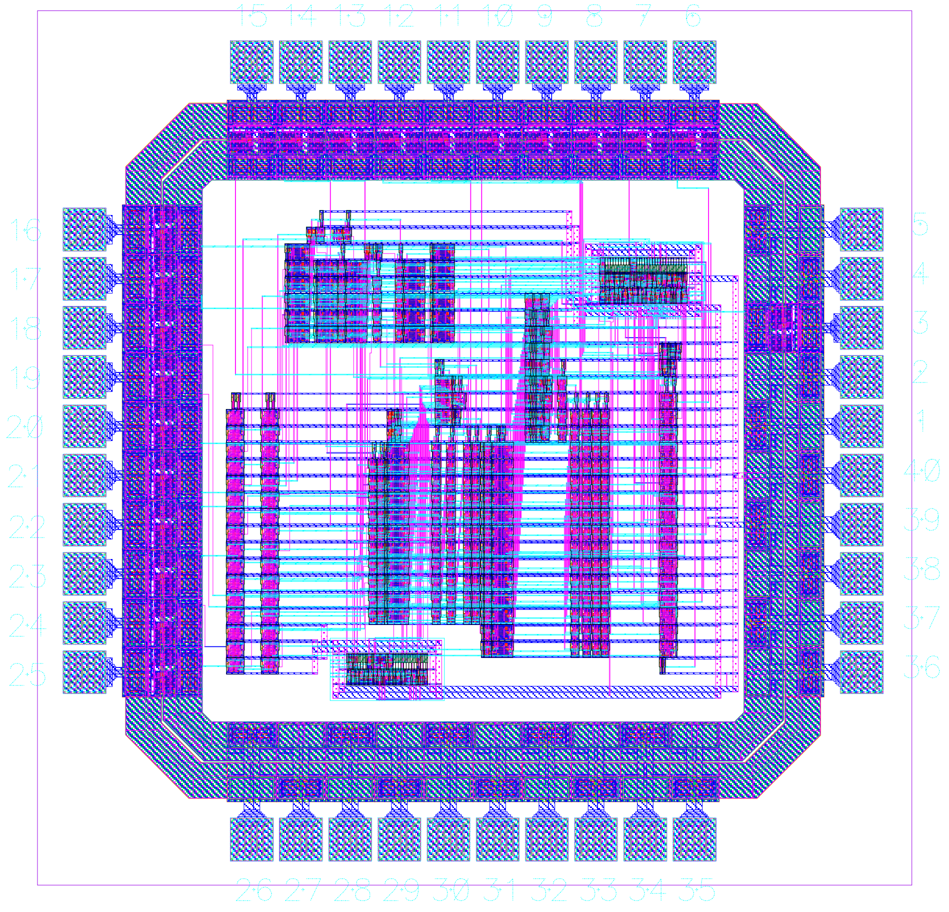
Half-Precision Floating Point Addition Unit

Tynan McAuley

April 18, 2011



Introduction

The goal of this project is to make a chip capable of adding two half-precision floating point numbers. The adder employs round-to-zero, and treats denormalized numbers as underflow. This project does not aim to achieve any particular energy use or delay through the chip, but rather implement a simple addition algorithm of floating point numbers, which will be briefly explained in the Specifications section.

The chip, including a padframe, is laid out in a 0.6 μm CMOS process, and fits within a single MOSIS TinyChip unit, or a 1.5 mm $\times$ 1.5 mm square.

Specifications

Floating point addition involves encoding inputs and outputs to properly represent the different parts of a number represented in scientific notation. Further, the addition requires a nontrivial algorithm to properly decode, add, and re-encode the floating point values. This section will describe both the encoding and algorithm, as well as the inputs and outputs to the chip.

Floating Point Encoding

The inputs and outputs are encoded according to the "binary16" interchange format, specified in IEEE Standard 754-2008[1], and illustrated in Table 1.

Sign - 1 bit	Exponent - 5 bits	Mantissa - 10 bits

Table 1: IEEE "binary16" floating point format.

This encoding represents a number whose value is given in Equation 1.

$$(-1)^{\text{Sign}} \times 1.\text{Mantissa} \times 2^{\text{Exponent}-15_{10}} \tag{1}$$

Note that the mantissa encoding omits a leading 1, which allows extra room for precision in the mantissa. Additionally, to find the value of the exponent, subtract 15_{10} from the encoded exponent. In this way, the exponent covers a range of positive and negative numbers without using a two's complement encoding.

The encoding of floating point numbers also includes several exceptions, tabulated in Table 2.

Exception	**Encoding**
NaN	0_11111_1111111111
$+\infty$	0_11111_0000000000
$-\infty$	1_11111_0000000000
Underflow	0_00000_0000000000

Table 2: Floating point exceptions.

The NaN exception, short for "not a number," covers any cases where addition generates an illegal output. This includes whenever the input is a NaN, or when $+\infty$ is added to $-\infty$. The two infinity exceptions are triggered when addition results in an overflow, and the sign of the larger

[1] *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society, 29 August 2008.

input determines the sign of infinity. The underflow exception covers whenever an input exponent field is 0, or when the output exponent field is 0.

Floating Point Addition Algorithm

The algorithm used in this chip was adapted from several sources ([1], [3], and [4]), but closely follows the naive implementation of floating point addition. A particularly useful set of diagrams describing this algorithm are available in [4].

Input Unpacking and Comparison

The first step is to compare the two inputs and determine which number's absolute value is larger. The two inputs are then ordered so that the smaller input can be adjusted to make addition simpler, and the leading 1's are added onto the mantissas. The smaller input's mantissa is shifted right by the difference of the two input exponents, so that the two mantissas can be added in a standard adder circuit. To ensure that as little precision is lost as possible, two bits are added onto the least-significant end of the mantissa, a guard bit and then a sticky bit. The guard bit is equal to the last bit shifted off, and increases precision during the forthcoming normalization step. The sticky bit is the bitwise OR of all the bits shifted off, so if any 1's were shifted off the mantissa, we set the sticky bit to 1.

Add Mantissas

At this point, the sign of the inputs is taken into account. The operation (addition or subtraction) is determined by XORing the two input signs (0 corresponds to addition, 1 to subtraction). The operation signal then carries-in to the adder, so that subtraction is implemented through that carry-in and the conditional inversion of the shifted mantissa. The larger input's mantissa has two 0's appended to it's least-significant end to match the guard and sticky bits in the shifted mantissa. To properly sign the output of this operation, the carry-out bit of the adder's output is ANDed with the inverse of the operation signal (if we're subtracting, the carry-out should always be 0).

Normalization and Output Packaging

For the output to be properly encoded as a half-precision floating point number, the MSB must be a 1, which the encoding will hide. Converting the number into this form is called "normalization." To normalize the number, a Leading-Zero Detector finds the location of the most-significant 1 in the mantissa. Note that since the mantissa addition created a carry-out, the most-significant 1 should be located in the second most-significant bit location of the mantissa. To increase the ease of normalization, the mantissa and exponent were biased, but right shifting the mantissa by 1 and adding 1 to the exponent (note that this exponent is chosen). The biased mantissa was then shifted left and the exponent subtracted from by the amount dictated by the Leading-Zero Detector. The resulting 15-bit number has the two most-significant and three least-significant bits stripped from it to form the properly encoded mantissa (the top two come from the hidden 1 and an overflow bit, while the bottom three come from the guard bit, the sticky bit, and the biasing bit).

This mantissa is joined with the normalized exponent and the output sign (the sign of the larger input) to form a complete half-precision floating point number. This output is then selected as either this number, all zeros for underflow, signed infinity for overflow, or NaN depending on the exceptions signals.

Inputs and Outputs

The inputs and outputs for the floating point adder are in Table 3.

Direction	Signal Name	Description
Input	sdi_a	Serial Data Input A
Input	sdi_b	Serial Data Input B
Input	scka	Clock 1
Input	sckb	Clock 2
Output	y[15:0]	16-bit Floating Point Output
Output	scka_sense	Clock 1 Debug Signal

Table 3: List of all inputs and outputs on the chip.

Floorplan

Figure 1 contains the chip floorplan proposed at this project's outset, while the actual layout can be seen on this report's title page.

