (with “Compare FET Parameters” turned off, for the same reasons as before). There are no discrepancies or concerns with the chip verification process.

## Test Plan

To test the chip, we would attach four seven-segment displays to the correct output pins on the FPGA. A start-stop and reset button would also be attached to the correct pins, possibly with debouncing circuitry attached in between the button and the chip. Finally, we need a two-phase 500 kHz clock to run the system with correct timing. This could be generated by a signal generator and analog circuitry to generate a delay, or by a digital system running at a higher speed.

Once everything is attached to the correct pins, the system can be tested by resetting it, then starting the timer at the same time as a different stopwatch. When they are both stopped

simultaneously, they should read the same value, within a margin related to clock frequency error. The system could also be tested with a slower clock to check the accuracy of the tenths and hundredths digit: using a 50 kHz clock, we expect it to run such that the tenths digit corresponds to seconds.

## Design Time

*Proposal and Floorplan:* 4 hours

*Verilog and Testbench:* 5.5 hours

*Schematics:* 8 hours

*Layout:* 17 hours

## File Locations

All of the files are located on `chips.eng.hmc.edu`, the VLSI server for Harvey Mudd College.

*Verilog code:* `~cferguson/proj2/timerchip_2.v` and `~cferguson/proj2/testbench.v`

*Test vectors:* None (contained within testbench)

*Synthesis results:* `~cferguson/proj2/chip_final/`

*Cadence Libraries:* `~cferguson/IC_CAD/cadence/`

*CIF File:* `~cferguson/proj2/timerchip.cif`

*PDF chip plot:* `~cferguson/proj2/fullchip.pdf`

*PDF of this report:* `~cferguson/proj2/finalreport.pdf`

## Appendix A: Verilog Code

## timerchip\_2.v

```
// Chris Ferguson and Matt Keeter
// cferguson@hmc.edu mkeeter@hmc.edu
// E158 Project 2
// Spring 2010

//Timing clk frequency is 0.5 MHz
//.01 second = 5000 clock ticks
//needs 13 bit counter for initial counting.

// Our project is an ASIC for a stopwatch. It should either be counting upwards
// or paused, and output a time on four seven-segment displays
// (hundredths through tens of seconds)
module timerchip_2(input ph1, ph2, reset, startstop,
    output [6:0] hundredths, tenths, secs, tens);

    wire mode;
    wire [3:0] hundredths_val, tenths_val, secs_val, tens_val;

    datapath datapath(ph1,ph2, reset, startstop, mode,
        hundredths_val, tenths_val, secs_val, tens_val);

    controller controller(ph1, ph2, reset, startstop,
        hundredths_val, tenths_val, secs_val, tens_val,
        mode, hundredths, tenths, secs, tens);
endmodule

//Datapath containing all of the regular logic of the timer
//takes in inputs, and 'mode' from controller
//outputs value for each of the digits
module datapath( input ph1, ph2, reset, startstop, mode,
    output [3:0] hundredths_val, tenths_val, secs_val, tens_val);

    wire [4:0] enables, resets, compares;

    wire [12:0] ticks_val; // note: this could change depending on clock frequency

    // counters, consisting of a register and incrementor (custom logic)
    counter #(13) ticks_reg(ph1, ph2, resets[0], enables[0], ticks_val);
    counter #(4) hundredths_reg(ph1, ph2, resets[1], enables[1], hundredths_val);
    counter #(4) tenths_reg(ph1, ph2, resets[2], enables[2], tenths_val);
    counter #(4) secs_reg(ph1, ph2, resets[3], enables[3], secs_val);
    counter #(4) tens_reg(ph1, ph2, resets[4], enables[4], tens_val);
    // a set of comparator to detect when each value should roll over (custom logic)
    comparator #(13) ticks_comp(ticks_val, 13'd4999, compares[0]);
    comparator #(4) hundredths_comp(hundredths_val, 4'd9, compares[1]);
    comparator #(4) tenths_comp(tenths_val, 4'd9, compares[2]);
    comparator #(4) secs_comp(secs_val, 4'd9, compares[3]);
    comparator #(4) tens_comp(tens_val, 4'd9, compares[4]);

    //Enables calculation to be in zipper of datapath
```

```

assign enables[0] = mode;
assign enables[1] = enables[0] && compares[0];
assign enables[2] = enables[1] && compares[1];
assign enables[3] = enables[2] && compares[2];
assign enables[4] = enables[3] && compares[3];
assign enables5 = enables[4] && compares[4];
//Resets also contained in zipper!
assign resets[0] = enables[1] || reset;
assign resets[1] = enables[2] || reset;
assign resets[2] = enables[3] || reset;
assign resets[3] = enables[4] || reset;
assign resets[4] = enables5 || reset;

endmodule

//controller contains our irregular logic. It consists of:
//a mode calculation (paused or unpaused)
//and 4 seven-segment displays for the output
module controller(input ph1, ph2, reset, startstop,
    input [3:0] hundredths_val, tenths_val, secs_val, tens_val,
    output mode,
    output [6:0] hundredths, tenths, secs, tens);
//calculation to see if timing is stopped or not!
wire newmode; // mode = 1 when counting up, 0 when paused
wire helddown;
assign newmode = (startstop && !helddown) ? ~mode : mode;
flop #(1) modeflop(ph1, ph2, reset, newmode, mode);
flop #(1) heldflop(ph1, ph2, reset, startstop, helddown);

//output calculation! lots of ugly combinational logic
sevenseg hundredths_7seg(hundredths_val, hundredths);
sevenseg tenths_7seg(tenths_val, tenths);
sevenseg secs_7seg(secs_val, secs);
sevenseg tens_7seg(tens_val, tens);

endmodule

// controller module: keeps track of the current mode (paused or counting),
// and tells each register to increment, stay the same, or reset itself

// simple counter: counts up when enabled
module counter #(parameter WIDTH = 8)
    (input ph1, ph2, reset, enable, output [WIDTH-1:0] counterout);

    wire [WIDTH-1:0] newval;
    flop #(WIDTH) countflop(ph1, ph2, reset, newval, counterout);
    incremter #(WIDTH) countinc(counterout, enable, newval);
endmodule

// incremter: returns value in + 1 if increment is high, or value in if increment is low
module incremter #(parameter WIDTH = 8)
    (input [WIDTH-1:0] valin, input increment, output [WIDTH-1:0] valout);
    assign valout = valin + increment;

```

```

endmodule

// comparator: checks if two values are equal
module comparator #(parameter WIDTH = 8)
    (input [WIDTH-1:0] val1, val2, output equals);
    assign equals = (val1 == val2);
endmodule

// Chris Ferguson
// cferguson@hmc.edu
// 9/18/09
//
// 7-segment control module:
// This module is designed to take a 4 binary digit input, aka a hex character,
// and then show this letter on a 7-segment display.

module sevenseg(
    input [3:0] s,    //4-digit binary number as input
    output reg [6:0] segs //number segments to be lit up
    );

//THESE CONTROLS ARE FOR A COMMON CATHODE DISPLAY. To use them for
// a common anode display, take away the ~s

always @(*)
    case (s)    //Going to enumerate all 4-digit binary numbers
        //ordered [G: A]
        4'b0000: segs <= ~7'b0111111; // hex 0
        4'b0001: segs <= ~7'b0000110; // hex 1
        4'b0010: segs <= ~7'b1011011; // hex 2
        4'b0011: segs <= ~7'b1001111; // hex 3
        4'b0100: segs <= ~7'b1100110; // hex 4
        4'b0101: segs <= ~7'b1101101; // hex 5
        4'b0110: segs <= ~7'b1111101; // hex 6
        4'b0111: segs <= ~7'b0000111; // hex 7
        4'b1000: segs <= ~7'b1111111; // hex 8
        4'b1001: segs <= ~7'b1100111; // hex 9
        4'b1010: segs <= ~7'b1110111; // hex A
        4'b1011: segs <= ~7'b1111100; // hex B
        4'b1100: segs <= ~7'b1011000; // hex C
        4'b1101: segs <= ~7'b1011110; // hex D
        4'b1110: segs <= ~7'b1111001; // hex E
        4'b1111: segs <= ~7'b1110001; // hex F
        default: segs <= ~7'b0000001; // dummy default. Shouldn't be used.
    endcase

// The segs digits correspond to the segments as shown below
//      t[0]
//      ----
//      |   |t[1]
// t[5] |t[6]|
//      ----
//      |   |t[2]
// t[4] |   |

```

```

//      ----
//      t[3]
// (note that t[6] is the interior horizontal segment)
endmodule

// code below this is based on the verilog model of the mips processor
// by David Money Harris, converted back into verilog from SystemVerilog
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 latch #(parameter WIDTH = 8)
    (input      ph,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

    always @(*)
        if (ph) q <= d;
endmodule

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

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

    assign resetval = 0;

    mux2 #(WIDTH) enrmux(d, resetval, reset, d2);
    flop #(WIDTH) f(ph1, ph2, d2, q);
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

```

## testbench.v

```

`timescale 100ns / 100ns

module testbench();
    // define parameters of 7 segment display outputs

```

```

parameter ZERO = ~7'b0111111;
parameter ONE = ~7'b0000110;
parameter TWO = ~7'b1011011;
parameter THREE = ~7'b1001111;
parameter FOUR = ~7'b1100110;
parameter FIVE = ~7'b1101101;
parameter SIX = ~7'b1111101;
parameter SEVEN = ~7'b0000111;
parameter EIGHT = ~7'b1111111;
parameter NINE = ~7'b1100111;

reg ph1, ph2;

reg reset, startstop;
wire [6:0] hundredths, tenths, secs, tens;

reg errordetector;

// initialize test
initial
begin
    // start up the stopwatch
    errordetector<=0;
    reset <= 1; startstop <= 0; # 40;
    reset <= 0; # 40;
    startstop <= 1; #40;
    startstop <= 0;
    #54300040;//@5.43 sec, timer 5.43
    if(tens != ZERO || secs != FIVE || tenths != FOUR || hundredths != THREE)
    begin
        $display("Error in first test (starting)");
        errordetector<=1;
    end else $display("First test completed successfully");

    // keep running for a known amount of time, then check it
    #21400000;//@7.57 sec, timer 7.47
    if(tens != ZERO || secs != SEVEN || tenths != FIVE || hundredths != SEVEN)
    begin
        $display("Error in second test (counting)");
        errordetector<=1;
    end else $display("Second test completed successfully");

    // pause the counter, and make sure it maintains its old value
    #11100000;//@8.68 sec, timer 8.68
    startstop<=1; #02500000; //Pauses counting
    startstop<=0; //Button not pressed down anymore
    #02500000;//@9.18 sec, timer 8.68
    if(tens != ZERO || secs != EIGHT || tenths != SIX || hundredths != EIGHT)
    begin
        $display("Error in third test (pausing)");
        errordetector<=1;
    end else $display("Third test completed successfully");

    // restart the counter, and make sure it keeps counting up

