E158 Final Project: UART

Final Report

Dillon Ayers and Kramer Straube

19 April, 2010

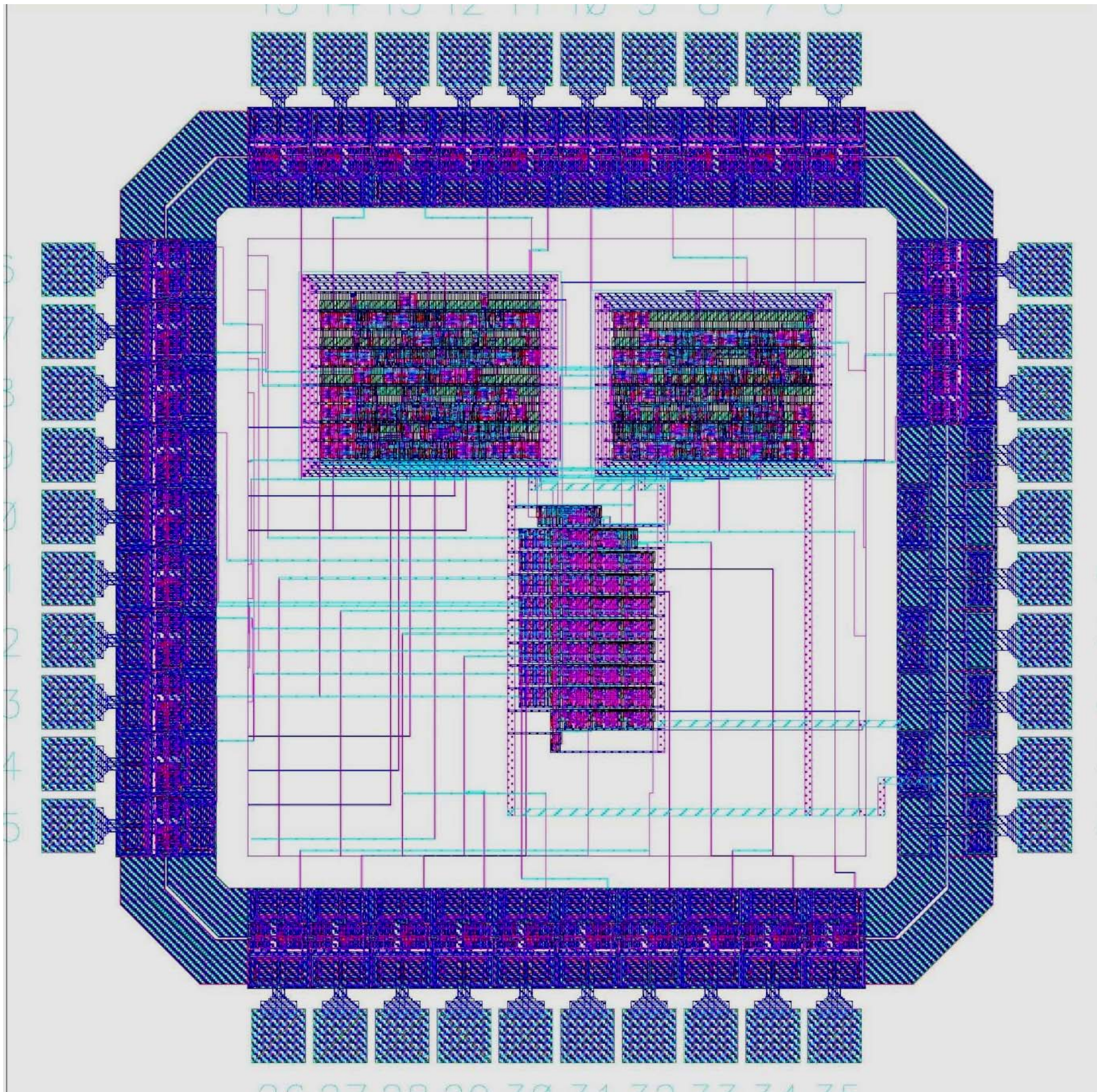

Figure 1: Complete chip layout

Introduction

We designed a universal asynchronous receiver-transmitter (UART) to be implemented in a 0.6 μm process and packaged in a 1.5 × 1.5 mm 40-pin MOSIS "TinyChip". UARTs convert data between parallel and serial forms, and are widely used by modern computer processors and microcontrollers to communicate with modems and other peripherals. Our UART design can independently receive and transmit serial data in 10-bit packets consisting of a start bit (always logic high), 8 data bits, and a stop bit (always logic low). However, transmission and reception both occur at a single user specified bit rate. Designating the rate of serial communication with a bit period, an 8-bit value that indicates the number of clock cycles required to receive or transmit a single bit, rather than the traditional baud rate, is both more intuitive for the user and easier to implement on the chip itself.

Functional Specifications

Commercial UARTs typically have a number of status bits and error flags to give the user information about the state of the device. Since our design is limited to 40 pins, 6 of which are reserved for power and ground, we focused on producing the essential functionality of a UART and allowing precise user control of the bit rate at the cost of limited status signals. Table 1 shows the inputs and outputs of our chip. The core consists of two synthesized logic blocks, *tx_module* and *rx_module*, that control the flow of data through the design, and two custom logic blocks, *tx_line* and *rx_line*, that perform the conversion between serial and parallel data formats.

Table 1: Summary of overall I/O

Pin/Bus Name	Direction	Description
data_tx<7:0>	input	Parallel data input
ph1, ph2	input(s)	Clock consisting of two non-overlapping out-of-phase signals
reset	input	Global reset
bit_period<7:0>	input	Determines the frequency of serial transmission and reception
RxD	input	Serial input data
txdata_set	input	Set for one clock cycle when parallel data is written to transmitter holding register. Clears internal *trdy* status bit.
rxdata_rdy	output	Set for one clock cycle when data is transferred from the receiver shift register to the receiver holding register.
rx_fe	output	Set when a framing error (illegal stop bit) is detected
TxD	output	Serial output data
data_rx<7:0>	output	Parallel output data

The following outline describes the functional relationship between these inputs and outputs.

Transmission:

1. The *txdata_rdy* input is pulsed when the parallel input, *data_tx*, represents meaningful data to be transmitted. This pulse enables the transmitter holding register *txdata* and clears the status bit *trdy*.
2. If status bit *tmt* is set, indicating that the previous transmission is complete, the contents of *txdata* are written to the transmitter shift register *txshift*, the transmitter BRG is reset, *tmt* is cleared, and *trdy* is set.

3. When the value of the 8-bit counter in the transmitter BRG reaches the value of the *bit_period* input, the *txshift* enable is pulsed for one clock cycle to move the next data bit on to pin *TxD* and the counter is reset.
4. The previous step repeats until an entire 10-bit data packet has been serially transmitted.
5. If new data was written to *txdata* during transmission, the above process repeats starting with step 2. Otherwise the transmitter is idle until *txdata_rdy* is pulsed again.

Reception:

1. When a start bit is detected at the serial input the value of the 8-bit counter in the receiver BRG is initialized to one half the input bit period.
2. When the value of the receiver BRG counter reaches that of *bit_period*, the *rxshift* enable is pulsed for one clock cycle to move the next serial data bit in to the receiver shift register and the counter is reset.
3. The previous step repeats until an entire 10-bit data packet has been received. If the most recently received bit is not a stop bit, *rx_fe* is set indicating a framing error.
4. The *rxdata_rdy* output is pulsed for one clock cycle to indicate that data is ready to be read out. This pulse also enables the receiver holding register, copying the 8 data bits in *rxshift* to *rxdata*.

Our design receives serial data properly if it changes with a bit period that differs from the expected (a number of clock cycles equal to *bit_period*) by 2.5% or less for values of *bit_period* greater than or equal to 40. For values of *bit_period* less than 40 but greater than or equal to the minimum bit period of 8 clock cycles, Serial data is correctly received if it changes with a period that matches the expected.

Floorplan and Pinout

Figure 2 shows a floorplan of the chip with approximate areas of top level logic blocks. The transmitter/receiver datapath block in Figure 2 consists of the *tx_line* and *rx_line* blocks placed side by side with the former on the left (see Appendix D). These blocks were laid out separately to facilitate simulation, but they share wells, power, and ground in the core layout.