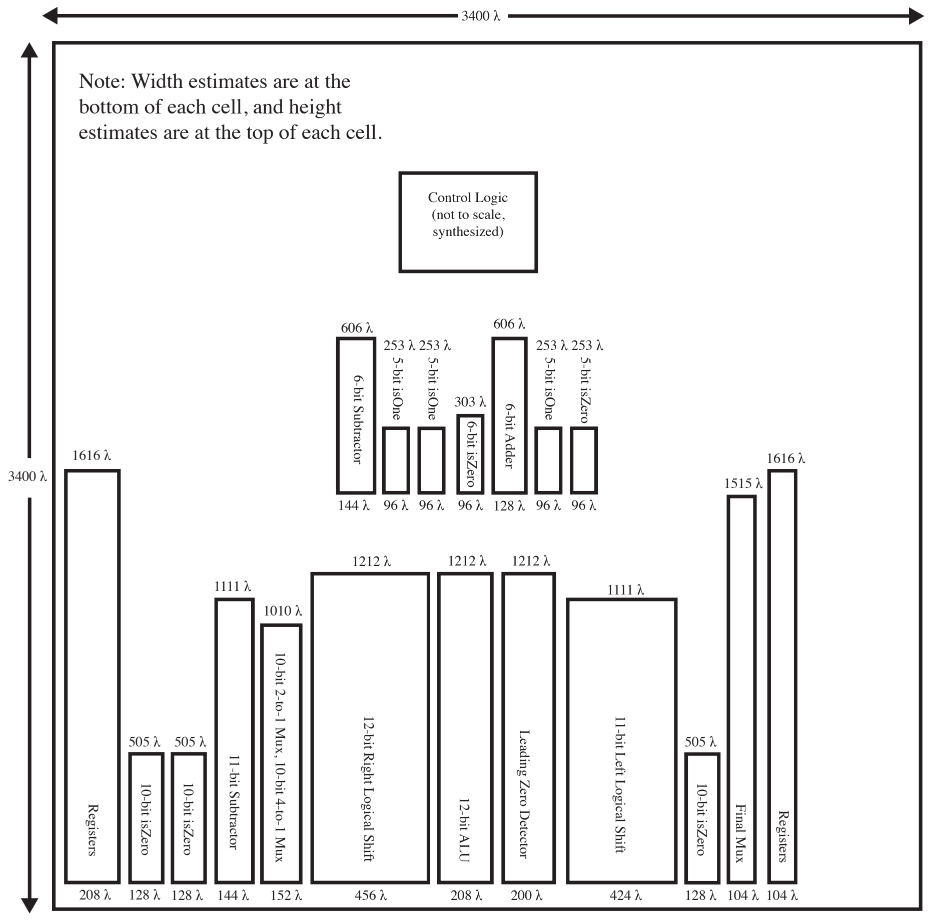


Figure 1: Proposed Floorplan

In the actual layout, the left side of the chip contains two vertical bars, which are the shift registers for the two floating point inputs. The block in the top-left corner is the exponent datapath, the block in the center is the mantissa datapath (including the small block in the lower-left corner), the block at the top-right corner is the control logic, and the vertical bar on the right side is the output multiplexer.

While the floorplans are not markedly different, there are some notable differences. First, all of the "isZero" and "isOne" detectors in both the mantissa and exponent datapaths from the proposed floorplan were moved into the control logic in the actual floorplan. However, the exponent datapath in the actual floorplan contains three extra 5-bit addition units which were necessary to calculate the initial mantissa shift amount and the biased exponent.

In the mantissa datapath, the two shifters' widths were reduced, and the total size of the leading zero detector was reduced as well. However, the layout of the leading zero detector did not take into account fitting gracefully into the datapath, so it was forced above the datapath to allow for metal3 lines to track horizontally into the left-shifter. The leading zero detector can be clearly seen in the chip layout, protruding above the mantissa datapath between the exponent datapath and control logic.

The last major discrepancy pertains to the adder's output. The project proposal dictated that the output would be shifted out in the same was the two inputs were shifted into the chip. However, it was decided to send the entire 16-bit output to 16 different pins on the chip to simplify chip design and test vector creation. As such, the last set of shift registers from the floorplan was omitted.

Figure 2 shows slice plans for the mantissa and exponent datapaths, illustrating the layout of all of the functional units and their interconnects. Additionally, Figure 4 shows the padframe schematic, with input and output pins labelled with their pin numbers on the chip.

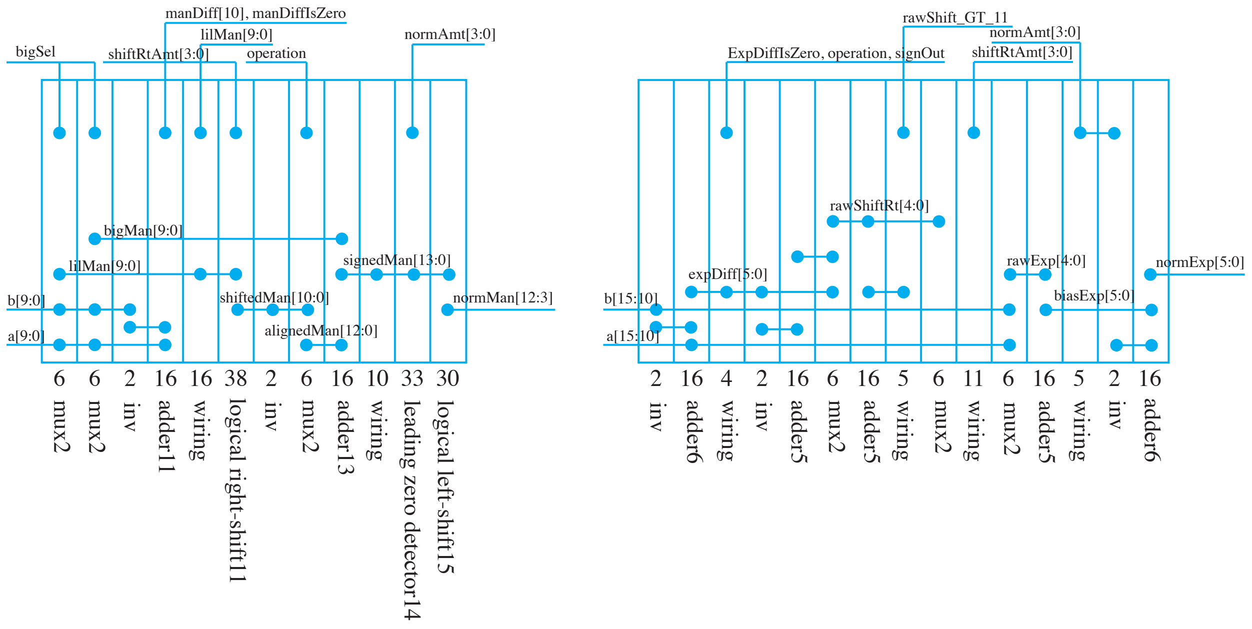


Figure 2: Slice plans for mantissa (left) and exponent (right) datapaths. Widths quoted in wire tracks (8λ).

Verification

To verify the adder's functionality during the Verilog and schematic stages of the design process, two sets of testvectors were developed. One contained 352 simple hand-generated vectors to test a few edge cases (NaN, Infinity) as well as simple addition cases. The other was generated by a set of Perl scripts to test a wide variety of cases. The exponent and mantissa values were chosen to test the corners of the adder's performance, as well as a number of basic addition cases. This file contained 1827904 testvectors. The Verilog model and schematic both passed these two testbenches.

The layout then needed to be verified by using the Design Rules Check (DRC) and Layout-Versus-Schematic (LVS) tests. DRC checks to see if any part of the layout violates MOSIS design rules, and LVS ensures that the layout and schematic netlists are identical. The final chip layout passed both DRC and LVS.

To prepare for tapeout, a CIF version of the chip layout was created. To ensure that MOSIS will receive a functional layout file, this CIF file was imported as a layout. This imported layout passed both DRC and LVS, with the exception of the 144 pad frame DRC warnings.

Postfabrication Test Plan

If fabricated, the chip will return from MOSIS as a 40-pin DIP. This DIP will be inserted into a TestosterIC Device Under Test (DUT) board from One Hot Logic. This allows a TestosterIC Brain Box to stream test vectors from a computer to the chip and then stream the chip's output back to the computer. The testvectors currently developed for the Verilog testbench must be converted into the IRSIM format to accomplish this testing. The chip will be judged to be functioning correctly if the chip passes all of the IRSIM testvectors.

Design Time

Design time on each section of the project can be broken down in Table 4.

Project Section	**Hours Spent**
Project Proposal	20
Verilog Model	20
Testvector Generation	8
Schematic	15
Layout	30
Total:	**93**

Table 4: Time spent on each section of the project.

File Locations

All important files are on Harvey Mudd College's "Chips" server.

- Verilog Model: /home/tmcauley/vlsiFinal/fpAddHalf.v
- Handwritten Testvectors: /home/tmcauley/vlsiFinal/fp16.tv
- Perl-Generated Testvectors: /home/tmcauley/vlsiFinal/tv16

- Perl Testvector Scripts: /home/tmcauley/vlsiFinal/tvgen.pl and /home/tmcauley/vlsiFinal/tvgen_shift.pl
- Synthesis Results (Sticky Bit): /home/tmcauley/IC_CAD/synth/ and /home/tmcauley/IC_CAD/soc/sticky/
- Synthesis Results (Control Logic): /home/tmcauley/IC_CAD/synth_control/ and /home/tmcauley/IC_CAD/soc/control/
- Cadence Libraries: /home/tmcauley/IC_CAD/cadence/fp_add and /home/tmcauley/IC_CAD/cadence/muddlib11/
- CIF: /home/tmcauley/IC_CAD/cadence/fpAddHalf_chip.cif
- PDF Chip Plot: /home/tmcauley/vlsiFinal/chipPlot.pdf
- PDF of Report: /home/tmcauley/vlsiFinal/VLSI_FinalReport.pdf

Acknowledgements

Andrew Carter assisted me in creating a floorplan for the project proposal, which was the most crucial part of this project. The initial plan was to create a single-precision (32-bit) floating point adder, but two iterations of floorplan development revealed that this option would most likely be infeasible.

References

[1] D. Harris and S. Harris, "Digital Building Blocks" in *Digital Design and Computer Architecture*. Burlington, MA: Elsevier, 2007, pp. 249-253.

[2] V. Oklobdzija, "An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis" in *IEEE Transactions on VLSI Systems*, 1994, pp. 124-128.

[3] B. Parhami, "Real Arithmetic" in *Computer Arithmetic - Algorithms and Hardware Designs*, 2nd Ed. Oxford U. Press, 2010.

[4] D. Patterson and J. Hennessy, "Arithmetic for Computers" in *Computer Organization and Design - The Hardware/Software Interface*, 4th Ed. Burlington, MA: Elsevier, 2009, pp. 250-254.