```

```

startstop<=1;
#10000000; //button pressed for 1 sec
startstop<=0;
#17400000; //@ 11.92 sec, timer 11.42
if(tens != ONE || secs != ONE || tenths != FOUR || hundredths != TWO)
begin
    $display("Error in fourth test (restarting)");
    errordetector<=1;
end else $display("Fourth test completed successfully");

// test holding down startstop and reset, make sure that reset wins
startstop<=1;
#05000000;
reset <= 1;
#05000000; //@12.92 sec, timer = 0
if(tens != ZERO || secs != ZERO || tenths != ZERO || hundredths != ZERO)
begin
    $display("Error in fifth test (multi-button press)");
    errordetector<=1;
end else $display("Fifth test completed successfully");

// toggle startstop while reset is held down, make sure reset still wins
startstop<=0;
#05000000;
startstop<=1;
#05000000; //@13.92 sec, timer still = 0
startstop<=0;
if(tens != ZERO || secs != ZERO || tenths != ZERO || hundredths != ZERO)
begin
    $display("Error in sixth test (pressing SS while reset is held)");
    errordetector<=1;
end else $display("Sixth test completed successfully");

// make sure that the digit 9 works properly
reset<=0;
#01000000;
startstop<=1; #20;
#09000020; //@14.92 sec, timer = 0.9
if(tens != ZERO || secs != ZERO || tenths != NINE || hundredths != ZERO)
begin
    $display("Error in seventh test (testing nine) %b %b",tenths, hundredths);
    errordetector = 1;
end else $display("Seventh test completed successfully");

if(errordetector == 1)
    $display("Error(s) detected. Please fix them.");
else
    $display("No errors detected. Good job!");

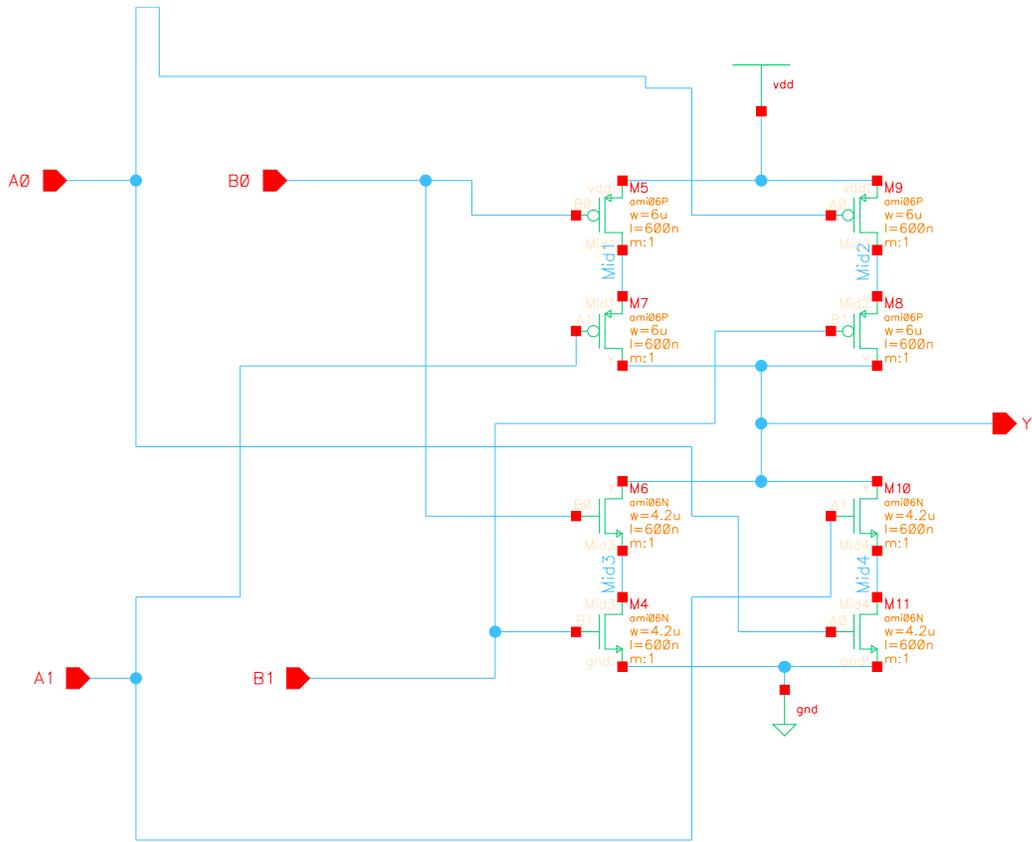
end
// generate clock to sequence tests
always
begin
    ph1 <= 0; ph2 <= 0; # 2;

```

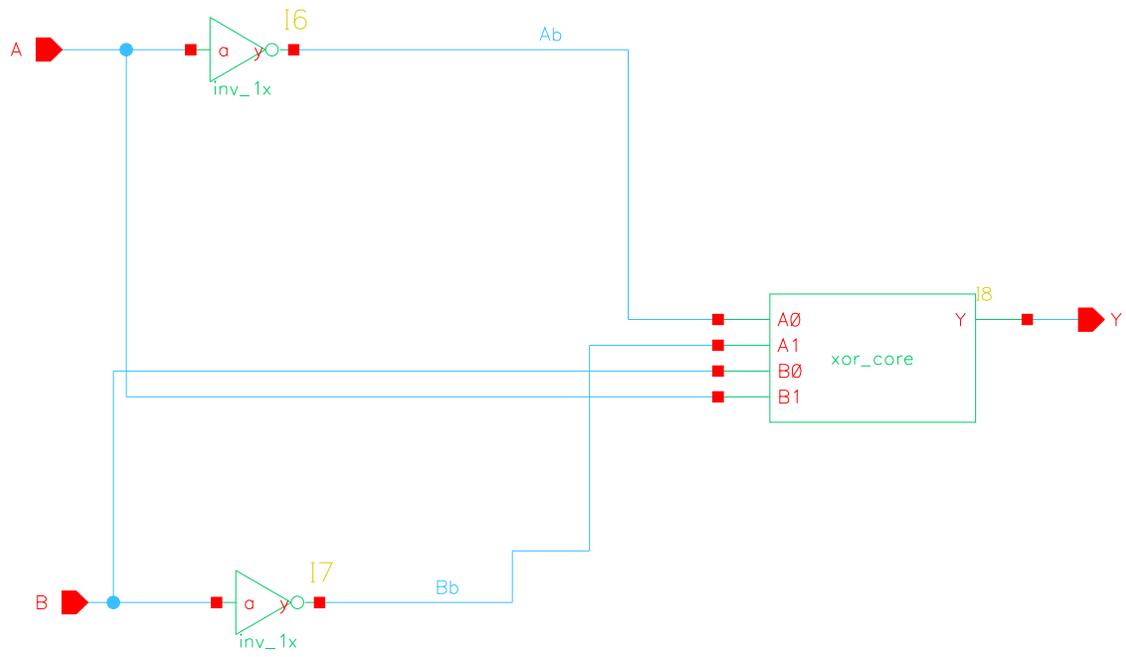
```
    ph1 <= 1; # 8;
    ph1 <= 0; # 2;
    ph2 <= 1; # 8;
end

    timerchip_2 dut(ph1, ph2, reset, startstop, hundredths, tenths, secs, tens);
endmodule
```