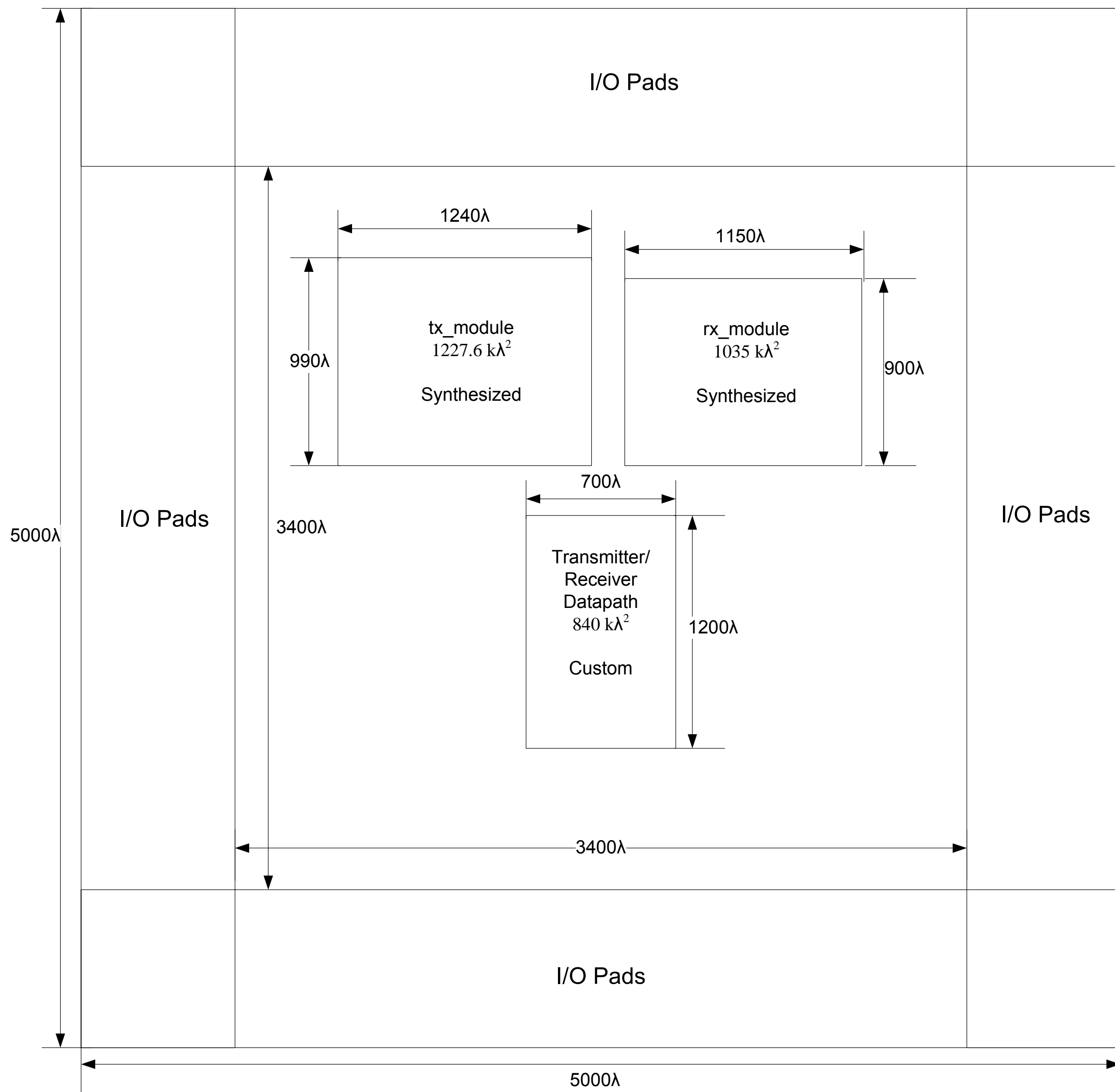

Figure 2: Final floorplan

The custom block matches the preliminary floorplan quite closely in structure and area. It is one bit-slice taller because the inverters used to buffer and produce complementary forms of the inputs were not considered in preliminary area estimation. The datapath is also slightly narrower than estimated because the proposed floorplan assumed a worst case in which all sequential wordslices used resettable, enabled flip-flops, making the overall area slightly smaller than the initial estimate.

The synthesized logic is separated based on reception vs. transmission, rather than on bit-rate generation vs. control of data flow as proposed, to facilitate simulation and exploit regularity between these two functions. The total area of the synthesized blocks is approximately 90% larger than initially estimated. This is largely due to the fact that the design evolved to include separate bit-rate generator modules for reception and transmission out of a need to initialize the counter measuring the receiver bit period to half the value of *bit_period*. Furthermore, the synthesized logic areas indicated in Figure 1 include power and ground rings and filler cells not accounted for in the preliminary area estimates. Even with nearly double the expected synthesized logic area, the final design fits comfortably within the constraints of the MOSIS TinyChip package.

Figure 3 shows the chip pinout with signal names and corresponding pin numbers. The obtuse triangles point toward the interior of the padframe for inputs and toward the exterior for outputs. The final design utilizes 33 pins for I/O; one less than the allowable maximum.