Appendices

Verilog Code

Listing 1: Half-Precision Floating Point Adder

```verilog
/*  Written By:  Tynan McAuley
 *  Started On:  Mar 19, 2011
 *  Last Edited: Mar 23, 2011
 *  Contact:     tynan_mcauley@hmc.edu
 *  Purpose:     Naive implementation of half-precision floating point
 *               addition, but without handling subnormal numbers. Also,
 *               rounding is accomplished through trunctation
 *
 *
 *               Half-precision floating point numbers take the following form:
 *
 *               [15]  [14:10]   [9:0]
 *               sign  exponent  mantissa
 */

module fpAddHalf( input             sdi_a, sdi_b,
                  input             scka, sckb,
                  output    [15:0]  y );

  reg         a_buf, b_buf; // Buffers in between the two shift latches
  reg  [15:0] a, b;         // Shifted in inputs
  wire [5:0]  expDiff;      // Difference between exponents for inputs a and b
  wire [10:0] manDiff;      // Difference between mantissas for inputs a and b
  wire        expDiffIsZero;// Is expDiff equal to 0?
  wire        manDiffIsZero;// Is manDiff equal to 0?
  wire        bigSel;       // If 1, b is bigger. If 0, a is bigger/equal to b
  wire [4:0]  rawShiftRt;   // How much to shift smaller mantissa right by
  wire [3:0]  shiftRtAmt;   // Same as rawShiftRt, but less than or equal to 11
  wire        operation;    // 1 means subtraction, 0 means addition
  wire [4:0]  rawExp;       // Unnormalized exponent
  wire        signOut;      // Sign of calculated output
  wire [10:0] bigMan;       // Larger input mantissa
  wire [10:0] lilMan;       // Smaller input mantissa
  wire [10:0] shiftedMan;   // Shifted version of smaller input mantissa
  wire        guard;        // Guard bit
  reg  [10:0] mask;         // Mask to acquire sticky bit
  wire        sticky;       // Sticky bit
  wire [12:0] alignedMan;   // Smaller mantissa prepared for addition
  wire [13:0] rawMan;       // Mantissa output from big ALU
  wire [13:0] signedMan;    // Mantissa output with correct leading bit
  wire [3:0]  rawNormAmt;   // Output from leading zero detector
  wire        valid;        // Bit indicating whether signedMan is all 0's
  wire [3:0]  normAmt;      // How much to adjust mantissa/exponent by
  wire [5:0]  biasExp;      // Biased exponent ready for normalization
  wire [14:0] biasMan;      // Biased mantissa ready for normalization
  wire [5:0]  normExp;      // Normalized exponent
  wire [4:0]  expOut;       // Final exponent
  wire [14:0] normMan;      // Normalized mantissa
  wire [9:0]  manOut;       // Final mantissa
  wire [14:0] numOut;       // Calculated output

  wire        expAIsOne;
  wire        expBIsOne;
  wire        expAIsZero;
  wire        expBIsZero;
  wire        manAIsZero;
  wire        manBIsZero;
  wire        AIsNaN;
  wire        BIsNaN;
  wire        AIsInf;
  wire        BIsInf;
  wire        inIsNaN;
  wire        inIsInf;
  wire        inIsDenorm;
  wire        expOutIsOne;
  wire        expOutIsZero;
  wire        manOutIsZero;
  wire        outIsNaN;
  wire        outIsInf;
  wire        outIsDenorm;
  wire        overflow;
  wire        NaN;
  wire        underflow;

  ///////////////////////////////////////////////////////////////////////////////
  // Shift in Inputs
  ///////////////////////////////////////////////////////////////////////////////

  always @(sckb, sdi_a, sdi_b) begin
    if (sckb) begin
      a_buf <= sdi_a;
      b_buf <= sdi_b;
    end
  end

  always @(scka, a_buf, b_buf) begin
    if (scka) begin
      a <= {a[14:0], a_buf};
      b <= {b[14:0], b_buf};
    end
  end

  ///////////////////////////////////////////////////////////////////////////////
  // Compare Exponents
  ///////////////////////////////////////////////////////////////////////////////
  assign expDiff = {1'b0, a[14:10]} + {1'b1, ~b[14:10]} + 1;
  assign manDiff = {1'b0, a[9:0]}   + {1'b1, ~b[9:0]  } + 1;

  // Before deciding which input is larger, we need to check if they have the
  // same exponents. If so, then we need to compare the mantissas as well.
  assign expDiffIsZero = ~|expDiff;
  assign manDiffIsZero = ~|manDiff;

  // If the exponents are the same, then we must check the sign of the
  // mantissa comparison. If the exponents aren't the same, then we can just
  // go off of the expDiff.
  assign bigSel = expDiffIsZero ? manDiff[10] : expDiff[5];

  // We want the shift amount to be the absolute value of expDiff.
  assign rawShiftRt = expDiff[5] ? ~expDiff[4:0] + 1 : expDiff[4:0];
  assign shiftRtAmt = (rawShiftRt > 11) ? 4'b1011 : rawShiftRt[3:0];
```

```
// If both signs of the inputs are the same, then the ALU does addition. If
// they're different, then the ALU does subtraction. The smaller number
// always is two's complemented.
assign operation = a[15] ^ b[15];

// Use the bigSel and operation control signals to choose which input is
// larger and whether or not the smaller one should be two's complemented
assign rawExp  = bigSel ? b[14:10] : a[14:10];
assign signOut = bigSel ? b[15]    : a[15];
assign bigMan  = {1'b1, bigSel ? b[9:0] : a[9:0]};
assign lilMan  = {1'b1, bigSel ? a[9:0] : b[9:0]};

// Align mantissas, create guard/sticky bits, and selectively complement
assign shiftedMan = lilMan >> shiftRtAmt;
assign guard      = ((shiftRtAmt == 0) || (rawShiftRt > 11)) ?
                      1'b0 :
                      lilMan[shiftRtAmt-1];

// mask will tell us which bits have been shifted out. We can then bitwise
// AND it with lilMan to get the exact bits that were shifted out, and then
// OR that outcome to get the sticky bit
always @(*) begin
  casez ({shiftRtAmt, (rawShiftRt > 11)})
    5'b00000: mask = 11'b000_0000_0000;
    5'b00010: mask = 11'b000_0000_0000;
    5'b00100: mask = 11'b000_0000_0001;
    5'b00110: mask = 11'b000_0000_0011;
    5'b01000: mask = 11'b000_0000_0111;
    5'b01010: mask = 11'b000_0000_1111;
    5'b01100: mask = 11'b000_0001_1111;
    5'b01110: mask = 11'b000_0011_1111;
    5'b10000: mask = 11'b000_0111_1111;
    5'b10010: mask = 11'b000_1111_1111;
    5'b10100: mask = 11'b001_1111_1111;
    5'b10110: mask = 11'b011_1111_1111;
    5'b????1: mask = 11'b111_1111_1111;
    default:  mask = 11'b000_0000_0000;
  endcase
end

assign sticky     = |(lilMan[10:0] & mask[10:0]);
assign alignedMan = operation ? ~{shiftedMan, guard, sticky} :
                                 {shiftedMan, guard, sticky};

////////////////////////////////////////////////////////////////////////////////
// Add Mantissas
////////////////////////////////////////////////////////////////////////////////

assign rawMan    = alignedMan + {bigMan, 2'b0} + operation;
// In subtraction, we don't want the carry-out, so we replace it with a zero.
assign signedMan = {rawMan[13] & ~operation, rawMan[12:0]};

////////////////////////////////////////////////////////////////////////////////
// Normalize/Package
////////////////////////////////////////////////////////////////////////////////

// In normalization, we will either right-shift by 1, do nothing, or
// left-shift by some variable amount. By biasing the normalization, we
// either do nothing or we only left-shift.
```

```
  lzd14 lzd(signedMan, rawNormAmt, valid);

  assign normAmt = valid ? rawNormAmt : 4'b0;

  assign biasExp = rawExp + 1;
  assign biasMan = {1'b0, signedMan};
  assign normExp = biasExp + ~{2'b0, normAmt} + 1;
  assign expOut  = normExp[4:0];

  assign normMan = biasMan << normAmt;
  assign manOut  = normMan[12:3];
  // Discard the guard/sticky bits (since we're rounding down), discard
  // the two upper bits (which are the overflow bit and the hidden 1), and
  // discard the bit just above the guard/sticky bits, which was added in when
  // we biased the mantissa.

  // Calculated output
  assign numOut = {expOut, manOut};

  // Input exception signals
  assign expAIsOne  = &a[14:10];
  assign expBIsOne  = &b[14:10];
  assign expAIsZero = ~|a[14:10];
  assign expBIsZero = ~|b[14:10];
  assign manAIsZero = ~|a[9:0];
  assign manBIsZero = ~|b[9:0];
  assign AIsNaN     = expAIsOne & ~manAIsZero;
  assign BIsNaN     = expBIsOne & ~manBIsZero;
  assign AIsInf     = expAIsOne & manAIsZero;
  assign BIsInf     = expBIsOne & manBIsZero;
  assign inIsNaN    = AIsNaN | BIsNaN | (AIsInf & BIsInf & operation);
  assign inIsInf    = AIsInf | BIsInf;
  assign inIsDenorm = expAIsZero | expBIsZero;
  assign zero       = expDiffIsZero & manDiffIsZero & operation;

  // Output exception signals
  assign expOutIsOne  = &expOut;
  assign expOutIsZero = ~|expOut;
  assign manOutIsZero = ~|manOut;
  assign outIsInf     = expOutIsOne & ~normExp[5];
  assign outIsDenorm  = expOutIsZero | normExp[5];

  // Final exception signals
  assign overflow  = (inIsInf | outIsInf) & ~zero;
  assign NaN       = inIsNaN;
  assign underflow = inIsDenorm | outIsDenorm | zero;

  // Choose output from exception signals
  assign y = NaN       ? 16'b0111_1111_1111_1111 :
             overflow  ? {signOut, 15'b111_1100_0000_0000} :
             underflow ? 16'b0000_0000_0000_0000 :
                         {signOut, numOut};

endmodule

`include "lzd.v"
```