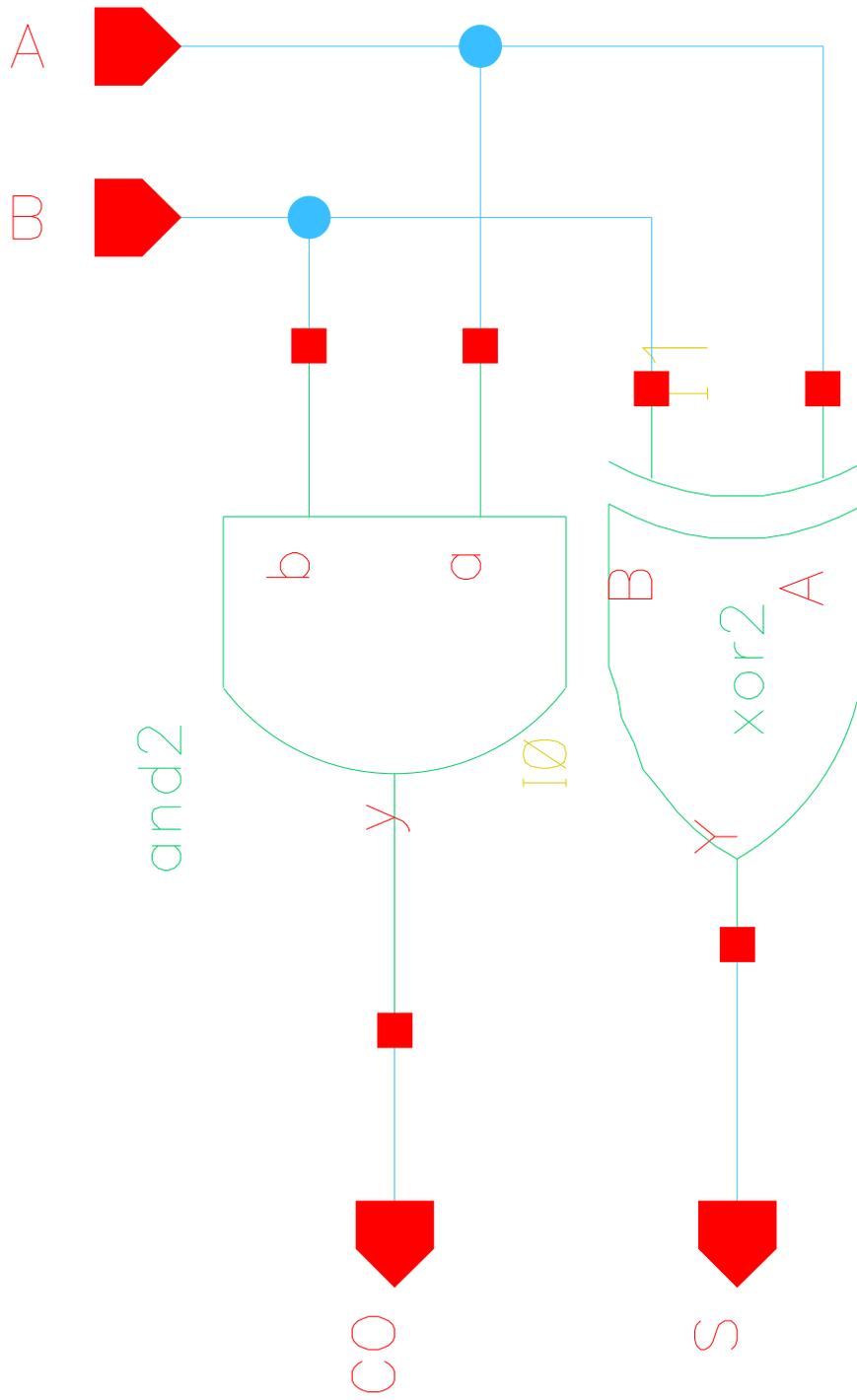
## Appendix B: Schematics



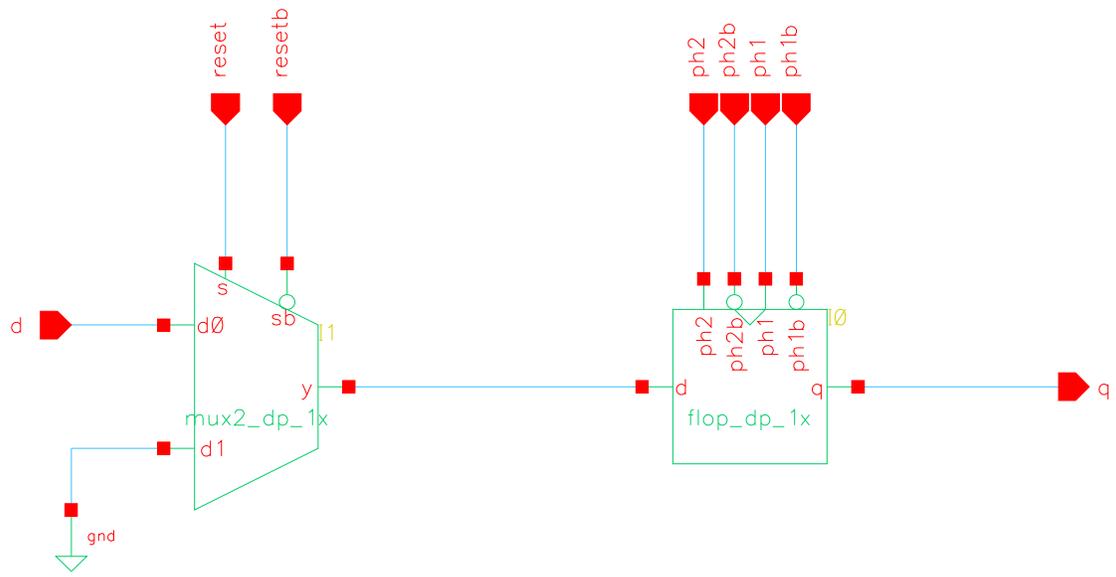
XOR\_CORE



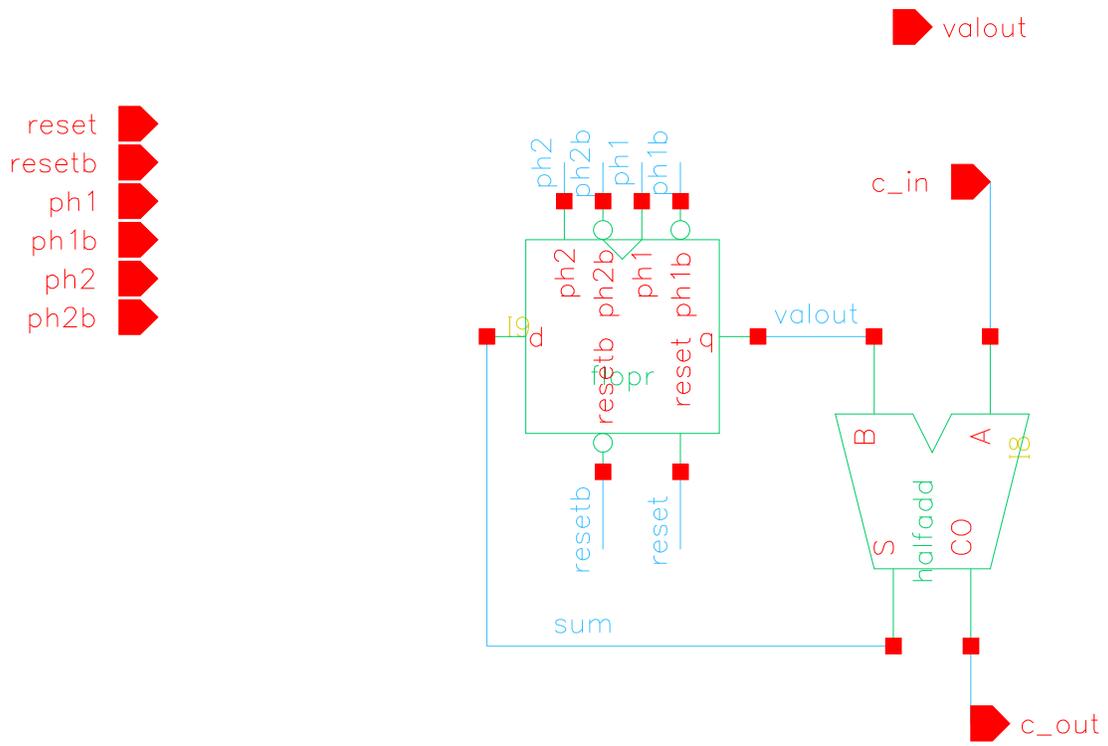
xor2



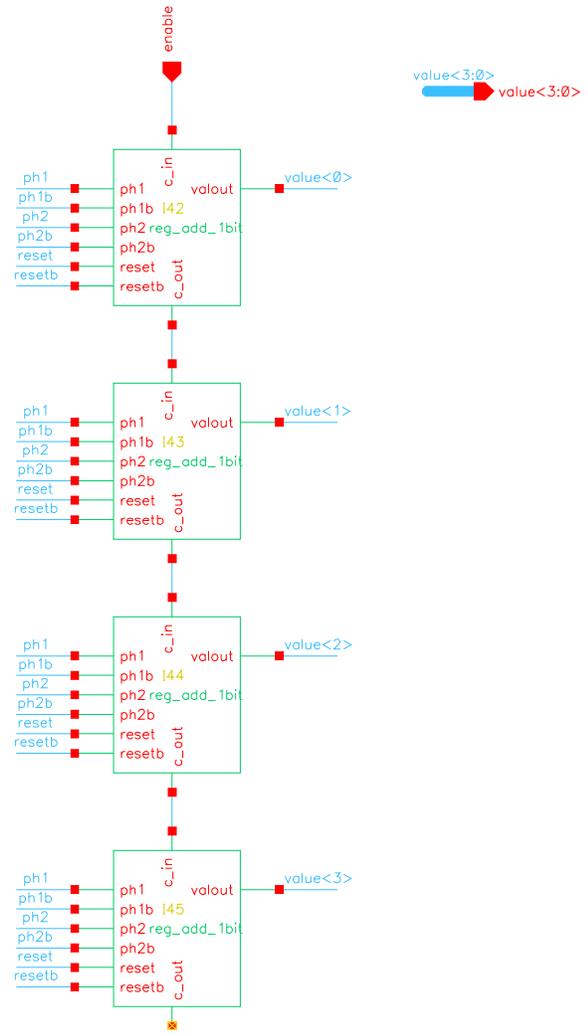
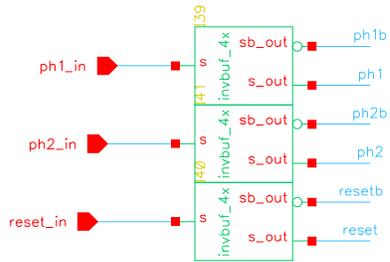
halfadd