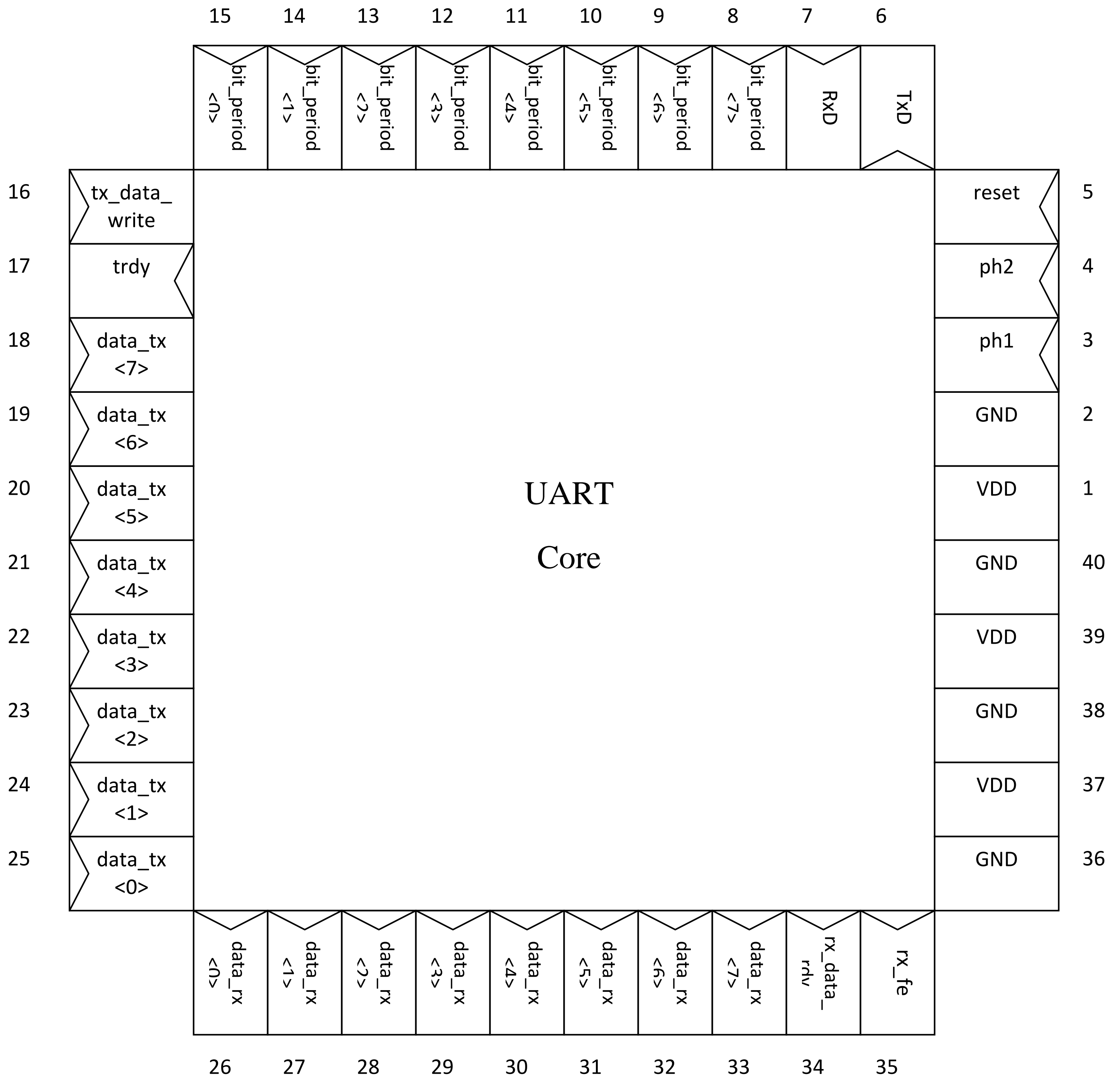

Figure 3: Pinout Diagram

Verification

We first implemented our design in Verilog (see Appendix B) and simulated this RTL description using NC Verilog by applying test vectors and checking outputs against expected vectors. Structural netlists of the core and chip schematics were simulated using the same test vectors. We used a python script to generate a comprehensive set of test vectors that tests both transmission and reception at such design corners as: minimum and maximum bit period, data bytes 0x00 and 0xFF, and reception of data changing at a rate that differs from that specified by *bit_period* by various percentages. Each block layout, as well as the core and chip layouts, was subjected to design rule check (DRC) and layout vs. schematic (LVS) verification tests. Finally, the chip layout was taped out using the Caltech Interchange Format (CIF), and the output file was subjected to DRC and LVS. Our design passed all of these checks, as Table 2 indicates.

Table 2: Verification Status

Design Component	Passes Simulation	Passes DRC	Passes LVS
Verilog RTL	Yes	N/A	N/A
Core Schematic	Yes	N/A	N/A
Chip Schematic	Yes	N/A	N/A
Custom flopr Layout	N/A	Yes	Yes
rx_module Layout	N/A	Yes	Yes
tx_module Layout	N/A	Yes	Yes
rx_line Layout	N/A	Yes	Yes
tx_line Layout	N/A	Yes	Yes
Core Layout	N/A	Yes	Yes
Padframe Layout	N/A	Yes	Yes
Chip Layout	N/A	Yes	Yes
CIF Output File	N/A	Yes	Yes

Design Time

Table 3 lists the time spent on each aspect of the design. More time was spent on simulation than expected due in part to incorrect simulation of a particular logic optimization made by Synopsys when generating synthesized logic blocks. Simulation design time also includes time spent debugging and optimizing the Python script used to generate test vectors. Resolving verification errors, on the other hand, took less time than expected. The estimate of total design time is conservative because it does not account for time spent on the initial research and "back of the envelope" design needed to specify the project proposal.

Table 3: Summary of Design Time

Project Component	Design Time (hours)					
	Schematic	Symbol	Layout	Simulation	DRC/LVS	Total
Verilog RTL	N/A	N/A	N/A	10	N/A	22*
Custom Cell: flopr	1	0.1	2	N/A	0.2	3.3
rx_module (synthesized)	2	0.1	0.4	N/A	0.1	2.6
tx_module (synthesized)	0.2	0.1	0.4	N/A	0.1	0.8
rx_line (custom)	2.1	0.2	3	N/A	1.4	6.7
tx_line (custom)	2.2	0.2	3.2	N/A	1.2	6.8
Core	1.2	0.3	2.5	8	0.5	13
Padframe	2.4	0.4	3	N/A	2.5	8.3
Chip	0.5	0.3	1.8	1	0.5	4.1
CIF Output	N/A	N/A	0.4	N/A	1	1.4
Total	11.6	1.7	16.7	19	7.5	**69**

* The extra 12 hours in this row represents the time spend writing and editing the project RTL

Post-Fabrication Test Plan

After receiving our fabricated chip, we will use a TestosterICs functional chip tester to apply test vectors to our design according to the following steps. In addition, the package should be touched periodically; exceptionally high temperature could indicate a short.

- Check VDD and GND using a multimeter: resistance between two VDD pins or two GND pins should be relatively low, while resistance between VDD and GND should be very large.
- Apply non-overlapping clock signals to *ph1* and *ph2* and assert *reset* for several cycles while leaving all other inputs low. This should cause *trdy* to be high regardless of other inputs.
- With all other inputs low, applying a single-cycle pulse to *txdata_write* should cause *trdy* to be low for two cycles while data is transferred to the transmitter shift register and then high again, at which point *TxD* should go high for a cycle.
- Load a distinctive value such as 0x55 on to *data_tx* and assert *txdata_write* for one cycle. This should cause *TxD* to alternate between high and low for 10 clock cycles.
- Adjust the *bit-period* input to several different values, asserting *txdata_write* each time, and ensure that *TxD* switches value after the specified number of clock cycles.
- With all other inputs low, asserting *RxD* for two clock cycles should cause *rxdata_rdy* to go high 10 clock cycles later.
- Applying 0x08 to *bit_period* and alternating the value of *RxD* every 8 clock cycles should cause *rxdata_rdy* to go high after 77 clock cycles, at which point *data_rx* should have the value 0x55.
- Continue applying vectors in this manner to test design corners. For large values of *bit_period*, it can take hundreds of clock cycles before meaningful output is observed, but the exact number of clock cycles one must wait is easily calculated.

An example of testing a design corner would be to set *bit_period* to 0x27 (decimal 39) but alternate the value of *RxD* every 40 clock cycles. Since the expected data rate differs from the actual by 2.5%, correct reception should occur: *rxdata_rdy* should go high after 380 cycles, at which point *rx_data* should have the value 0x55. Test vectors will have the following form used in pre-silicon simulation of RTL and schematics:

	Inputs					**Expected Ouputs**				
Signal	reset	RxD	txdata_write	bit_period	data_tx	TxD	trdy	rxdata_rdy	rx_fe	data_rx
Bits	1	1	1	8	8	1	1	1	1	8

Alternatively, test vectors could be hard coded in to a C routine used to program a microcontroller. Such a routine could replicate a serial input using one of the microcontroller's timers, poll for the *rxdata_rdy* signal, and display the received parallel data on a bank of LEDs.

Appendix A: File Locations

- Verilog code: \\chips.eng.hmc.edu\kstraube\home
- Test vectors: \\chips.eng.hmc.edu/kstraube\home\IC_CAD\cadence\chip_run1\golden.tv
- Synthesis results: \\chips.eng.hmc.edu\kstraube\home\IC_CAD\synth (results for several modules are in this directory)
- Cadence libraries: \\chips.eng.hmc.edu\kstraube\home\IC_CAD\cadence \uart
- CIF file: \\chips.eng.hmc.edu\kstraube\home\IC_CAD\cadence\chip.cif
- PDF of chip: \\chips.eng.hmc.edu\kstraube\home\chip.pdf
- PDF of Final Report: \\Charlie\HMCDFS\HMC_2011\dayers\desktop\VLSI\E158 Final Report.pdf