Listing 2: Leading Zero Detector

```
/*
 *  Source:
 *    An Algorithmic and Novel Design of a Leading Zero Detector Circuit:
 *      Comparison with Logic Synthesis
 *    By Vojin G. Oklobdzija.
 *    IEEE Transactions on Very Large Scale Integration (VLSI) Systems,
 *      Vol. 2, No. 1, March 1994
 */

module lzd14( input  [13:0] a,
              output [3:0]  position,
              output        valid );

  wire [2:0] pUpper, pLower;
  wire       vUpper, vLower;

  lzd8 lzd8_1( a[13:6],         pUpper[2:0], vUpper );
  lzd8 lzd8_2( {a[5:0], 2'b0},  pLower[2:0], vLower );

  assign valid       = vUpper | vLower;
  assign position[3] = ~vUpper;
  assign position[2] = vUpper ? pUpper[2] : pLower[2];
  assign position[1] = vUpper ? pUpper[1] : pLower[1];
  assign position[0] = vUpper ? pUpper[0] : pLower[0];

endmodule



module lzd8( input  [7:0] a,
             output [2:0] position,
             output       valid );

  wire [1:0] pUpper, pLower;
  wire       vUpper, vLower;

  lzd4 lzd4_1( a[7:4], pUpper[1:0], vUpper );
  lzd4 lzd4_2( a[3:0], pLower[1:0], vLower );

  assign valid       = vUpper | vLower;
  assign position[2] = ~vUpper;
  assign position[1] = vUpper ? pUpper[1] : pLower[1];
  assign position[0] = vUpper ? pUpper[0] : pLower[0];

endmodule



module lzd4( input  [3:0] a,
             output [1:0] position,
             output       valid );

  wire pUpper, pLower, vUpper, vLower;

  lzd2 lzd2_1( a[3:2], pUpper, vUpper );
  lzd2 lzd2_2( a[1:0], pLower, vLower );
  assign valid       = vUpper | vLower;
  assign position[1] = ~vUpper;
  assign position[0] = vUpper ? pUpper : pLower;

endmodule



module lzd2( input  [1:0] a,
             output       position,
             output       valid );

  assign valid    = a[1] | a[0];
  assign position = ~a[1];

endmodule
```

Listing 3: Verilog Testbench

```
`timescale 1ns / 100ps

module fptest();
  reg         a, b, scka, sckb, clk, reset;
  reg  [15:0] y_exp;
  wire [15:0] y;

  reg  [20:0] vectornum, errors;
  reg  [17:0] testvectors[20000000:0];

  // instantiate Device Under Test (DUT)
  fpAddHalf dut(a, b, scka, sckb, y);

  // generate clock
  always begin
    clk = 1; scka = 0; sckb = 0;
    #1;
    clk = 1; sckb = 1; scka = 0;
    #1;
    clk = 1; sckb = 0; scka = 0;
    #1;
    clk = 1; sckb = 0; scka = 1;
    #1;
    clk = 1; sckb = 0; scka = 0;
    #1;
    clk = 0; sckb = 0; scka = 0;
    #5;
  end

  // load vectors at start
  initial begin
    //$readmemb("tv16", testvectors); // load test vectors
    $readmemb("fp16reg.tv", testvectors); // load test vectors
    vectornum = 0; errors = 0;
    reset = 1; #17; reset = 0; // come out of reset before cycle 2
  end

  // apply test vectors on rising edge of clk
  always @(posedge clk) begin
```

```
    #1; {a, b, y_exp} = testvectors[vectornum];
  end

  // check results on falling edge of clk
  always @(negedge clk) begin
    if (!reset) begin // skip during reset
      if ((y !== y_exp) && (vectornum > 14)) begin
        $display("Error:   a = %b, b = %b,\ny      = %b\ny_exp = %b\n",
          a, b, y, y_exp);
        errors = errors + 1;
      end
      else begin
        $display("Success: a = %b, b = %b, y = %b", a, b, y);
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 18'bx) begin
        $display("\n\n%d tests completed with %d errors\n",
          vectornum, errors);
        $finish;
      end
    end
  end
endmodule

`include "fpAddHalf.v"
```

Listing 4: Handwritten Testvectors

```
0_0_0000000000000000
0_0_0000000000000000
1_1_0000000000000000
1_1_0000000000000000
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000101110000000
0_0_0001001100000000
0_0_0010001000000000
0_0_0100000000000000
1_1_0111110000000000
0_0_1111010000000010
1_1_1110010000000101
1_1_1100010000001011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000100111100000
0_0_0000111111000000
0_0_0001101110000000
0_0_0011001100000000
```

```
0_0_0110001000000000
0_0_1100000000000000
1_0_0111110000000000
0_0_1111010000000001
1_1_1110010000000011
1_1_1100010000000111
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0001011111111000
0_0_0010111100000111
0_0_0101111000000000
0_0_0000000000000000
0_0_0111110000000000
1_0_1111010000000000
0_1_1110010000000001
0_1_1100010000000011
0_1_0000000000000000
0_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000101111000000
0_0_0001001110000000
0_0_0010001100000000
0_0_0100001000000000
0_0_0000000000000000
0_1_0000000000000000
1_0_0000000000000000
1_0_0000000000000000
1_0_0000000000000000
1_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000101111000000
0_0_0001001110000000
0_0_0010001100000000
0_0_0100001000000000
1_1_0000000000000000
1_1_0000000000000000
1_1_0000000000000000
1_1_0000000000000000
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
```

```
0_0_0000000000000000
0_0_0000101111100000
0_0_0001001111000000
0_0_0010001110000000
0_0_0100001100000000
0_0_0111111111111111
0_0_1111110000000000
1_0_1111110000000000
1_1_1111010000000010
1_1_1110010000000101
1_1_1100010000001011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0001000011011000
0_0_0001111111111000
0_0_0011111100000111
0_0_0111111111111111
0_0_0111111111111111
0_1_1111110000000000
1_1_1111010000000010
1_1_1110010000000101
1_1_1100010000001011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0001000011011000
0_0_0001111111111000
0_0_0011111100000111
0_0_0111111111111111
0_0_0111111111111111
1_1_1111110000000000
1_1_1111010000000011
1_1_1110010000000111
1_1_1100010000001111
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000101111100000
0_0_0001001111000000
0_0_0010001110000000
0_0_0100001100000000
0_0_0111111111111111
1_0_0111111111111111
1_1_1111110000000000
1_1_1111010000000101
1_1_1110010000001011
1_1_1100010000010111
```

```
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0001000011101100
0_0_0010000000011110
0_0_0011111110000111
0_0_0111111111111111
0_0_0111111111111111
0_1_0111111111111111
1_1_1111110000000000
1_1_1111010000000101
1_1_1110010000001011
1_1_1100010000010111
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0001000011101100
0_0_0010000000011110
0_0_0011111110000111
0_0_0111111111111111
0_0_0111111111111111
1_1_0111111111111111
1_0_1111110000000000
1_1_1111010000000110
1_1_1110010000001101
1_1_1100010000011011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000101011110000
0_0_0001000111101000
0_0_0010000011011000
0_0_0011111111111000
0_0_0111111111111111
0_0_0111111111111111
1_0_0111111111111111
1_0_1111110000000000
1_1_1111010000000100
1_1_1110010000001001
1_1_1100010000010011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0001111111011111
0_0_0011111110000000
0_0_0111111111111111
0_0_0111111111111111
```

```
0_1_0111111111111111
1_0_1111110000000000
1_1_1111010000000100
1_1_1110010000001001
1_1_1100010000010011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000110111101000
0_0_0001100011011000
0_0_0010111111111000
0_0_0101111100000111
0_0_0111111111111111
1_1_0111111111111111
0_1_1111110000000000
1_1_1111010000000110
1_1_1110010000001101
1_1_1100010000011011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000101011110000
0_0_0001000111101000
0_0_0010000011011000
0_0_0011111111111000
0_0_0111111111111111
0_0_0111111111111111
1_0_0111111111111111
0_1_1111110000000000
1_1_1111010000000100
1_1_1110010000001001
1_1_1100010000010011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000110111101000
0_0_0001100011011000
0_0_0010111111111000
0_0_0101111100000111
0_0_0111111111111111
0_1_0111111111111111
0_1_1111110000000000
1_1_1111010000000100
1_1_1110010000001001
1_1_1100010000010011
1_1_0000000000000000
1_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
```

```
0_0_0000000000000000
0_0_0001111111011111
0_0_0011111110000000
0_0_0111111111111111
0_0_0111111111111111
1_1_0111111111111111
0_1_1111110000000000
1_0_1111010000000101
1_0_1110010000001011
1_0_1100010000010111
1_0_0000000000000000
0_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000100111111000
0_0_0001000000000000
0_0_0001110000000000
0_0_0011010000000000
0_0_0110010000000000
0_0_1100010000000000
0_0_0111011111111111
1_0_1111000000000000
0_1_1110000000000001
0_1_1011111111111111
0_1_0000000000000000
0_1_0000000000000000
1_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000110000000000
0_0_0001010000000000
0_0_0010010000000000
0_0_0100010000000000
0_0_0111011111111111
1_0_1110111111111111
1_0_1101111111111111
1_0_1011111111110111
1_0_0000000000000000
1_0_0000000000000000
0_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0111110000000000
0_0_1111011111111111
0_1_1110111111111111
0_1_1101111111111111
0_1_1011111111110111
0_1_0000000000000000
0_1_0000000000000000
```

```
1_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0111110000000000
0_0_1111011111111111
1_0_1110111111111111
1_0_1101111111111111
0_1_1011111111111001
0_1_0000000000000000
0_1_0000000000000000
1_0_0000000000000000
0_1_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0000000000000000
0_0_0001100010111010
0_0_0011000100000011
0_0_0110001000000000
0_0_1100001010000000
```