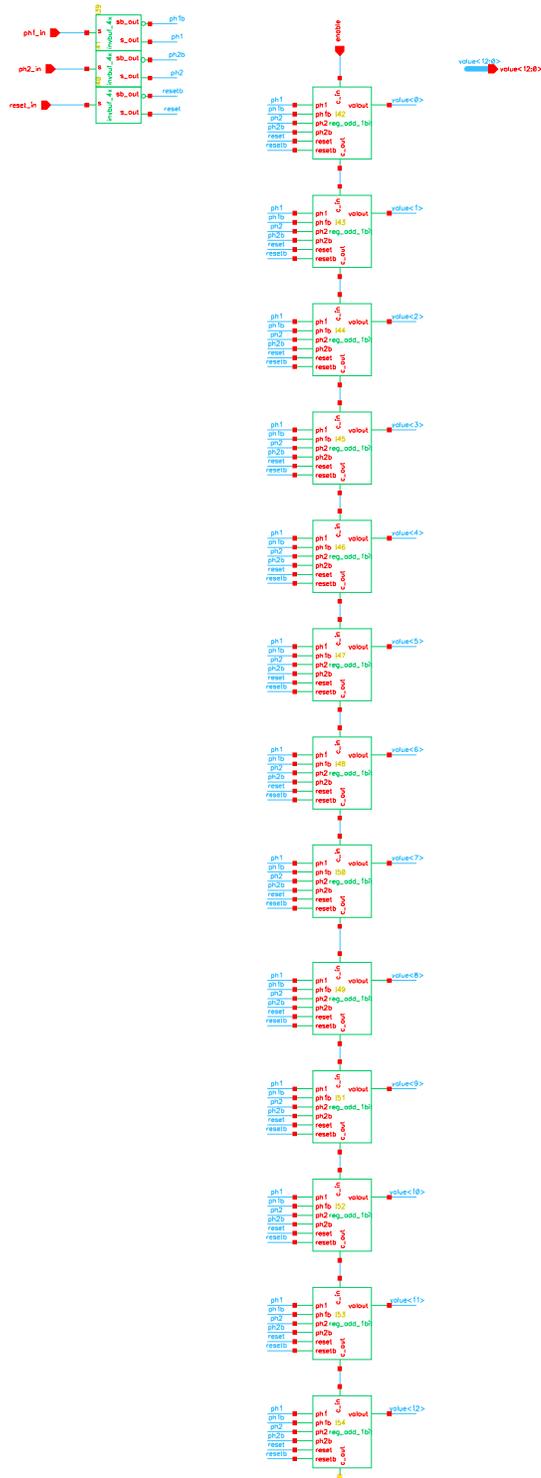
flopr



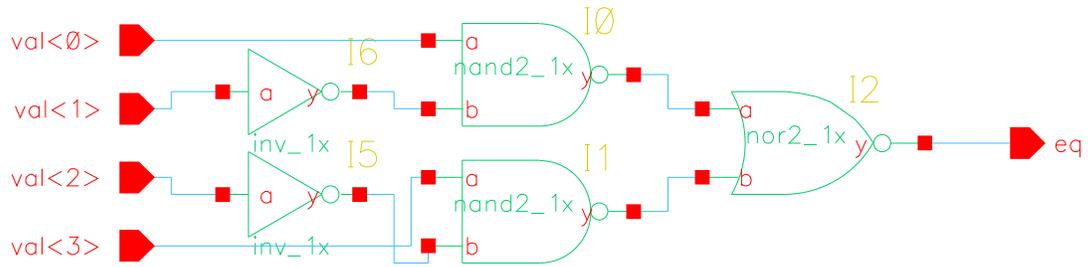
reg\_add\_1bit



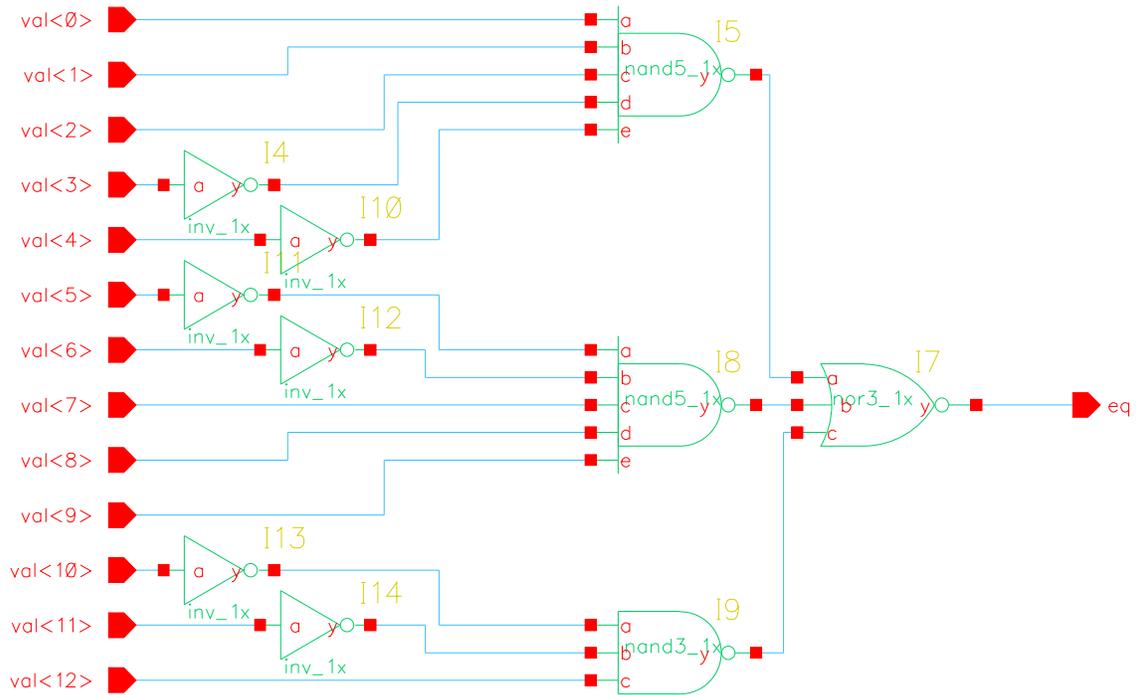
counter4



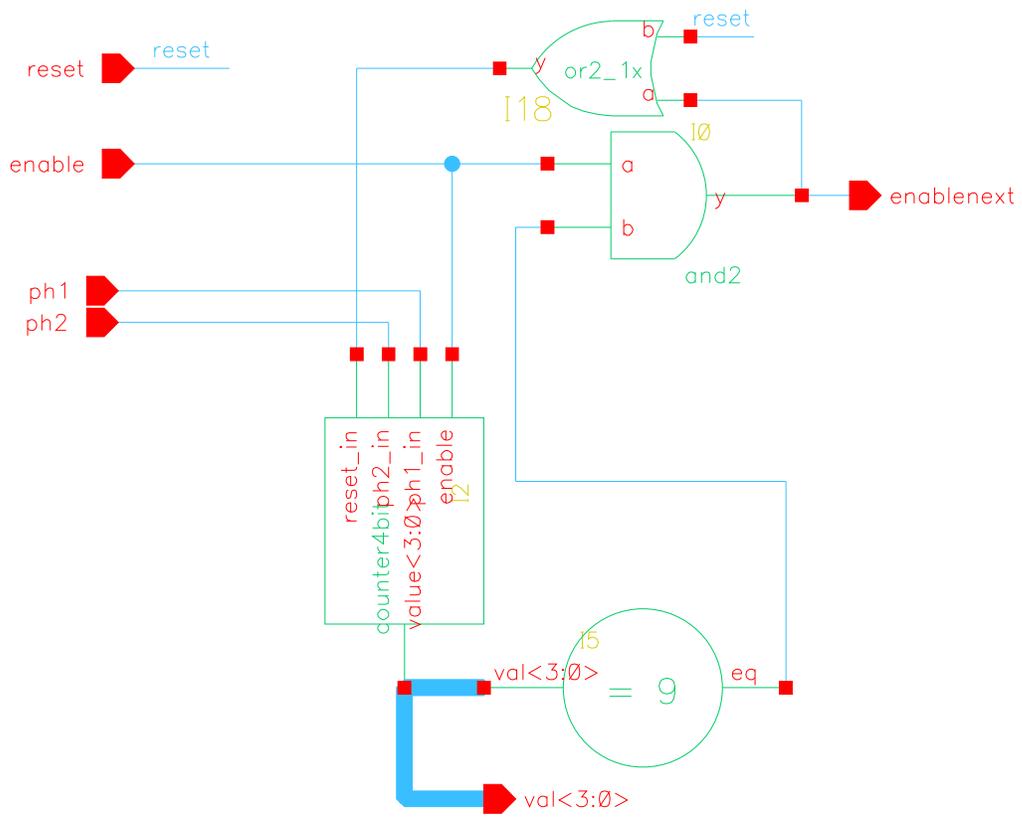
counter13



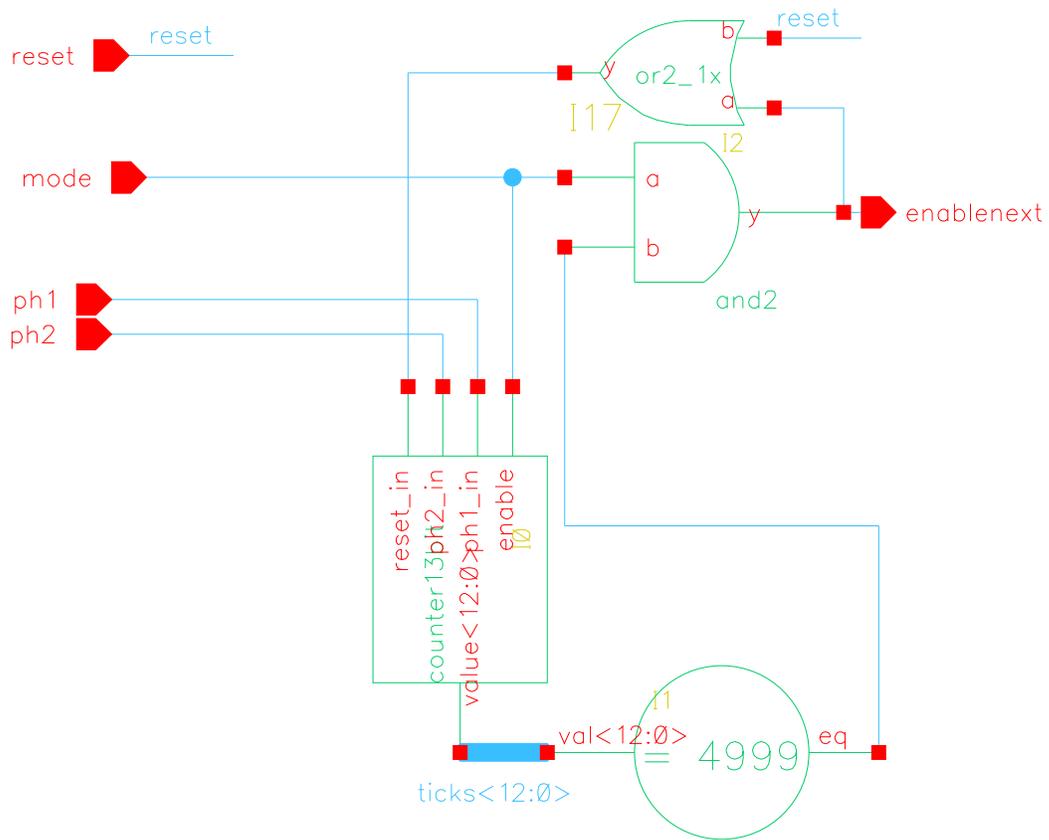
comparator4



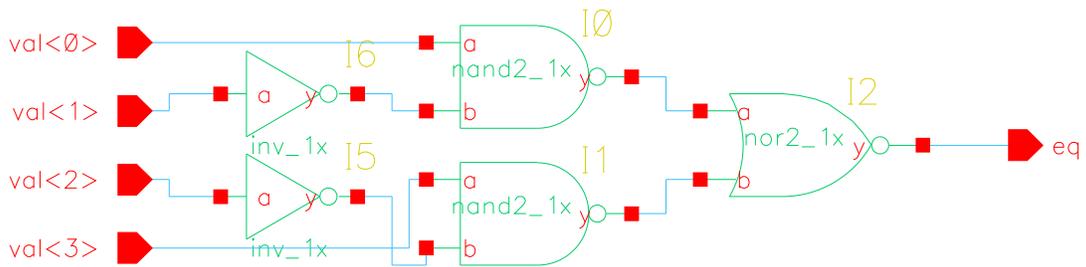
comparator13