•

Appendix B: Verilog Code

```
module UART( input   ph1,
                     ph2,
                     reset,
                     RxD,
                     txdata_write,
             input   [7:0] bit_period,
             input   [7:0] data_tx,
             output TxD,
                     trdy,
                     rxdata_rdy,
                     rx_fe,
             output [7:0] data_rx);

        //transmitter
        wire txshift_enable, tx_rdy;
        tx_module transmitter_synth(ph1, ph2, reset, txdata_write, bit_period, trdy, tx_rdy,
                                    txshift_enable);
        tx_line tx_datapath(ph1, ph2, reset, txshift_enable, data_tx, tx_rdy, TxD);

        //receiver
        wire stopbit, rxshift_enable;
        rx_module receiver_synth(ph1,ph2,reset,bit_period,stopbit,RxD,rxshift_enable,
                                 rxdata_rdy,rx_fe);
        rx_line rx_datapath(ph1,ph2,reset,RxD,rxshift_enable,rxdata_rdy,stopbit,data_rx);


endmodule

//receiver Verilog
module rx_module(   input   ph1, ph2,
                    input   reset,
                    input   [7:0] bit_period,
                    input   stopbit,
                    input   RxD,
                    output  rxshift_enable,
                    output rxdata_rdy,
                    output  rx_fe);

        //number of bits in a packet
```

```
	parameter packetBits = 4'b1010;

	//internal signals
	wire BRG_set, rx_busy, shiftbits, rmt;

	//start bit detection
	assign BRG_set = RxD & (~rx_busy);

	//status bit indicating state of shift register
	statusBit rx_busy_status(ph1,ph2,reset,rxdata_rdy,BRG_set,rmt);
	assign rx_busy = ~rmt;

	//generates enable pulses at bit rate
	rx_BRG rxshift_enable_gen(ph1,ph2,reset,bit_period,BRG_set,shiftbits);

	assign rxshift_enable = shiftbits & rx_busy;

	//counts the number of bits we've received
	bit_counter
count_BitsReceived(ph1,ph2,(reset|BRG_set),rxshift_enable,packetBits,rxdata_rdy);

	assign rx_fe = rxdata_rdy & stopbit;

endmodule

//transmitter control top-level
module tx_module(	input	ph1, ph2,
			input reset,
			input txdata_write,
			input [7:0] bit_period,
			output trdy,
			output tx_rdy,
			output txshift_enable);

	parameter packetBits = 4'b1010;

	wire txdata_read, tx_data, tx_done, tmt, tx_busy,txshiftbits;

	//status of holding register
	statusBit trdy_status(ph1,ph2,reset,txdata_read,txdata_write,trdy);
	assign tx_data = ~trdy;

	//status of shift register
```

```
        statusBit tmt_status(ph1,ph2,reset,tx_done,tx_rdy,tmt);
        assign tx_busy = ~tmt;

        //FSM controller
        transmit_FSM tx_control(ph1,ph2,reset,tx_data,tx_busy,tx_rdy,txdata_read);

        //counts sent bits
        bit_counter count_BitsSent(ph1,ph2,(reset|txdata_read),txshift_enable,packetBits,tx_done);

        //generates enable pulses at the bit rate
        tx_BRG txshift_enable_gen(ph1,ph2,reset,txdata_read,bit_period-1,txshiftbits);

        assign txshift_enable = txshiftbits & tx_busy;

endmodule

//self-reseting counter that keeps track of bits sent/received
module bit_counter(   input ph1,
                      input ph2,
                      input reset,
                      input enable,
                      input [3:0] limit,
                      output done);

        wire [3:0] current_value;
        wire [3:0] next_value;
        wire r;

        assign r = reset | done;

        flopenr #(4) bitCount_reg(ph1,ph2,r,enable,next_value,current_value);

        assign next_value = current_value + 4'b0001;
        assign done = (current_value == limit);

endmodule

//setable-resetable flop
module statusBit(     input   ph1,ph2,
                      input   reset,
                      input   s,
                      input   r,
                      output next_value);
```

```
	wire next_value, current_value,set;
	assign next_value = current_value;
	assign set = reset | s;
	flopsr #(1) status_reg(ph1,ph2,set,r,next_value,current_value);

endmodule

//generates enable pulses at bit rate
module rx_BRG(	input	ph1,
		input	ph2,
		input	reset,
		input	[7:0] bit_period,
		input	BRG_set,
		output equal);

	wire [7:0] current_value;
	wire [7:0] next_value;
	wire [7:0] half_bit_period;
	wire r;

	assign half_bit_period = {1'b0, bit_period[7:1]};
	rx_module
receiver_synth(ph1,ph2,reset,bit_period,stopbit,RxD,rxshift_enable,rxdata_rdy,rx_fe);
	assign r = reset | equal;

	flopr #(8) bitCount_reg(ph1,ph2,r,next_value,current_value);

	assign next_value = (BRG_set)? half_bit_period : current_value + 8'b00000001;
	assign equal = (current_value == bit_period-1);

endmodule

//FSM controller
module transmit_FSM(	input	ph1, ph2,
			input	reset,
			input	tx_data,
			input	tx_busy,
			output tx_rdy,
			output txdata_r);

	//internal signals
	wire [3:0] state;
```

```
	reg [3:0] nexstate;

	//state variables
	parameter S0 = 4'b0001;
	parameter S1 = 4'b0010;
	parameter S2 = 4'b0100;
	parameter S3 = 4'b1000;

	//state register
	flopr #(4)statereg(ph1,ph2,reset,nexstate,state);

	//nextstate logic
	always @ ( * )
		case(state)
			S0:	if ((tx_data) & (~tx_busy))	nexstate = S1;
					else				nexstate = S0;
			S1:	nexstate = S2;
			S2:	nexstate = S3;
			S3:	if (tx_busy)	nexstate = S3;
					else if ((~tx_busy) & (tx_data))  nexstate = S1;
					else if ((~tx_busy) & (~tx_data)) nexstate = S0;
			default nexstate = S0;
		endcase

	//output logic
	assign txdata_r = (state == S1);
	assign tx_rdy   = (state == S1) | (state == S2);

endmodule

//generates enable pulses at bit rate
module tx_BRG(		input ph1,
		input ph2,
		input reset,
	  input tx_rdy,
		input [7:0] bit_period,