Perl Code

Listing 5: Testvector Generation Script

```
#!/usr/bin/perl -w
#
# Written By: Tynan McAuley
#
# Generates testvectors for a half-precision floating point adder. Vectors are
# in the form of two 16-bit inputs and a 16-bit output.

use strict;
use warnings;

my $sign_bits = 1;
my $exp_bits = 5;
my $man_bits = 10;
my $bits = $sign_bits + $exp_bits + $man_bits;

my (@vecs, $signA, $signB, $expA, $expB, $manA, $manB, $fname, $i);

my $rand1_man = int(rand(2**$man_bits));
my $rand2_man = int(rand(2**$man_bits));

my $rand1_exp = int(rand(2**$exp_bits));
my $rand2_exp = int(rand(2**$exp_bits));

my @directed_man = (0, 1, 2, 3, 4, 2**($man_bits-2), 2**($man_bits-1)-1,
  2**($man_bits-1), 2**($man_bits-1)+1, 2**($man_bits)-2, 2**($man_bits)-1,
  $rand1_man, $rand2_man);
my @directed_exp = (0, 1, 2, 3, 4, 2**($exp_bits-2), 2**($exp_bits-1)-1,
  2**($exp_bits-1), 2**($exp_bits-1)+1, 2**($exp_bits)-2, 2**($exp_bits)-1,
  $rand1_exp, $rand2_exp);

for ($signA = 0; $signA < 2**$sign_bits; $signA++){
  for ($signB = 0; $signB < 2**$sign_bits; $signB++){
    for ($expA = 0; $expA <= $#directed_exp; $expA++){
      for ($expB = 0; $expB <= $#directed_exp; $expB++){
        for ($manA = 0; $manA <= $#directed_man; $manA++){
          for ($manB = 0; $manB <= $#directed_man; $manB++){
            push (@vecs, [$signA, $directed_exp[$expA], $directed_man[$manA],
              $signB, $directed_exp[$expB], $directed_man[$manB]]);
          }
        }
      }
    }
  }
}

$fname = "tv$bits";
open FILE, ">$fname" || die ("Can't write $fname\n");
for ($i=0; $i <= $#vecs; $i++) {
  $signA = $vecs[$i][0];
  $expA  = $vecs[$i][1];
  $manA  = $vecs[$i][2];
  $signB = $vecs[$i][3];
  $expB  = $vecs[$i][4];
  $manB  = $vecs[$i][5];
  my @sum = &fpSum($signA, $expA, $manA, $signB, $expB, $manB);
  my $signOut = $sum[0];
  my $expOut  = $sum[1];
  my $manOut  = $sum[2];
  print FILE &d2bS($signA) . &d2bE($expA) . &d2bM($manA) . "_" . &d2bS($signB) .
    &d2bE($expB) . &d2bM($manB) . "_" .&d2bS($signOut) . &d2bE($expOut) .
    &d2bM($manOut) . "\n";
}
close(FILE);

sub d2b {
  my $val = shift;
  my $str = "";
  for (my $i=0; $i< $bits; $i++) {
    $str = ($val % 2) . $str;
    $val = int($val/2);
  }
  return $str;
}

sub d2bS {
  my $val = shift;
  my $str = "";
  for (my $i=0; $i< $sign_bits; $i++) {
    $str = ($val % 2) . $str;
    $val = int($val/2);
  }
  return $str;
}
```

```
sub d2bE {
  my $val = shift;
  my $str = "";
  for (my $i=0; $i< $exp_bits; $i++) {
    $str = ($val % 2) . $str;
    $val = int($val/2);
  }
  return $str;
}

sub d2bM {
  my $val = shift;
  my $str = "";
  for (my $i=0; $i< $man_bits; $i++) {
    $str = ($val % 2) . $str;
    $val = int($val/2);
  }
  return $str;
}

sub fpSum {
  my ($ans, $signOut, $expOut, $manOut, $calc);

  my $signA = $_[0];
  my $expA  = $_[1] - 15;
  my $manA  = $_[2] + 2**$man_bits;
  my $signB = $_[3];
  my $expB  = $_[4] - 15;
  my $manB  = $_[5] + 2**$man_bits;

  my $zero      = ($expA == $expB) && ($manA == $manB) && ($signA != $signB);
  my $AIsInf    = ($expA + 15 == 31) && ($manA - 2**$man_bits == 0);
  my $BIsInf    = ($expB + 15 == 31) && ($manB - 2**$man_bits == 0);
  my $AIsNaN    = ($expA + 15 == 31) && ($manA - 2**$man_bits != 0);
  my $BIsNaN    = ($expB + 15 == 31) && ($manB - 2**$man_bits != 0);
  my $AIsDenorm = ($expA + 15 == 0);
  my $BIsDenorm = ($expB + 15 == 0);

  $ans = ((-1)**$signA)*$manA*(2**$expA) + ((-1)**$signB)*$manB*(2**$expB);

  $signOut = ($ans < 0) ? 1 : 0;

  $expOut = 0;
  $manOut = abs($ans);

  if ($ans != 0) {
    while (($manOut >= 2048) || ($manOut < 1024)) {
      if ($manOut >= 2048) {
        $manOut = $manOut/2;
        $expOut++;
      }
      elsif ($manOut < 1024) {
        $manOut = $manOut*2;
        $expOut--;
      }
    }
  }

  my $outIsInf    = ($expOut + 15 >= 31);
  my $outIsDenorm = ($expOut + 15 <= 0);

  my $NaN = ($AIsInf && $BIsInf && ($signA != $signB)) || $AIsNaN || $BIsNaN;
  my $overflow = ($AIsInf || $BIsInf || $outIsInf) && !$zero;
  my $underflow = $AIsDenorm || $BIsDenorm || $outIsDenorm || $zero;

  if ($NaN) {
    $signOut = 0;
    $expOut  = 31;
    $manOut  = 1023;
  }
  elsif ($overflow) {
    $expOut = 31;
    $manOut = 0;
  }
  elsif ($underflow) {
    $signOut = 0;
    $expOut  = 0;
    $manOut  = 0;
  }

  $calc = ((-1)**$signOut)*$manOut*(2**$expOut);

  if (!$NaN && !$overflow && !$underflow) {
    $expOut = $expOut + 15;
    $manOut = $manOut - 2**$man_bits;
  }

  if (($calc != $ans) && !$NaN && !$overflow && !$underflow) {
    print "Error! Calculated answer: $calc vs Initial answer: $ans\n";
    print "signA: $signA. expA: " . $expA . ". manA: " . $manA . "\n";
    print "signB: $signB. expB: " . $expB . ". manB: " . $manB . "\n";
  }

  return ($signOut, $expOut, $manOut);
}
```