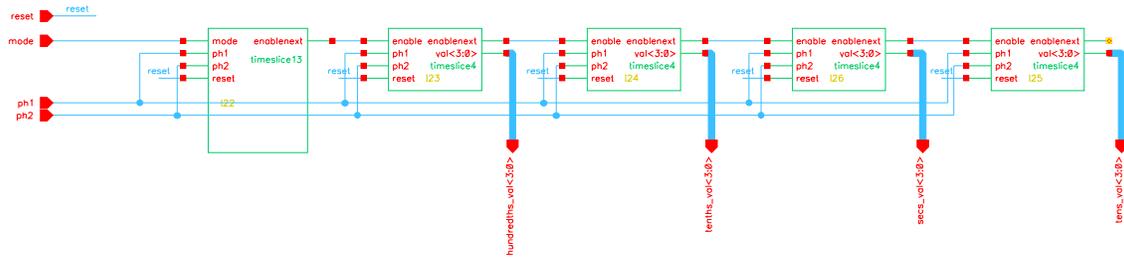
timeslice4



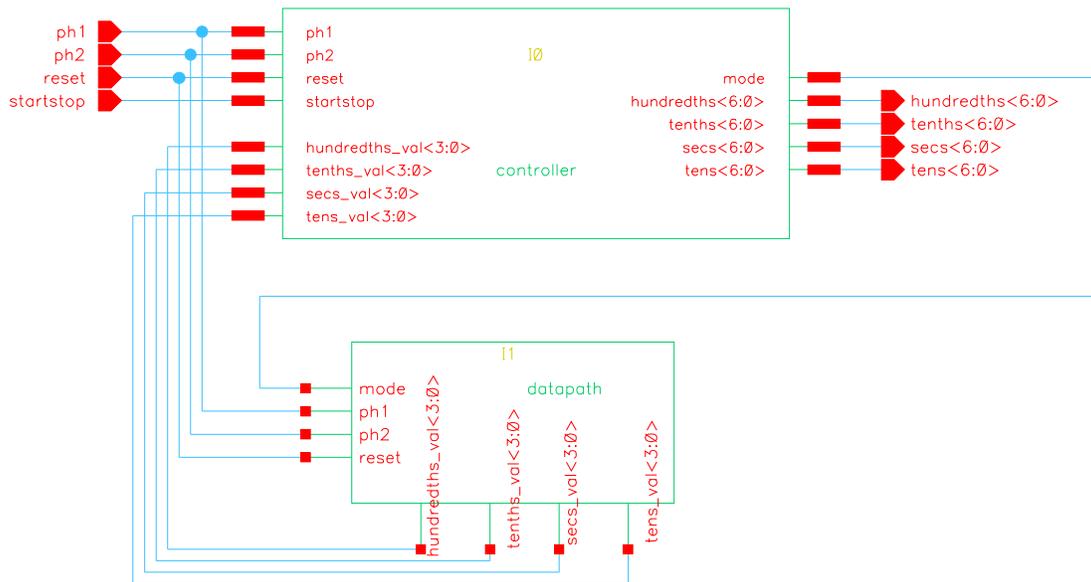
timeslice13



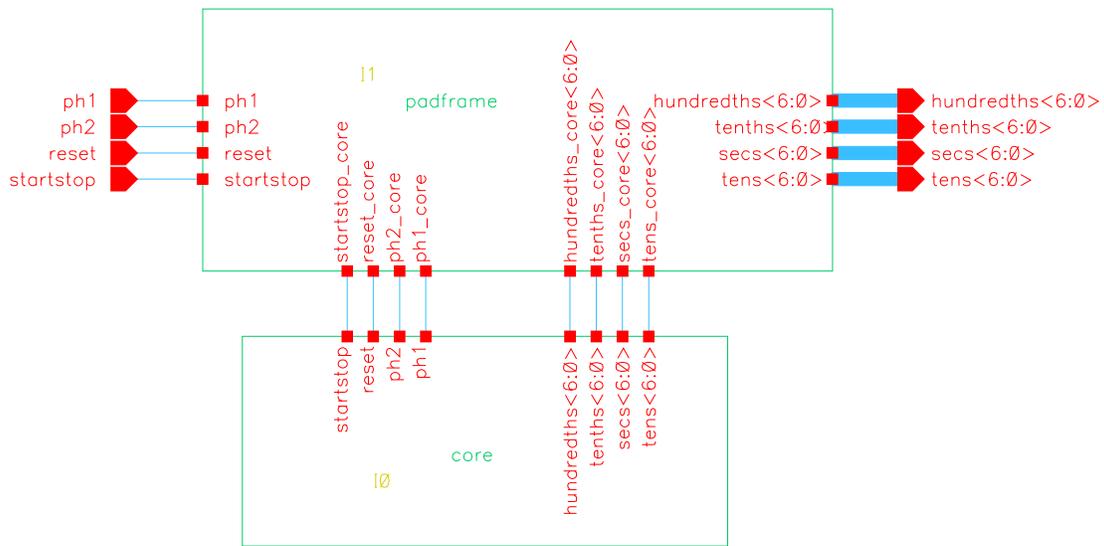
comparator4



datapath

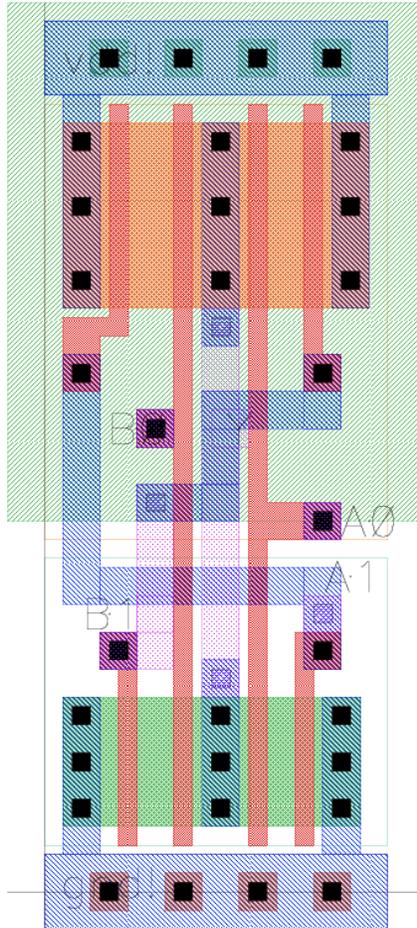


core

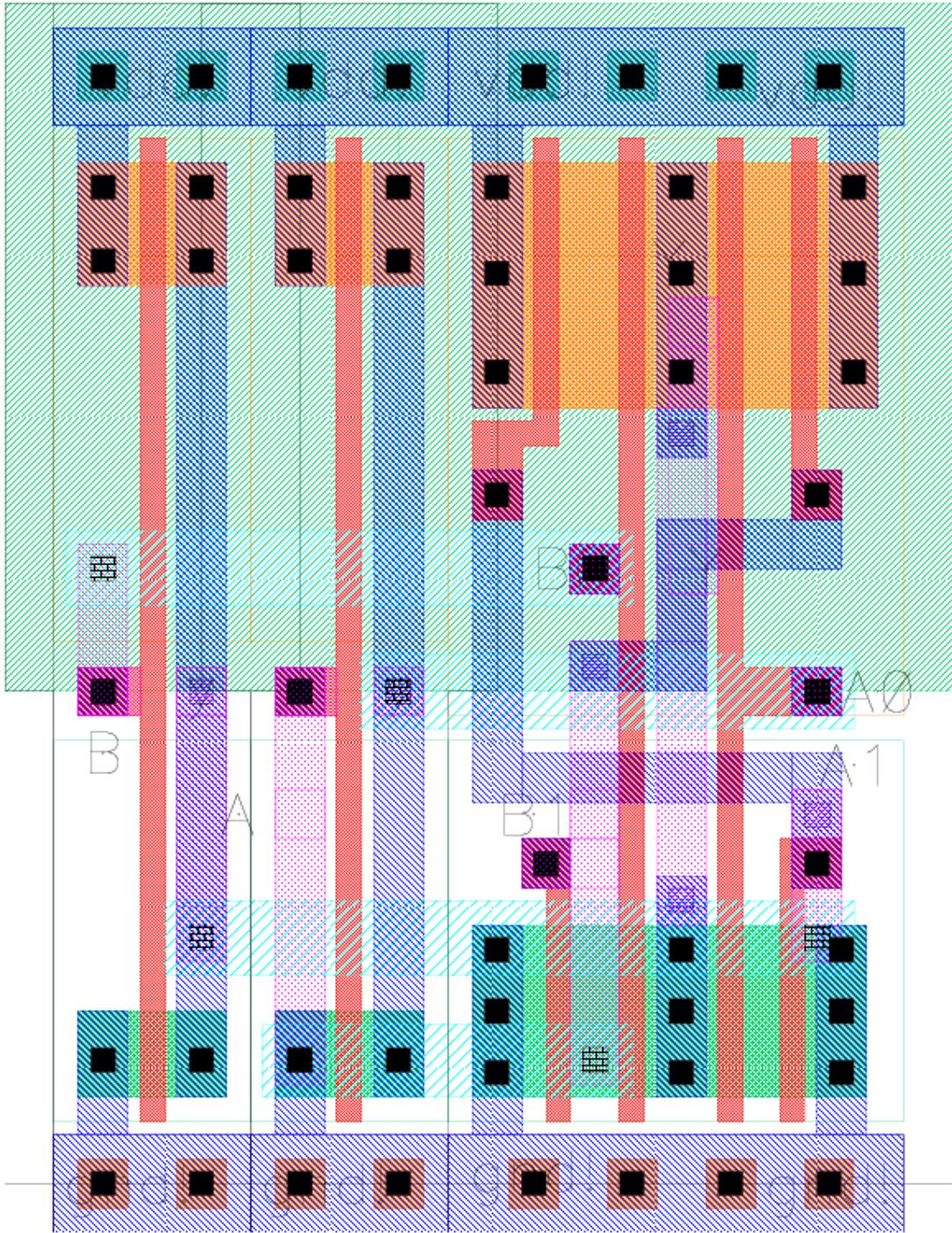


chip

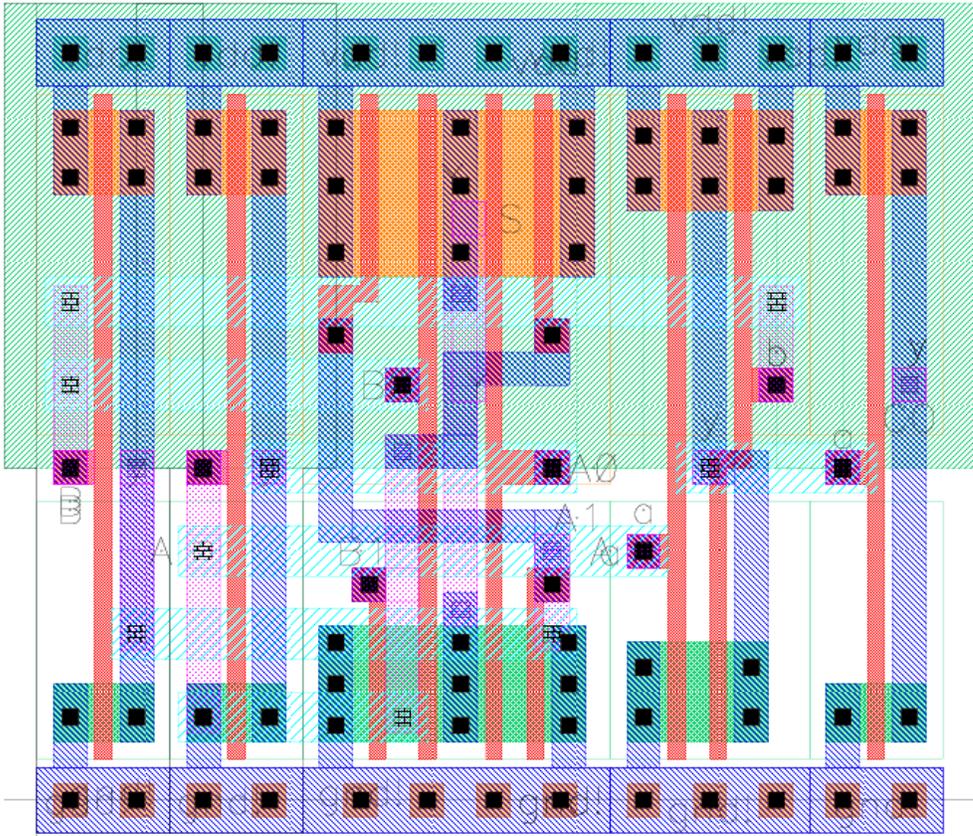