		output shiftbits);

	wire [7:0] current_value;
	wire [7:0] next_value;
	wire equal;
	wire r;
```

```
	assign equal = (current_value == bit_period);
	assign r = reset | equal;

	flopr #(8) bitCount_reg(ph1,ph2,(r|tx_rdy),next_value,current_value);

	assign next_value = current_value + 8'b00000001;
	assign equal = (current_value == bit_period);
	assign shiftbits = (current_value == 8'b0) | tx_rdy;

endmodule

//transmission datapath
module tx_line(		input	ph1, ph2,
			input	reset,
			input	enableSignal,
			input	[7:0] data,
			input	tx_rdy,
			output TxD);

	//internal signals
	wire	bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7,txdmuxed, txdprecatch;
	wire	bit0premux,bit1premux,bit2premux,bit3premux,
			bit4premux,bit5premux,bit6premux,bit7premux,txdpremux;
	wire  [7:0] data_tx;

	//transmitter holding register
	flopr #(8) txdata(ph1,ph2,reset,data,data_tx);

	//transmitter shift register
	flopenr #(1) txflop7(ph1, ph2,reset,enableSignal,1'b0, bit7premux);
	flopenr #(1) txflop6(ph1, ph2,reset,enableSignal,bit7, bit6premux);
	flopenr #(1) txflop5(ph1, ph2,reset,enableSignal,bit6, bit5premux);
	flopenr #(1) txflop4(ph1, ph2,reset,enableSignal,bit5, bit4premux);
	flopenr #(1) txflop3(ph1, ph2,reset,enableSignal,bit4, bit3premux);
	flopenr #(1) txflop2(ph1, ph2,reset,enableSignal,bit3, bit2premux);
	flopenr #(1) txflop1(ph1, ph2,reset,enableSignal,bit2, bit1premux);
	flopenr #(1) txflop0(ph1, ph2,reset,enableSignal,bit1, bit0premux);
	flopenr #(1) txflopstart(ph1, ph2,reset,enableSignal,bit0, txdpremux);
	flopenr #(1) txfloptxpin(ph1, ph2,reset,enableSignal,txdmuxed, txdprecatch);

	//muxs to select between serial and parallel data
	mux2 #(1) bit7mux(bit7premux, data_tx[7], tx_rdy, bit7);
	mux2 #(1) bit6mux(bit6premux, data_tx[6], tx_rdy, bit6);
```

```
	mux2 #(1) bit5mux(bit5premux, data_tx[5], tx_rdy, bit5);
	mux2 #(1) bit4mux(bit4premux, data_tx[4], tx_rdy, bit4);
	mux2 #(1) bit3mux(bit3premux, data_tx[3], tx_rdy, bit3);
	mux2 #(1) bit2mux(bit2premux, data_tx[2], tx_rdy, bit2);
	mux2 #(1) bit1mux(bit1premux, data_tx[1], tx_rdy, bit1);
	mux2 #(1) bit0mux(bit0premux, data_tx[0], tx_rdy, bit0);
	mux2 #(1) startbitmux(txdpremux, 1'b1, tx_rdy, txdmuxed);
	mux2 #(1) pinmux( txdprecatch,1'b0, tx_rdy, TxD);

endmodule

//receiver shift register
module rx_line( input ph1, ph2,
		input   reset,
		input   RxD,
		input   en,
		input   rx_rdy,
		output stopbit,
		output [7:0] rxdata);

	wire bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7,startbit,stopbit;

	flopenr  #(1) flop0(ph1, ph2, reset, en, bit0, startbit);
	flopenr  #(1) flop1(ph1, ph2, reset, en, bit1, bit0);
	flopenr  #(1) flop2(ph1, ph2, reset, en, bit2, bit1);
	flopenr  #(1) flop3(ph1, ph2, reset, en, bit3, bit2);
	flopenr  #(1) flop4(ph1, ph2, reset, en, bit4, bit3);
	flopenr  #(1) flop5(ph1, ph2, reset, en, bit5, bit4);
	flopenr  #(1) flop6(ph1, ph2, reset, en, bit6, bit5);
	flopenr  #(1) flop7(ph1, ph2, reset, en, bit7, bit6);
	flopenr  #(1) flop8(ph1, ph2, reset, en, stopbit, bit7);
	flopenr  #(1) flop9(ph1, ph2, reset, en, RxD, stopbit);

	wire [7:0] data = {bit7,bit6,bit5,bit4,bit3,bit2,bit1,bit0};

	flopenr #(8) rxbuffer(ph1, ph2,reset, rx_rdy, data, rxdata);

endmodule

//submodules borrowed from mips.sv
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

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

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

  assign resetval = 0;

  mux2 #(WIDTH) rmux(d, resetval, reset, d2);
  flop #(WIDTH) f(ph1, ph2, d2, 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 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 flopsr #(parameter WIDTH = 8)
```

```
           (input  logic            ph1, ph2, set, reset,
            input  logic [WIDTH-1:0] d,
            output logic [WIDTH-1:0] q);

  logic [WIDTH-1:0] d2, resetval, setval, srval;

  assign resetval = 0;
  assign setval   = 1;
  assign srval    = 1;

  mux4 #(WIDTH) srmux(d, resetval, setval, srval, {set, reset}, d2);
  flop #(WIDTH) f(ph1, ph2, d2, q);
endmodule

module latch_158 #(parameter WIDTH = 8)
          (input  logic            ph,
           input  logic [WIDTH-1:0] d,
           output logic [WIDTH-1:0] q);
// This construct should be used to infer latched logic.
        always_latch
                begin
                        if (ph) q <= d;
                end

endmodule

module latch_158r #(parameter WIDTH = 8)
          (input  logic            ph, reset,
           input  logic [WIDTH-1:0] d,
           output logic [WIDTH-1:0] q);
// This construct should be used to infer latched logic.
        always_latch
                begin
                        if (reset)
                                begin
                                        q <= 0;
                                end
                        else
                                begin
                                        if (ph)
                                                begin
                                                        q <= d;
                                                end
```

```
                    end
          end

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 mux4 #(parameter WIDTH = 8)
       (input  logic [WIDTH-1:0] d0, d1, d2, d3,
        input  logic [1:0]       s,
        output logic [WIDTH-1:0] y);

  always_comb
    case (s)
      2'b00: y = d0;
      2'b01: y = d1;
      2'b10: y = d2;
      2'b11: y = d3;
    endcase
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_158 #(WIDTH) master(ph2, d, mid);
  latch_158 #(WIDTH) slave(ph1, mid, q);
endmodule
```