Listing 6: Testvector Conversion Script

```
#!/usr/bin/perl -w
#
# Written By: Tynan McAuley
#
# Converts a set of testvectors for a half-precision floating point adder to
# simulate shifting single bits into the adder.

use strict;
use warnings;

my $sign_bits = 1;
my $exp_bits = 5;
my $man_bits = 10;

my ($fnameIn, $fnameOut, $line, $a, $b, $y, $signA, $expA, $manA, $signB);
my ($expB, $manB, $signY, $expY, $manY, @sum, $signAbin, $expAbin, $manAbin);
my ($signBbin, $expBbin, $manBbin, $signYbin, $expYbin, $manYbin, $signObin);
my ($expObin, $manObin, $i, $a_shift, $b_shift, $signAshift, $expAshift);
my ($manAshift, $signBshift, $expBshift, $manBshift);
```

```
$a_shift = "0000000000000000";
$b_shift = "0000000000000000";

$fnameIn  = "tv16";
$fnameOut = "tv16reg";
open INFILE,  "<$fnameIn"  || die ("Can't read $fnameIn\n");
open OUTFILE, ">$fnameOut" || die ("Can't write $fnameOut\n");
while (<INFILE>) {
  chomp;
  $line = $_;
  $a = substr($line, 0, 16);
  $b = substr($line, 17, 16);
  $y = substr($line, 34, 16);
  $signA = &b2d(substr($a, 0, 1), 1);
  $expA  = &b2d(substr($a, 1, 5), 5);
  $manA  = &b2d(substr($a, 6, 10), 10);
  $signB = &b2d(substr($b, 0, 1), 1);
  $expB  = &b2d(substr($b, 1, 5), 5);
  $manB  = &b2d(substr($b, 6, 10), 10);
  $signY = &b2d(substr($y, 0, 1), 1);
  $expY  = &b2d(substr($y, 1, 5), 5);
  $manY  = &b2d(substr($y, 6, 10), 10);

  @sum = &fpSum($signA, $expA, $manA, $signB, $expB, $manB);

  $signAbin = &d2b($signA, 1);
  $expAbin  = &d2b($expA, 5);
  $manAbin  = &d2b($manA, 10);
  $signBbin = &d2b($signB, 1);
  $expBbin  = &d2b($expB, 5);
  $manBbin  = &d2b($manB, 10);
  $signYbin = &d2b($signY, 1);
  $expYbin  = &d2b($expY, 5);
  $manYbin  = &d2b($manY, 10);
  $signObin = &d2b($sum[0], 1);
  $expObin  = &d2b($sum[1], 5);
  $manObin  = &d2b($sum[2], 10);

  if (($signYbin != $signObin) || ($expYbin != $expObin) ||
      ($manYbin != $manObin)) {
    print "Error!\n";
    print "Original:   signY: $signY. expY: $expY. manY: $manY\n";
    print "Calculated: signY: " . $sum[0] . ". expY: " . $sum[1] . ". manY: "
      . $sum[2] . "\n";
  }

  for ($i=0; $i<16; $i++) {
    $a_shift = substr($a_shift, 1, 15) . substr($a, $i, 1);
    $b_shift = substr($b_shift, 1, 15) . substr($b, $i, 1);
    $signAshift = &b2d(substr($a_shift, 0, 1), 1);
    $expAshift  = &b2d(substr($a_shift, 1, 5), 5);
    $manAshift  = &b2d(substr($a_shift, 6, 10), 10);
    $signBshift = &b2d(substr($b_shift, 0, 1), 1);
    $expBshift  = &b2d(substr($b_shift, 1, 5), 5);
    $manBshift  = &b2d(substr($b_shift, 6, 10), 10);
    @sum = &fpSum($signAshift, $expAshift, $manAshift, $signBshift,
      $expBshift, $manBshift);
    print OUTFILE substr($a, $i, 1) . "_" . substr($b, $i, 1) . "_" .
```

```
      &d2b($sum[0], 1) . &d2b($sum[1], 5) . &d2b($sum[2], 10) . "\n";
  }
}
close(INFILE);
close(OUTFILE);

sub d2b {
  my $val  = $_[0];
  my $bits = $_[1];
  my $str = "";
  for (my $i=0; $i<$bits; $i++) {
    $str = ($val % 2) . $str;
    $val = int($val/2);
  }
  return $str;
}

sub b2d {
  my $val  = $_[0];
  my $bits = $_[1];
  my $out = 0;
  my $str;
  for (my $i=0; $i<$bits; $i++) {
    $str = substr($val, $i, 1);
    $out = $out + $str*2**($bits-1-$i);
  }
  return $out;
}

sub fpSum {
  my ($ans, $signOut, $expOut, $manOut, $calc);

  my $signA = $_[0];
  my $expA  = $_[1] - 15;
  my $manA  = $_[2] + 2**$man_bits;
  my $signB = $_[3];
  my $expB  = $_[4] - 15;
  my $manB  = $_[5] + 2**$man_bits;

  my $zero       = ($expA == $expB) && ($manA == $manB) && ($signA != $signB);
  my $AIsInf     = ($expA + 15 == 31) && ($manA - 2**$man_bits == 0);
  my $BIsInf     = ($expB + 15 == 31) && ($manB - 2**$man_bits == 0);
  my $AIsNaN     = ($expA + 15 == 31) && ($manA - 2**$man_bits != 0);
  my $BIsNaN     = ($expB + 15 == 31) && ($manB - 2**$man_bits != 0);
  my $AIsDenorm = ($expA + 15 == 0);
  my $BIsDenorm = ($expB + 15 == 0);

  $ans = ((-1)**$signA)*$manA*(2**$expA) + ((-1)**$signB)*$manB*(2**$expB);

  $signOut = ($ans < 0) ? 1 : 0;

  $expOut = 0;
  $manOut = abs($ans);

  if ($ans != 0) {
    while (($manOut >= 2048) || ($manOut < 1024)) {
      if ($manOut >= 2048) {
        $manOut = $manOut/2;
        $expOut++;
      }
      elsif ($manOut < 1024) {
        $manOut = $manOut*2;
        $expOut--;
      }
    }
  }

  my $outIsInf    = ($expOut + 15 >= 31);
  my $outIsDenorm = ($expOut + 15 <= 0);

  my $NaN = ($AIsInf && $BIsInf && ($signA != $signB)) || $AIsNaN || $BIsNaN;
  my $overflow = ($AIsInf || $BIsInf || $outIsInf) && !$zero;
  my $underflow = $AIsDenorm || $BIsDenorm || $outIsDenorm || $zero;

  if ($NaN) {
    $signOut = 0;
    $expOut  = 31;
    $manOut  = 1023;
  }
  elsif ($overflow) {
    $expOut = 31;
    $manOut = 0;
  }
  elsif ($underflow) {
    $signOut = 0;
    $expOut  = 0;
    $manOut  = 0;
  }

  $calc = ((-1)**$signOut)*$manOut*(2**$expOut);

  if (!$NaN && !$overflow && !$underflow) {
    $expOut = $expOut + 15;
    $manOut = $manOut - 2**$man_bits;
  }

  if (($calc != $ans) && !$NaN && !$overflow && !$underflow) {
    print "Error! Calculated answer: $calc vs Initial answer: $ans\n";
    print "signA: $signA. expA: " . $expA . ". manA: " . $manA . "\n";
    print "signB: $signB. expB: " . $expB . ". manB: " . $manB . "\n";
  }

  return ($signOut, $expOut, $manOut);
}
```