## Appendix C: Layouts



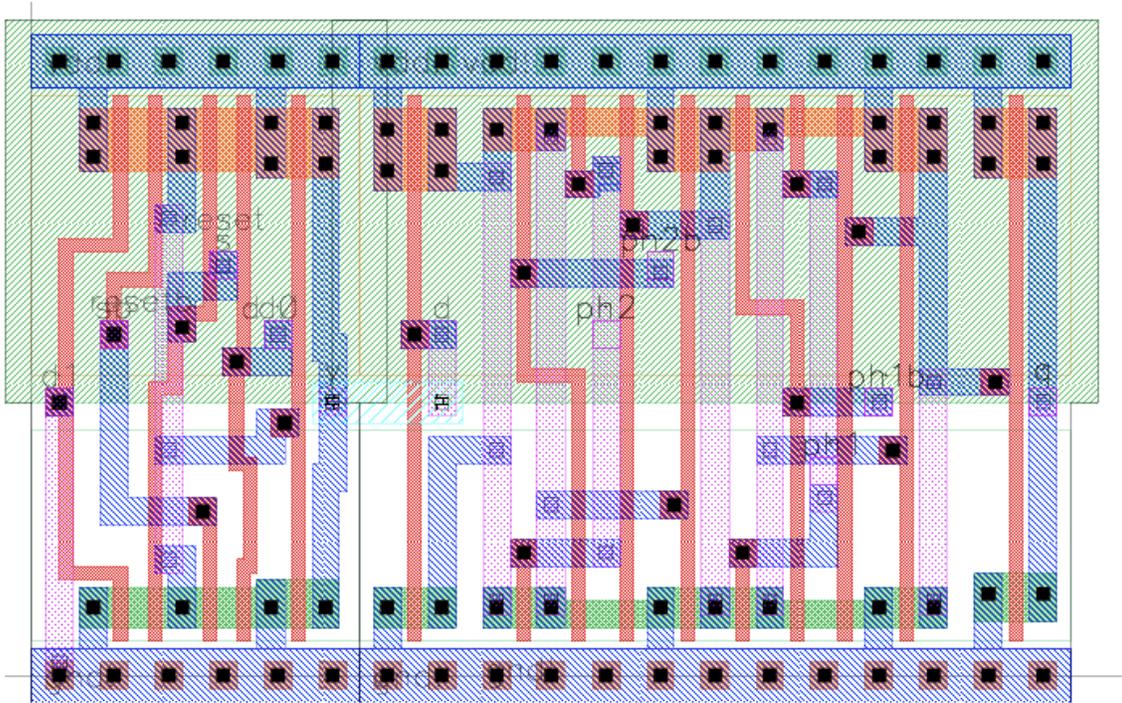
XOR\_CORE



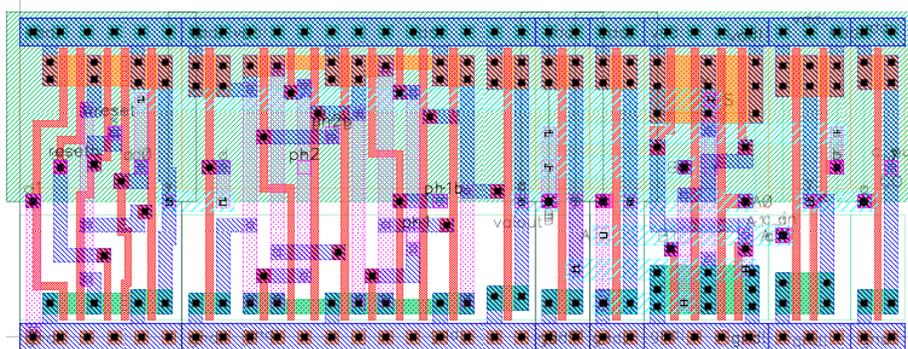
xor2



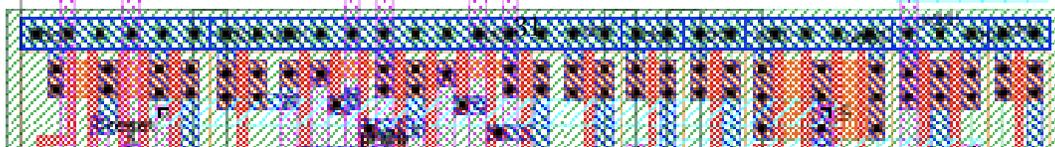
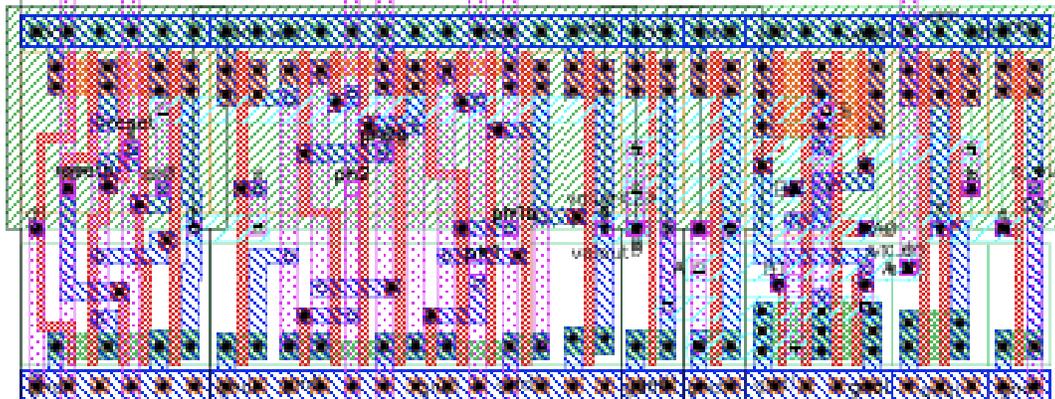
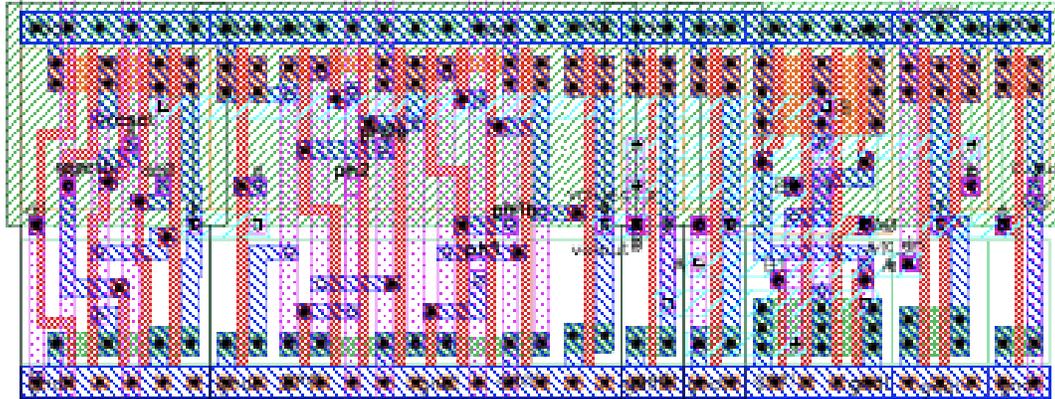
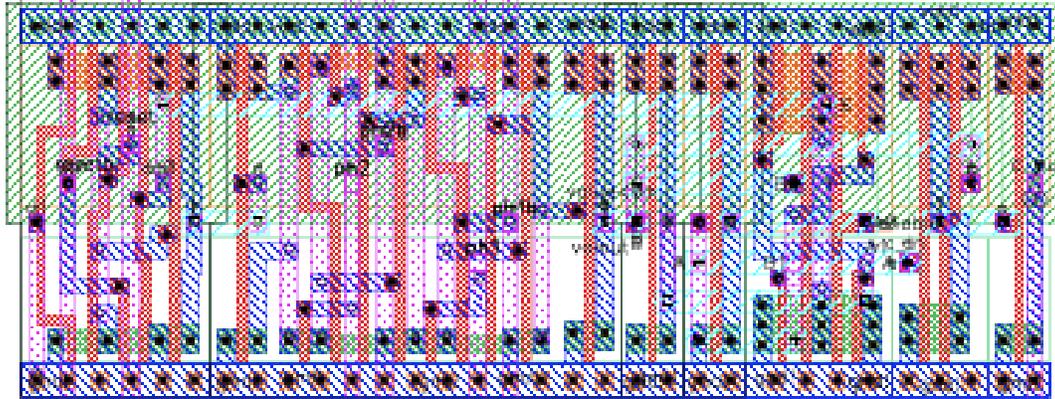
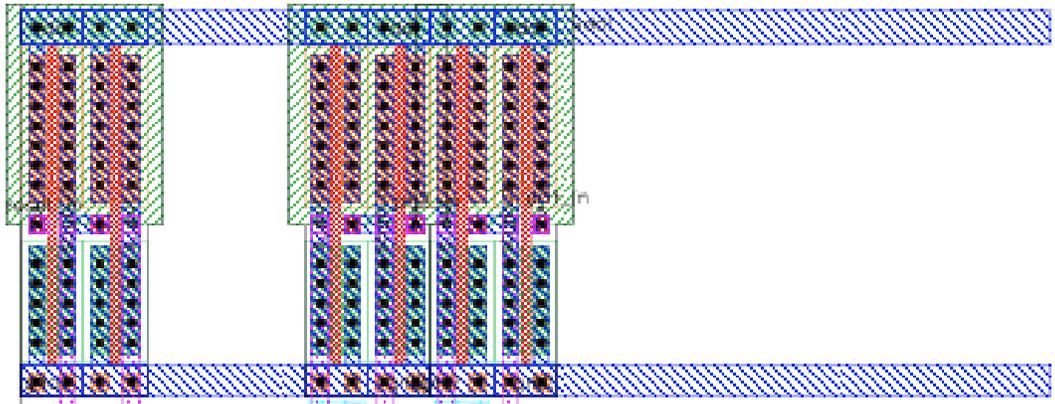
halfadd