Appendix C: Schematics

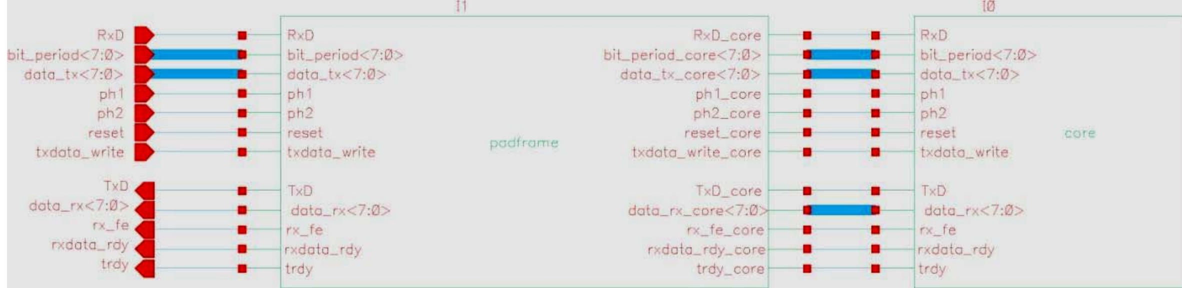

Figure 4: Chip Schematic

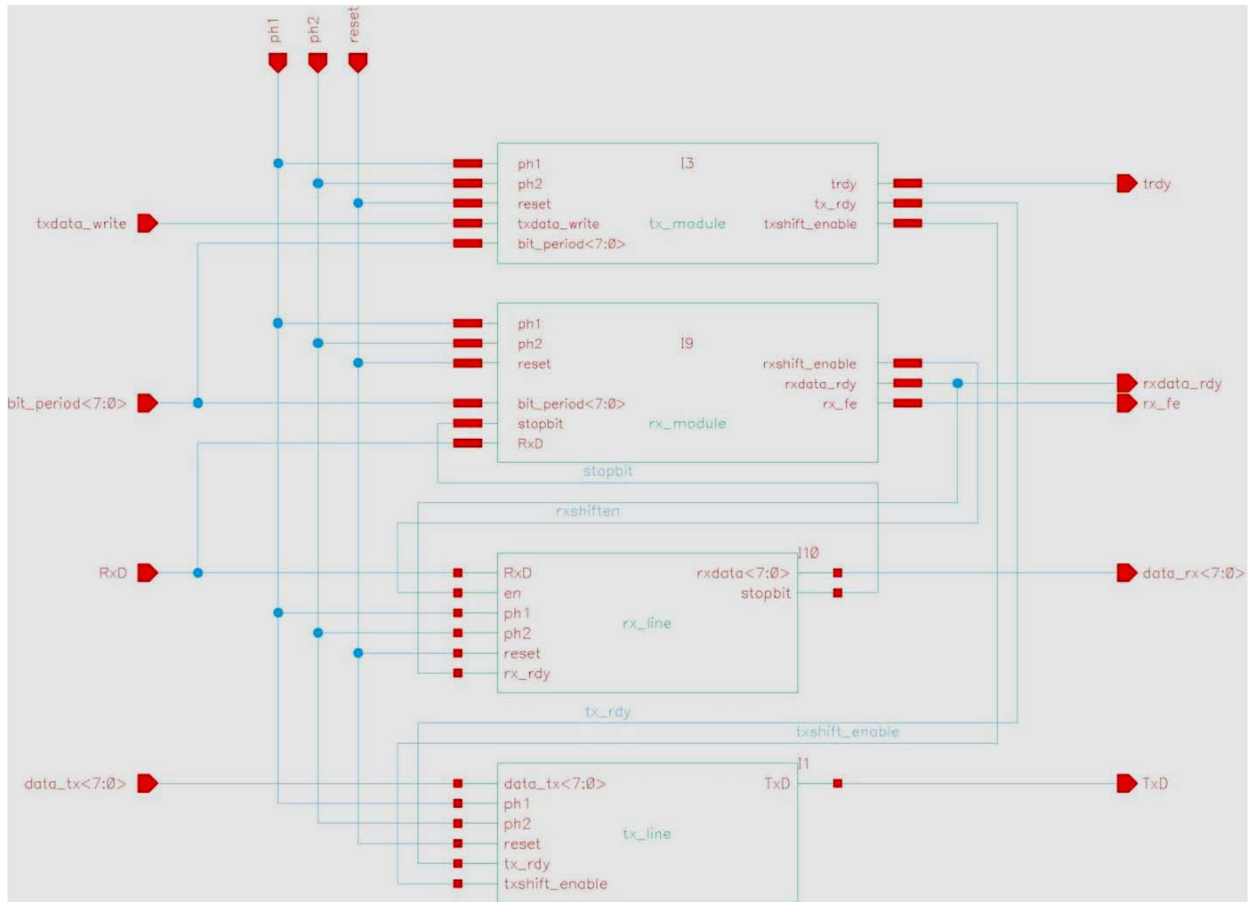

Figure 5: Core Schematic

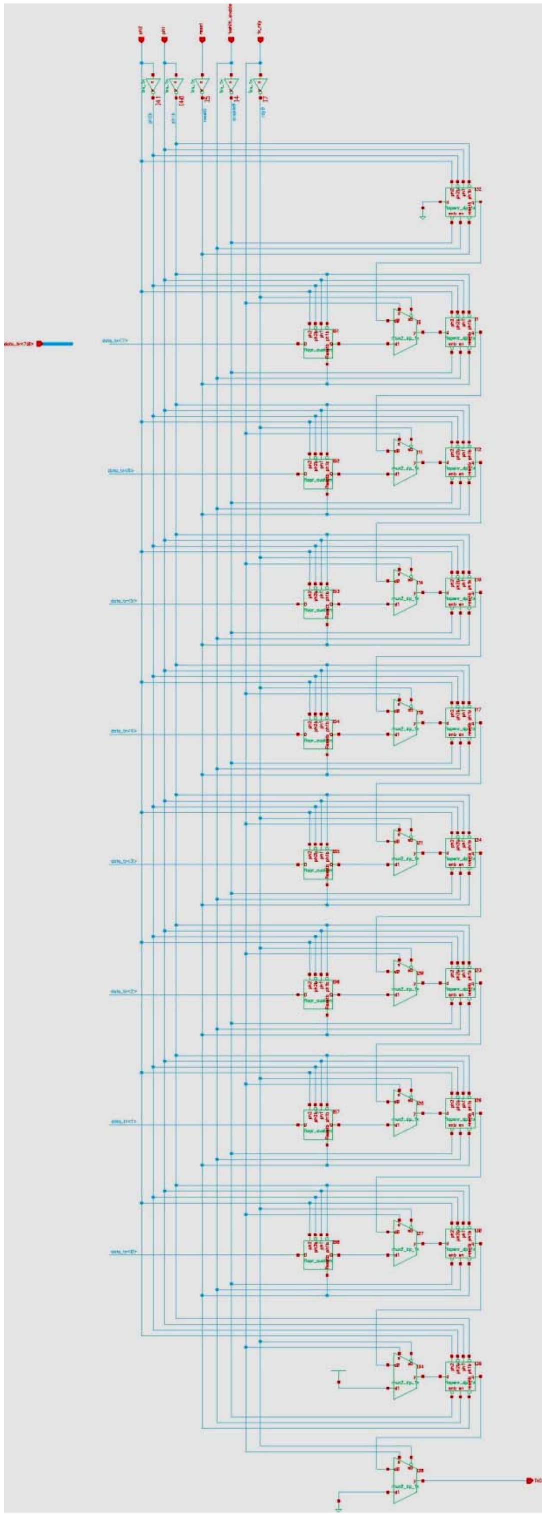

Figure 6: tx_line schematic