Custom Schematics

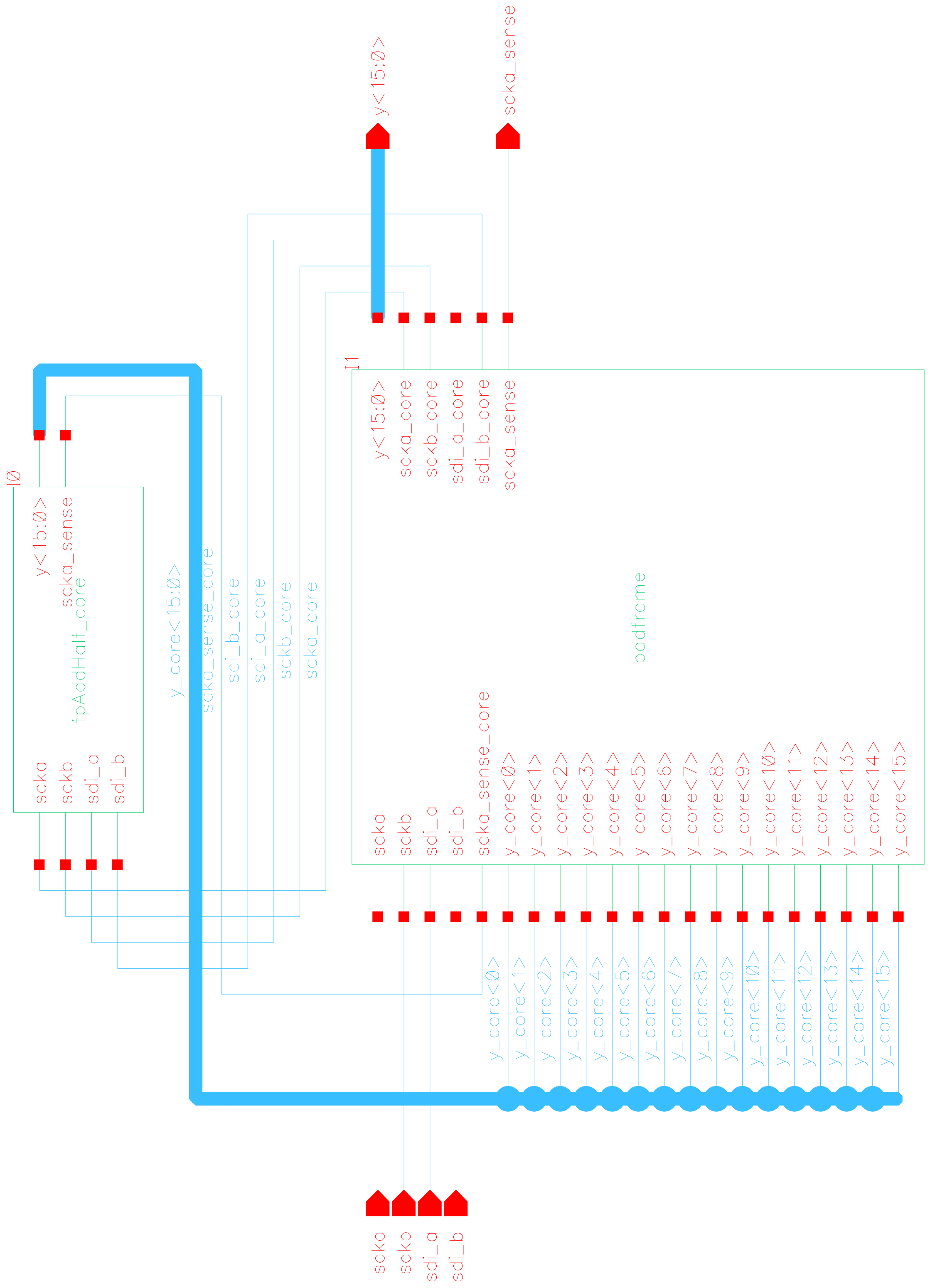


Figure 3: `fpAddHalf_chip` Schematic

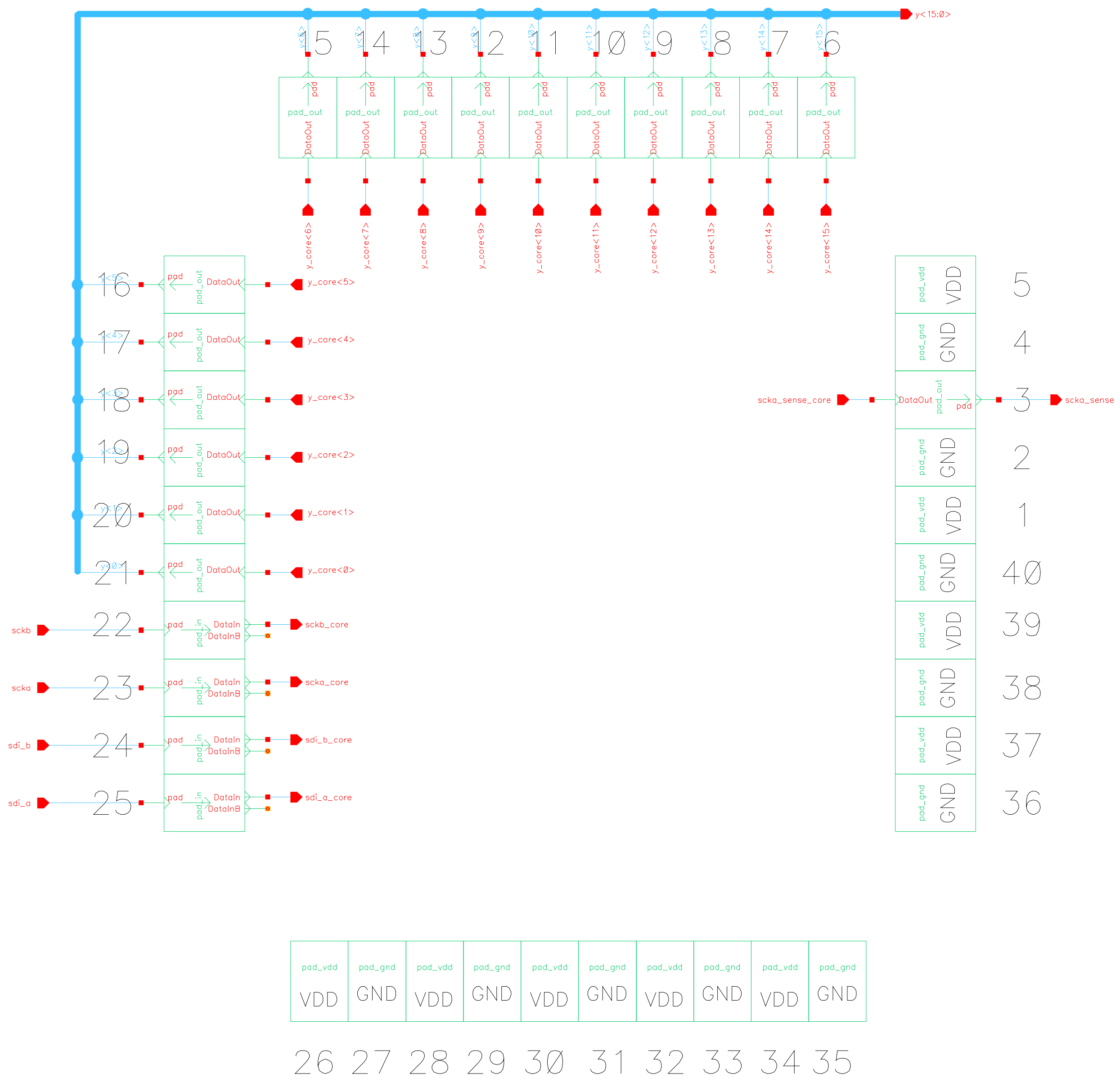


Figure 4: `padframe` Schematic

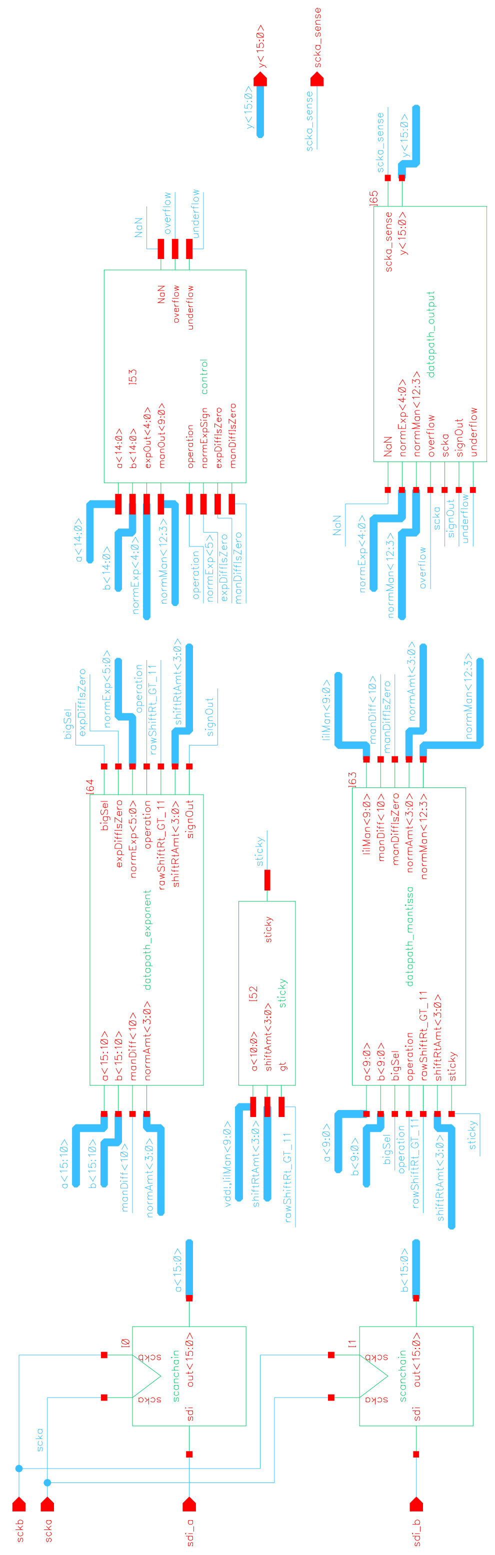


Figure 5: `fpAddHalf_core` Schematic

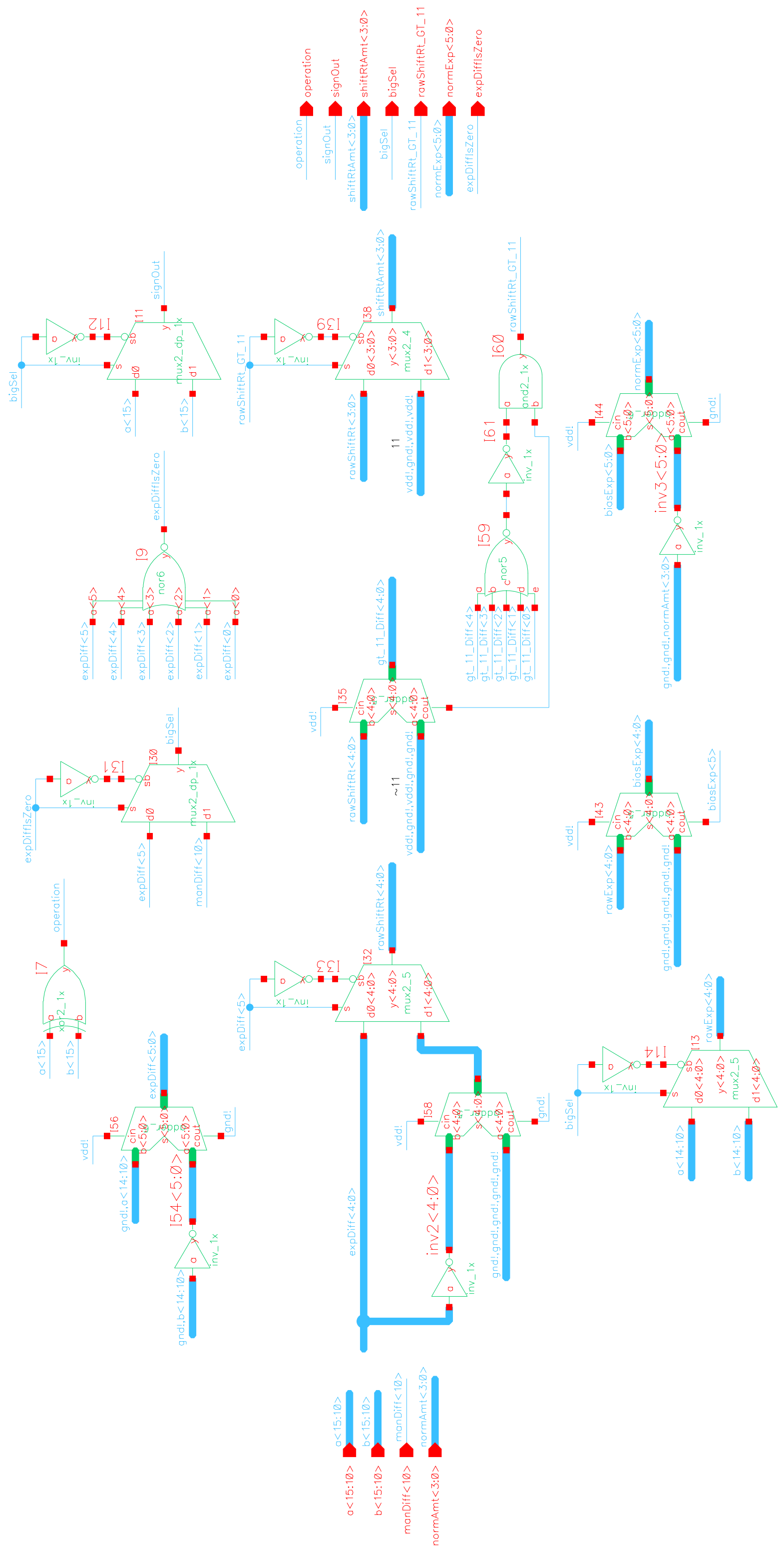


Figure 6: `datapath_exponent` Schematic

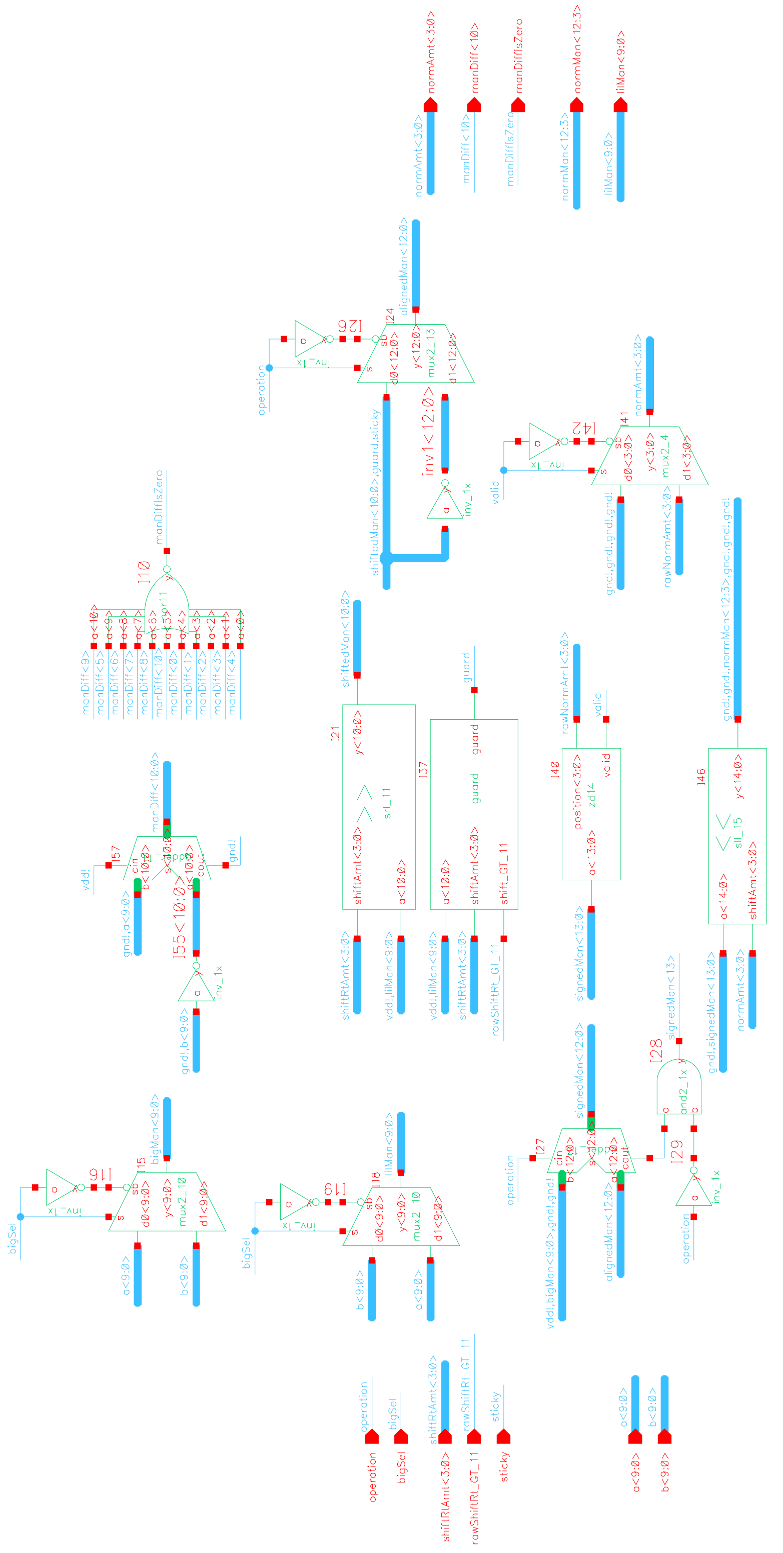


Figure 7: `datapath_mantissa` Schematic

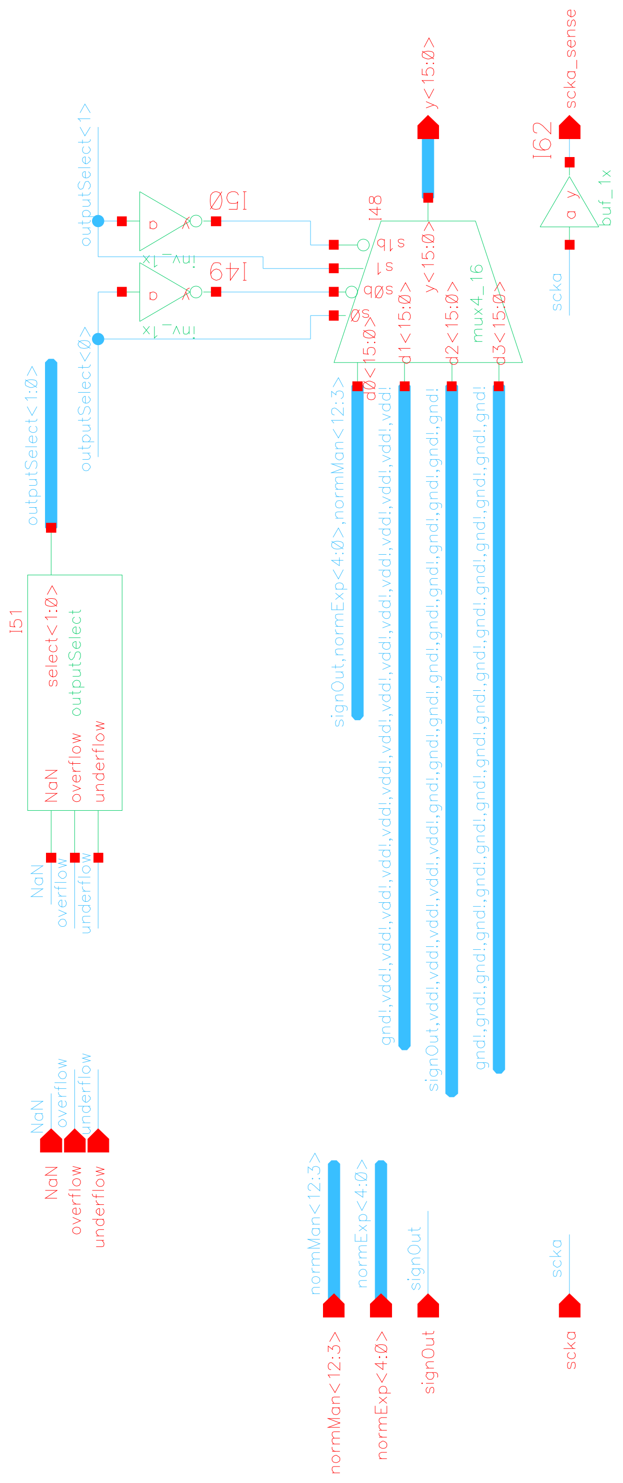

outputSelect<1:0>
outputSelect<0>
outputSelect<1>
I51
select<1:0>