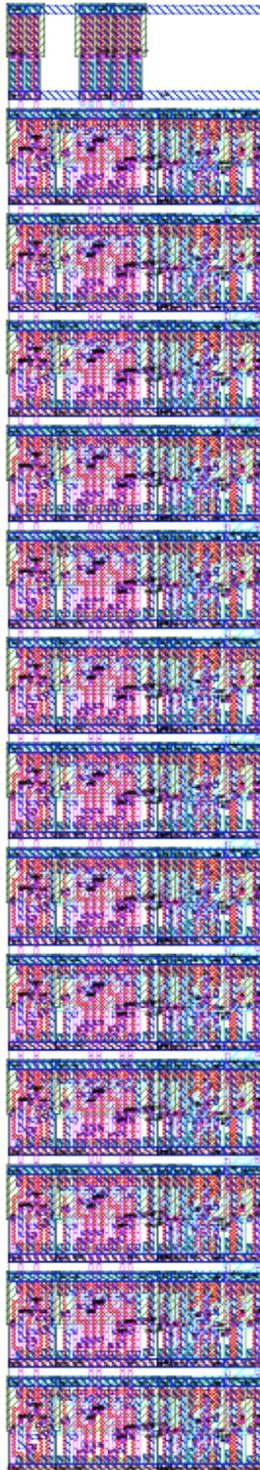


flopr

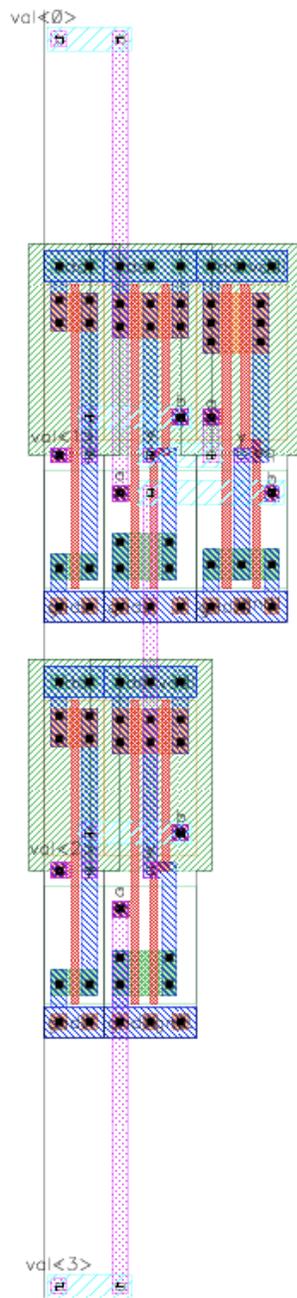


reg\_add\_1bit



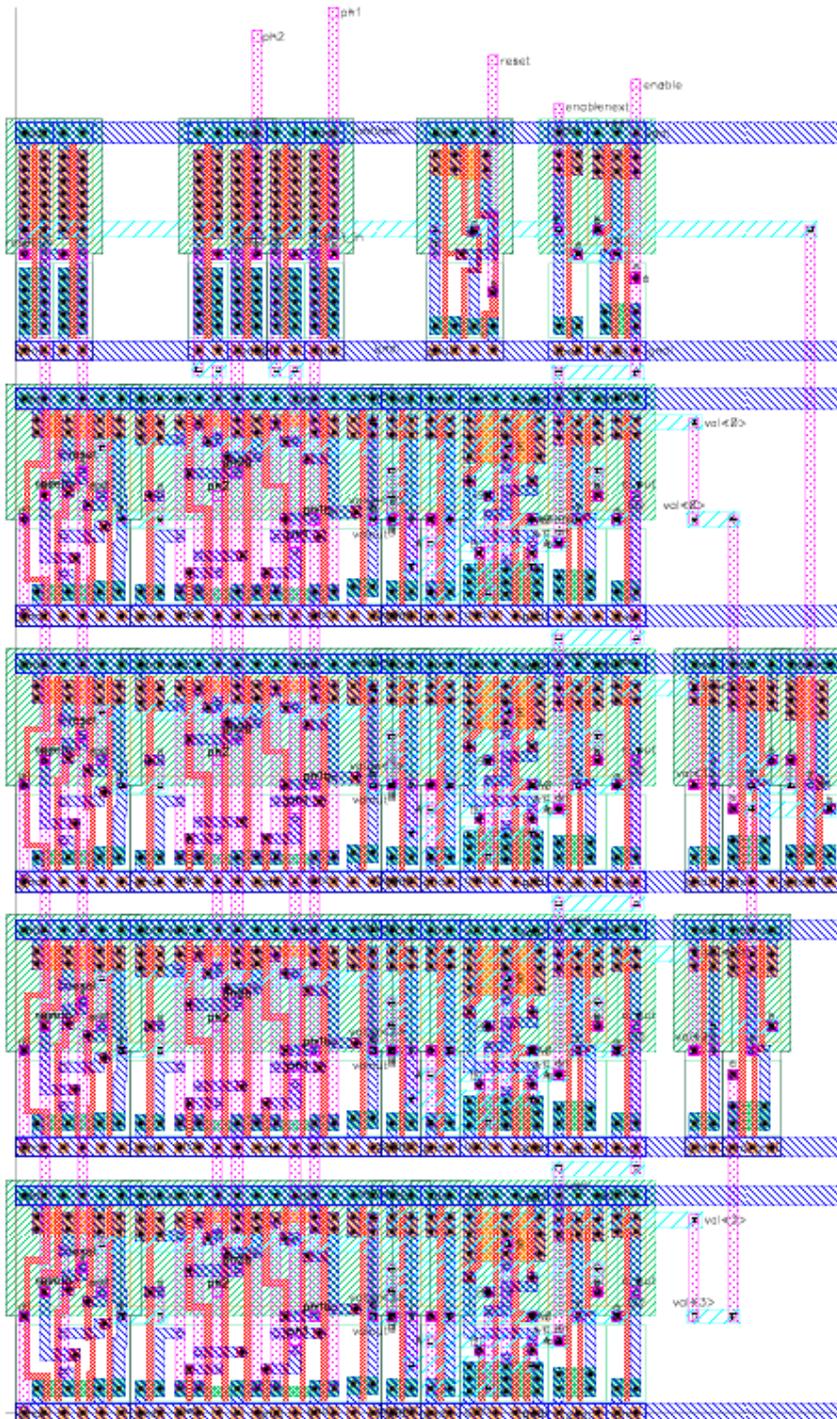


counter13

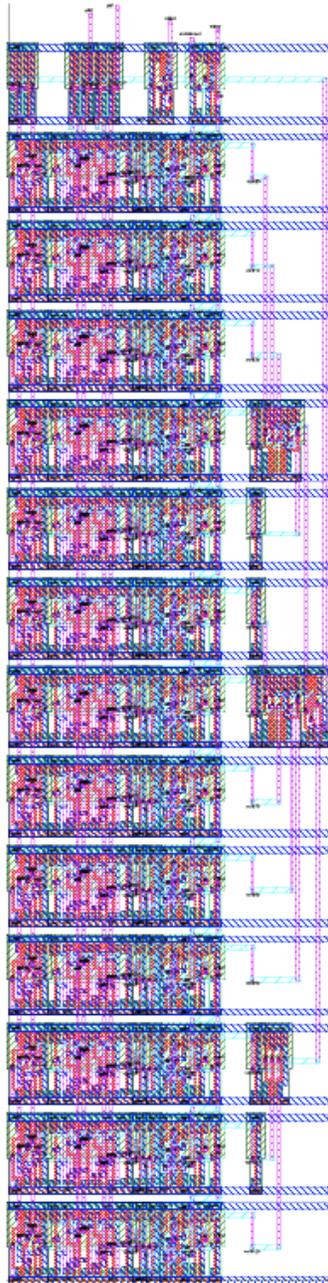


comparator4