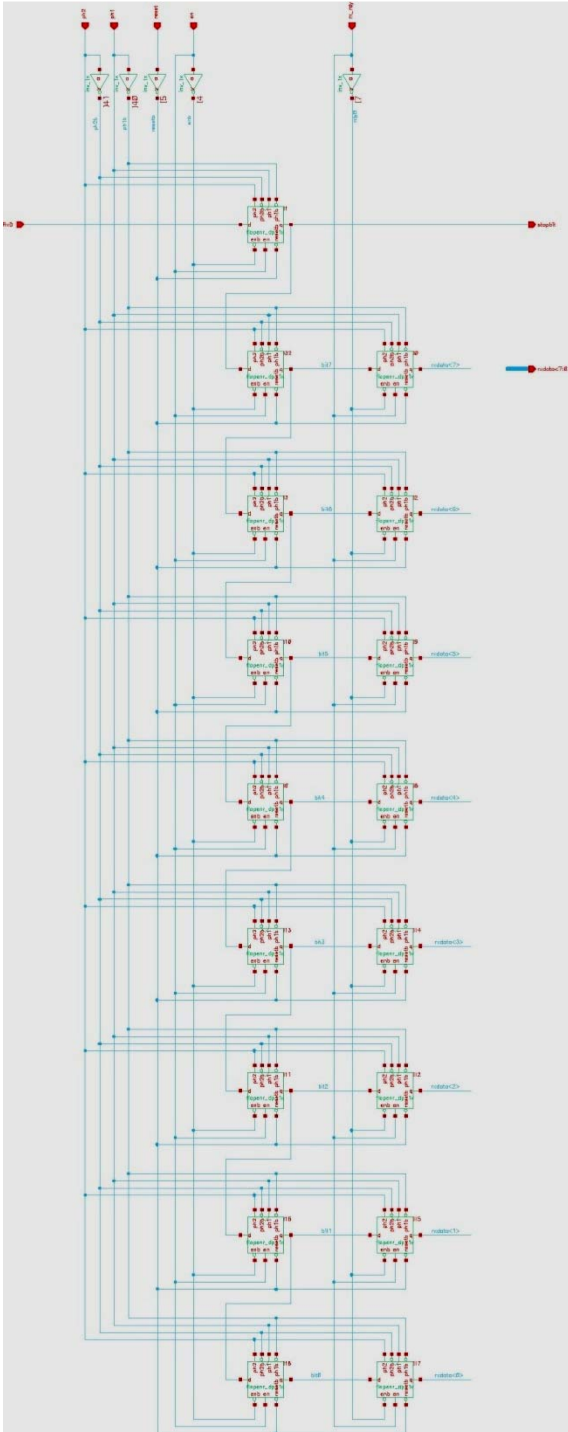

Figure 7: rx_line schematic

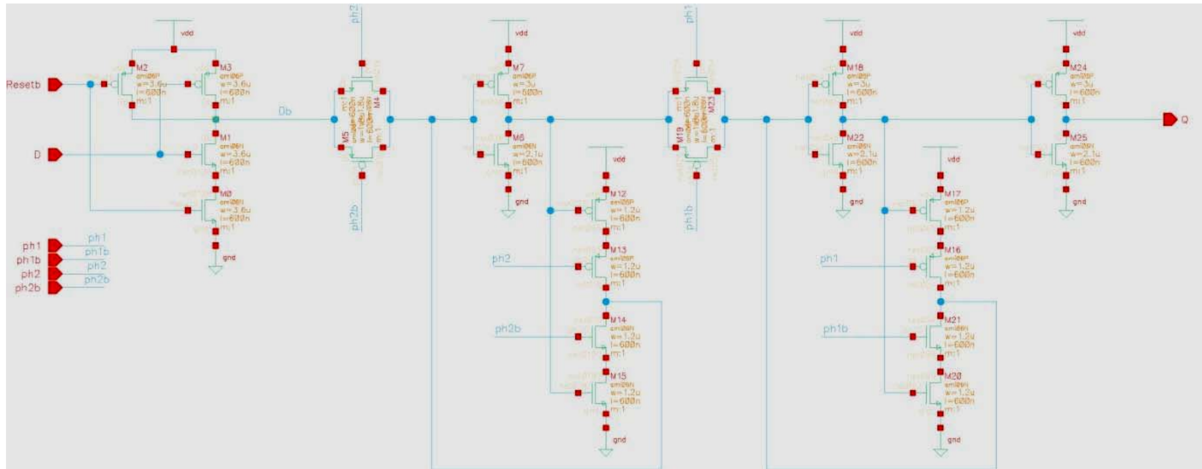

Figure 8: Custom flopr schematic

Appendix D: Layouts

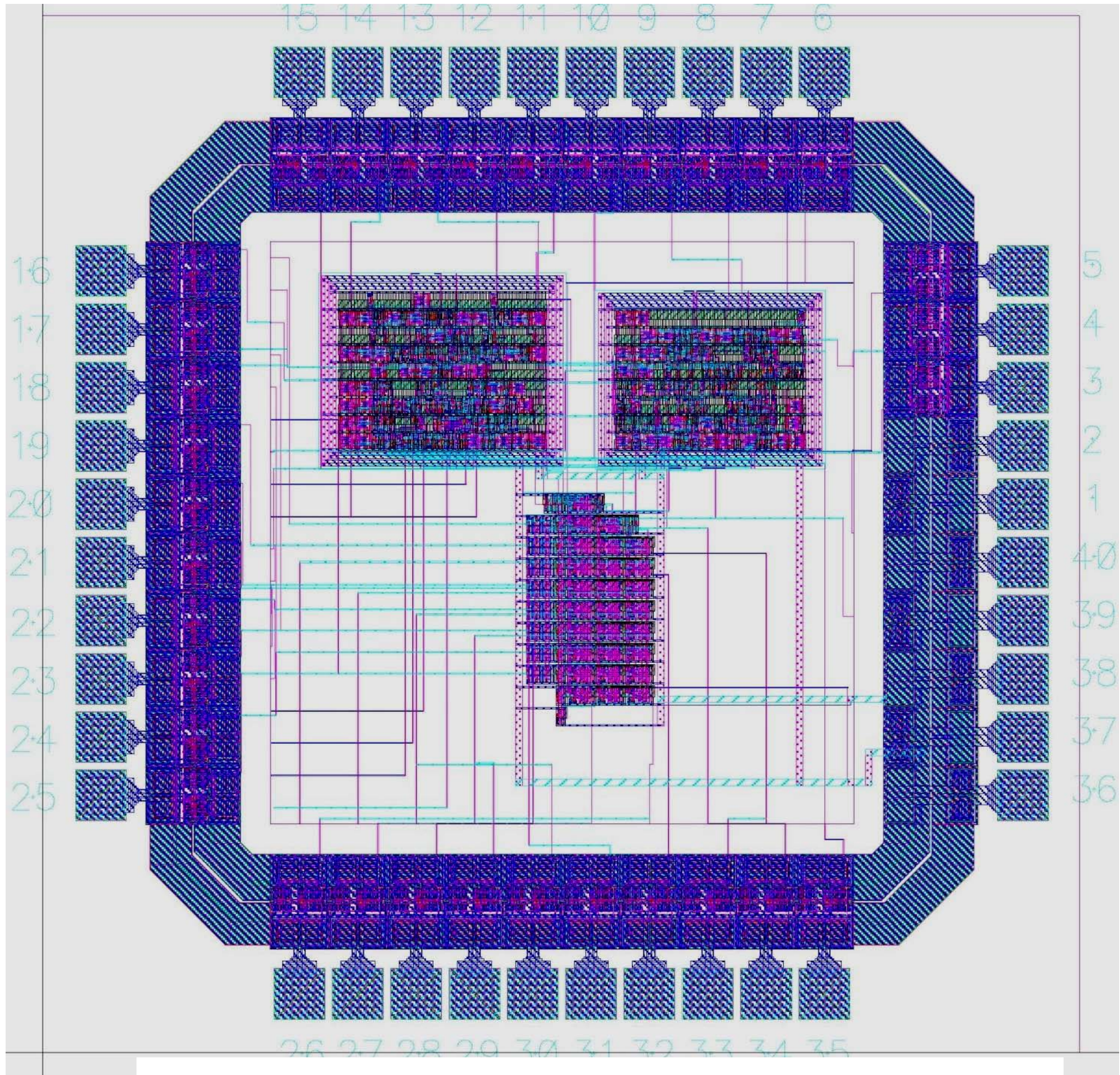

Figure 9: Chip Layout

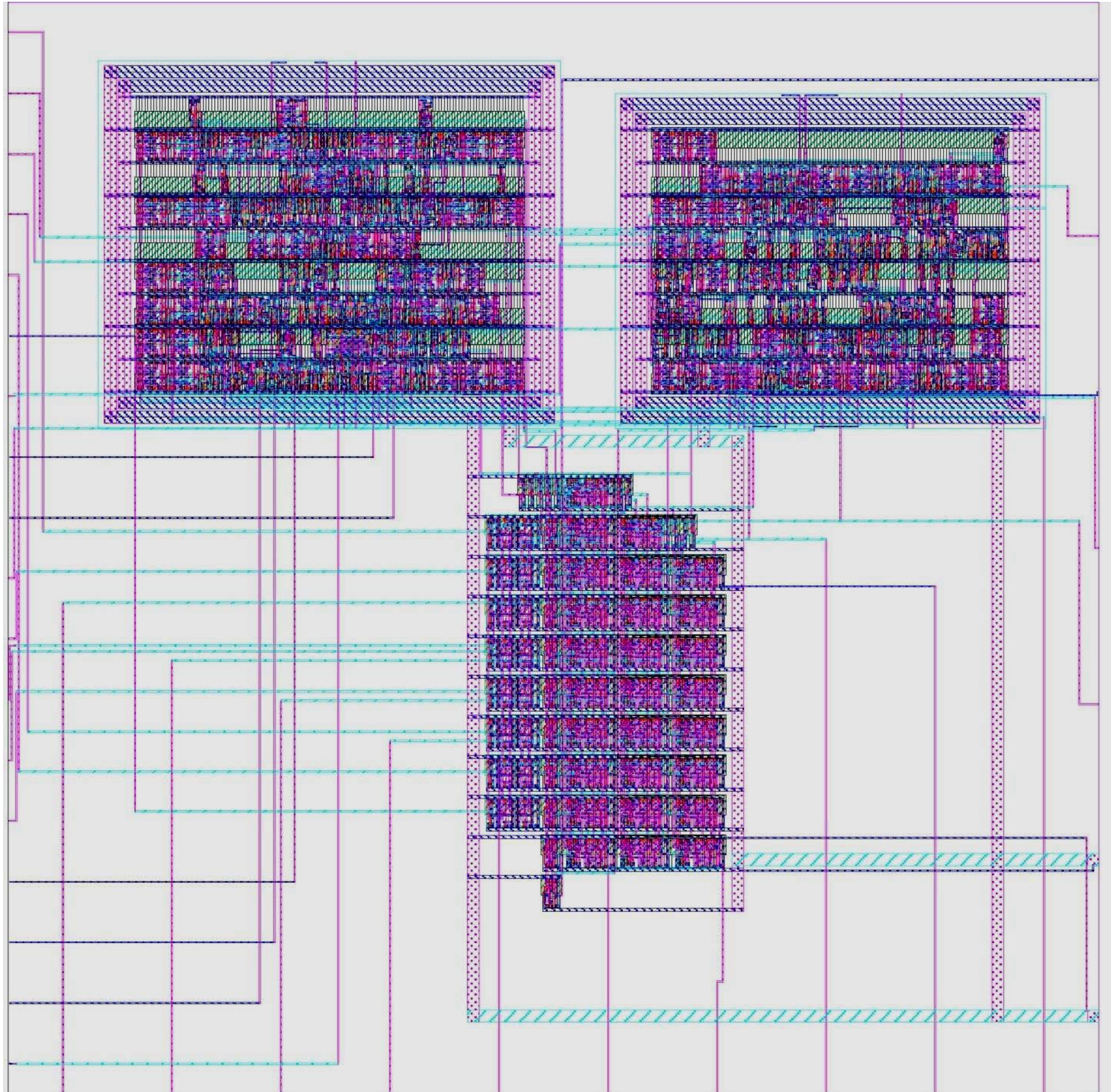

Figure 10: Core layout

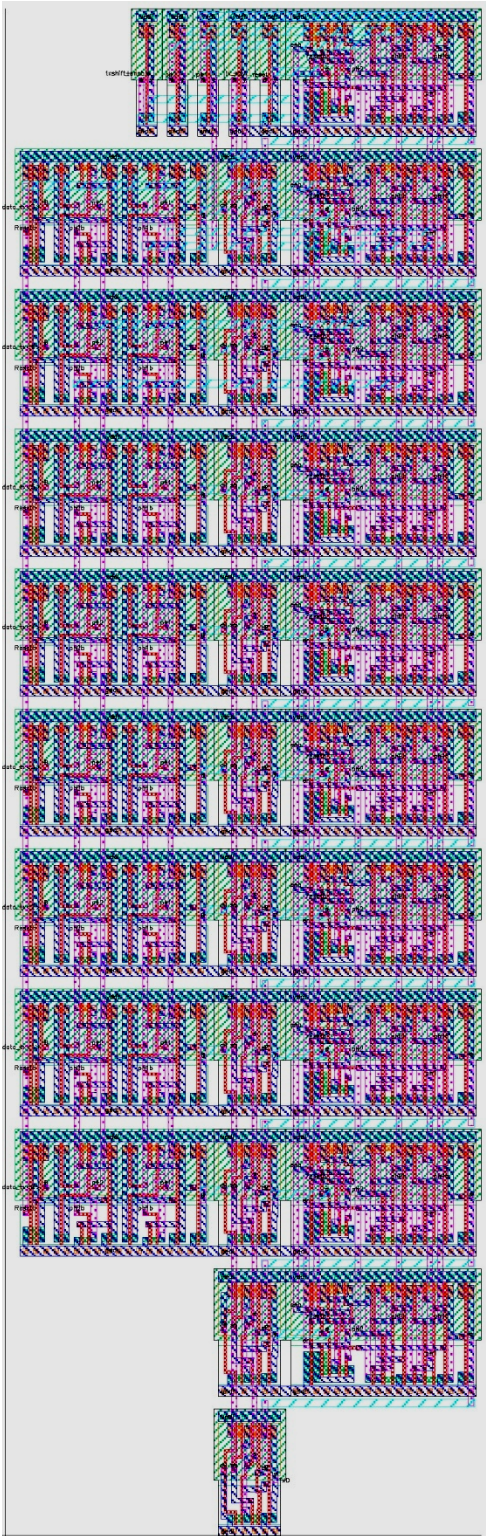

Figure 11: tx_line layout

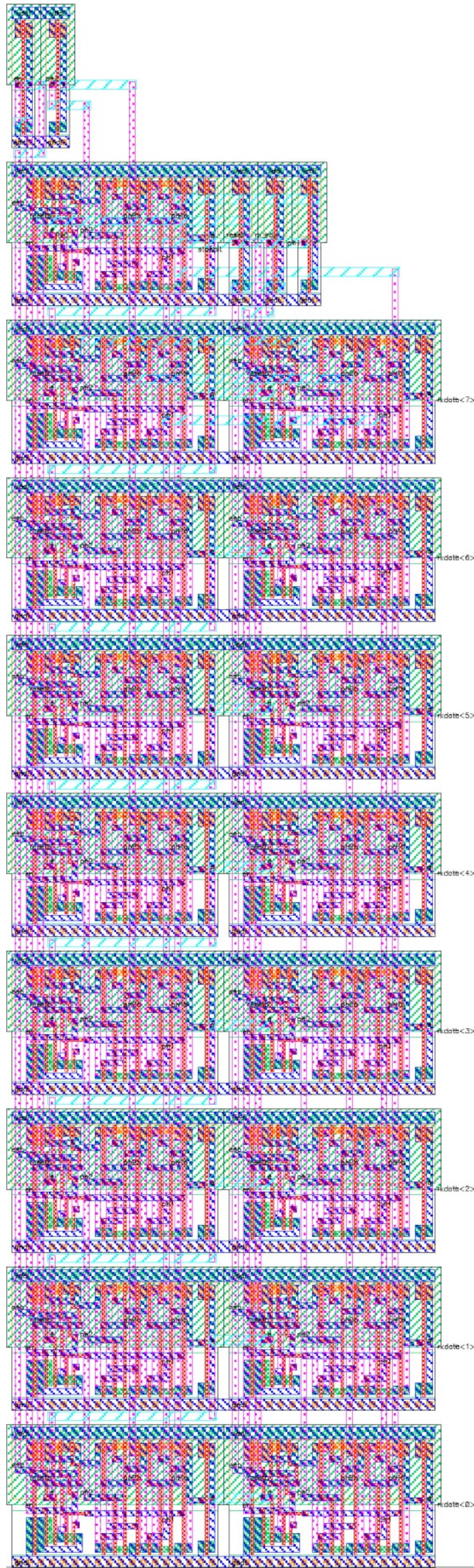

Figure 12: rx_line layout

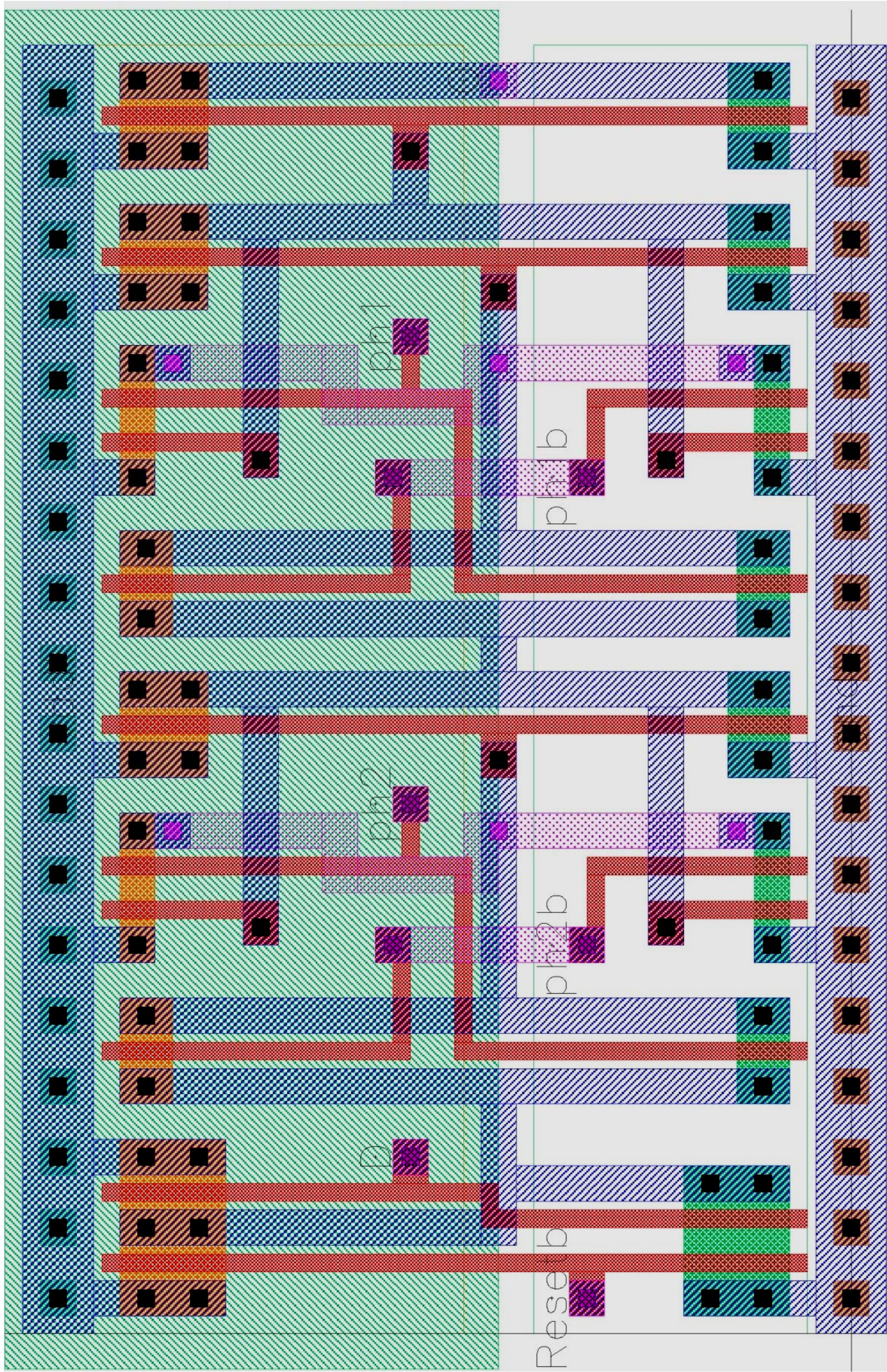

Figure 13: Custom flopr layout