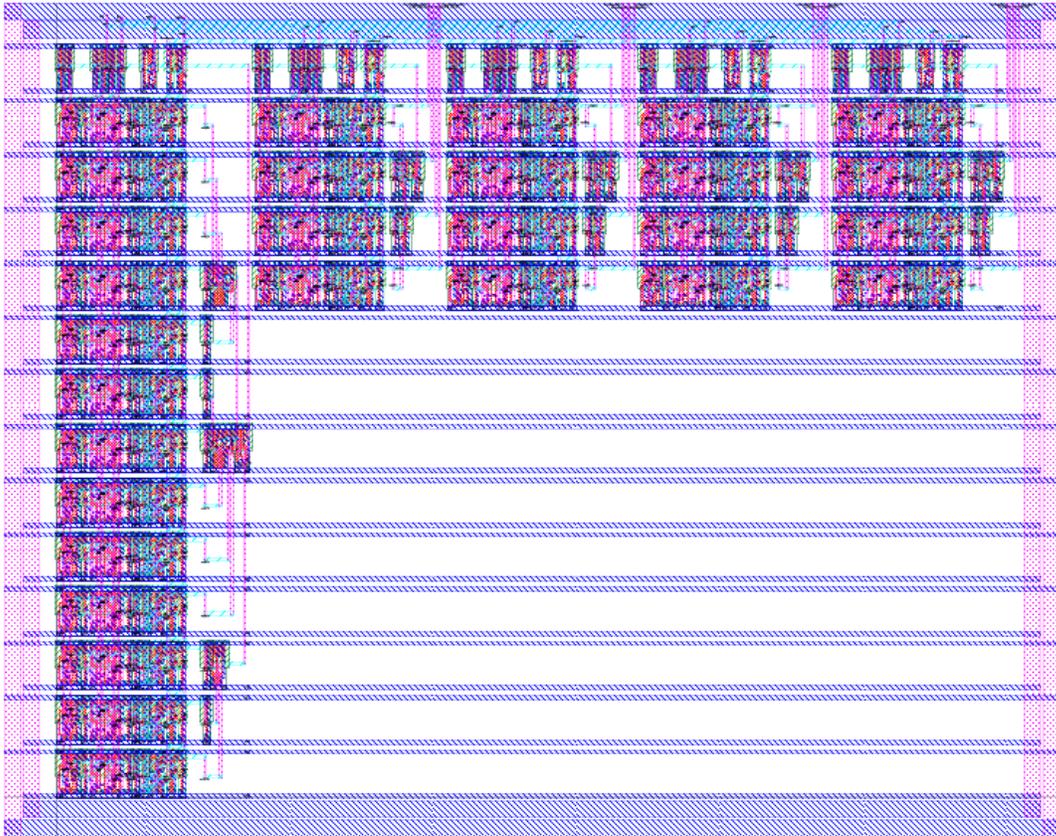




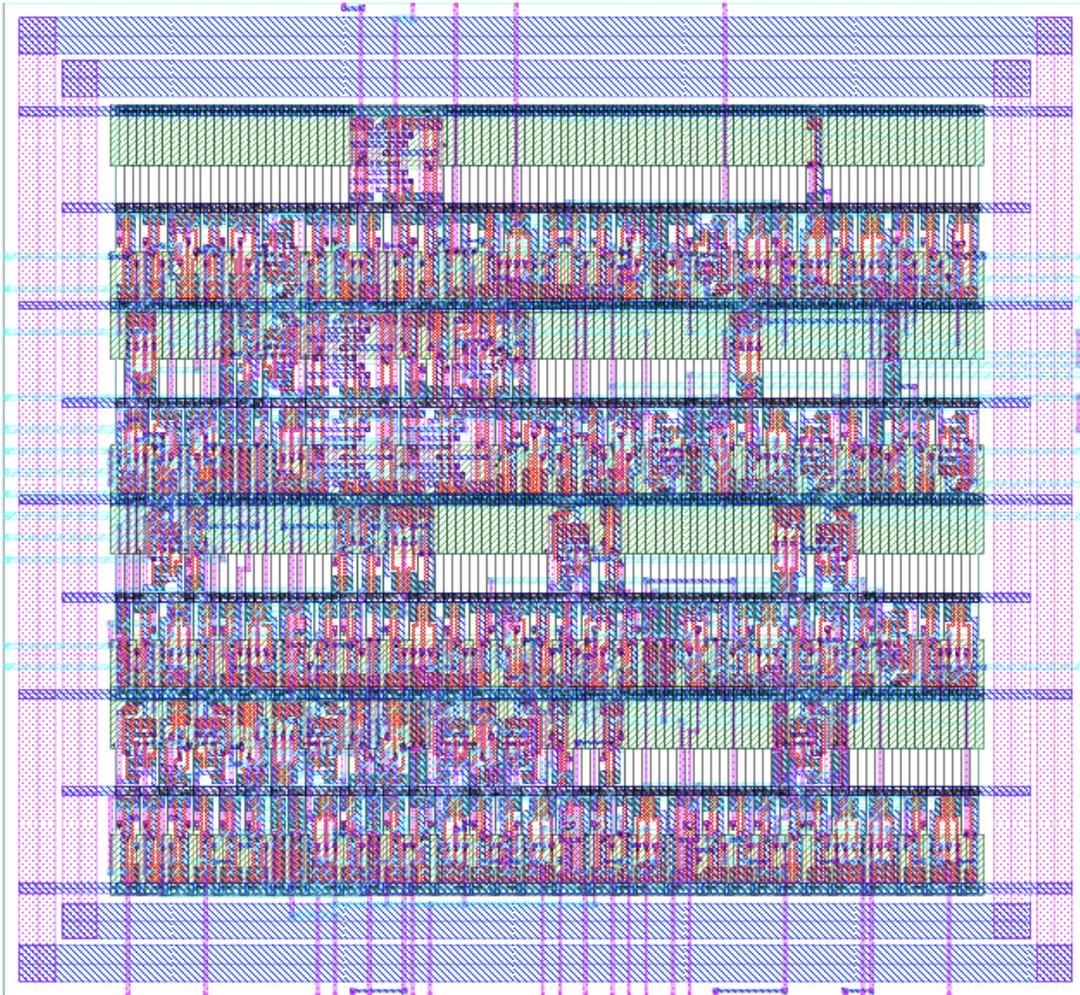
timeslice4



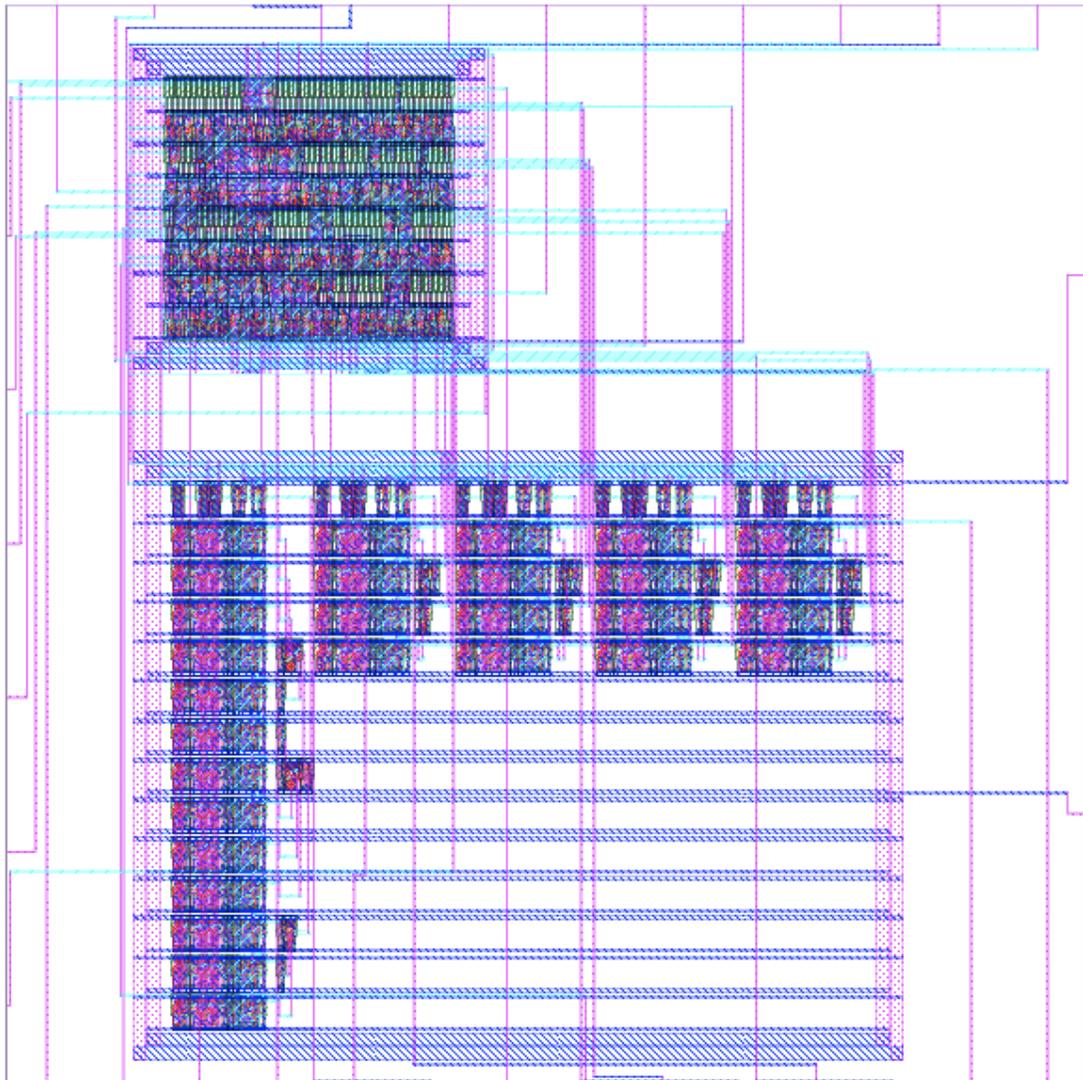
timeslice13



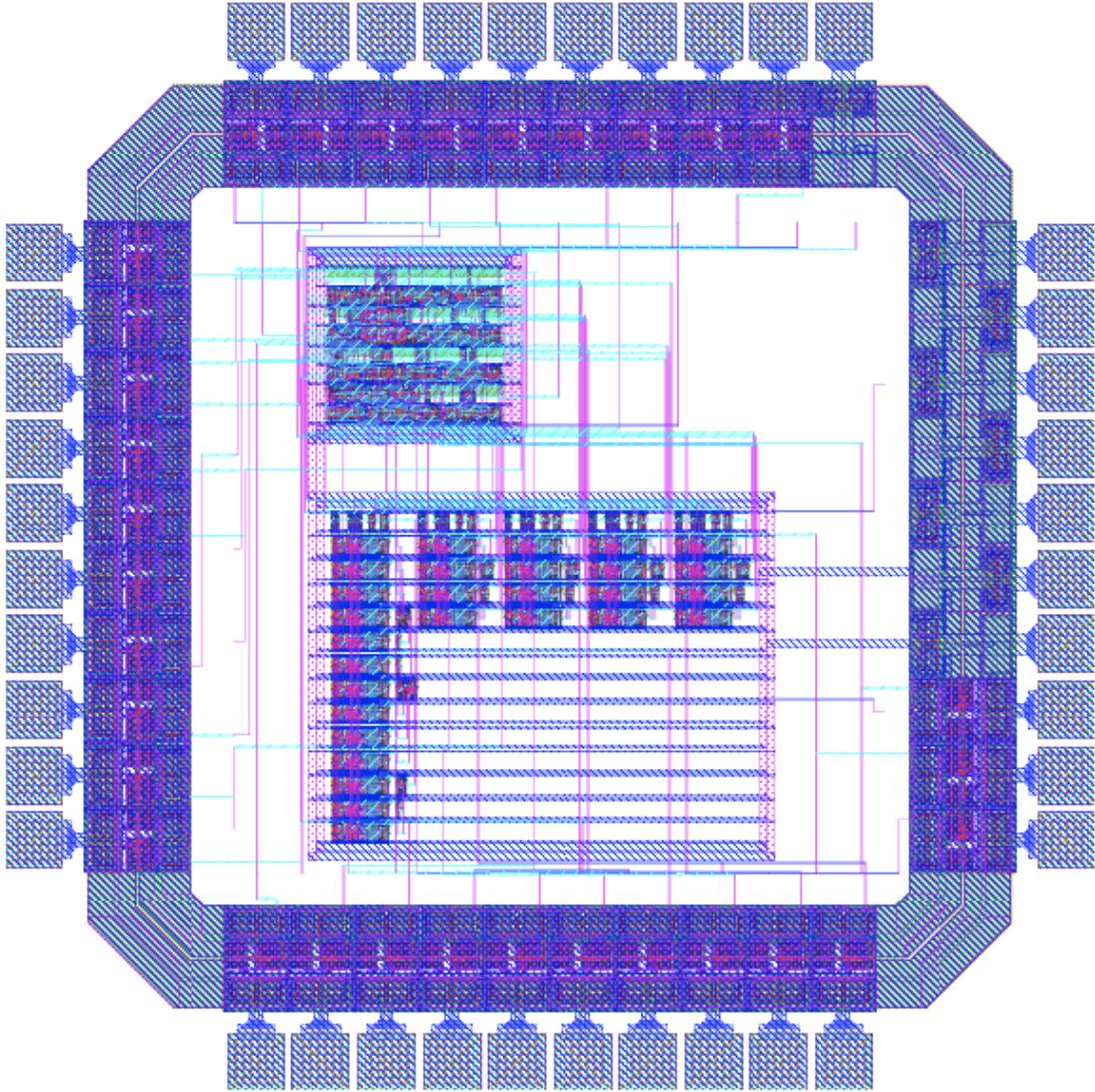
datapath



controller



core



chip