outputSelect
NaN
overflow
underflow
inv_1x
I49
I50
I48
mux4_16
s0
s0b
s1
s1b
d0<15:0>
d1<15:0>
d2<15:0>
d3<15:0>
y<15:0>
signOut,normExp<4:0>,normMan<12:3>
gnd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!,vdd!
signOut,vdd!,vdd!,vdd!,vdd!,vdd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!
gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!,gnd!
I62
buf_1x
scka_sense
scka
normMan<12:3>
normExp<4:0>
signOut

Figure 8: `datapath_output` Schematic

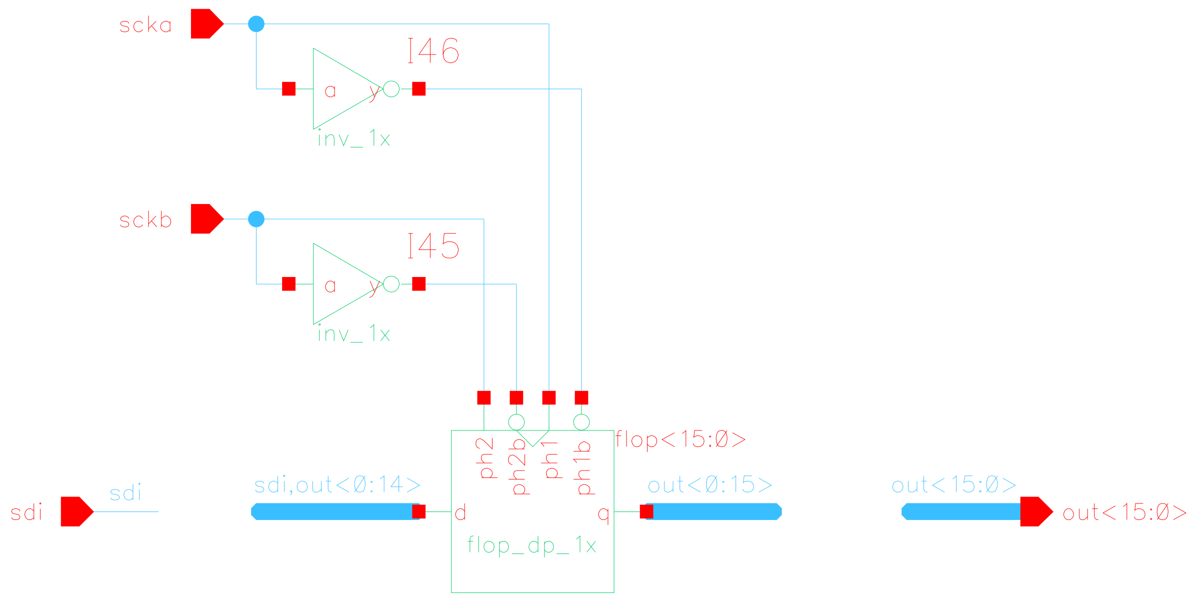


Figure 9: `scanchain` Schematic

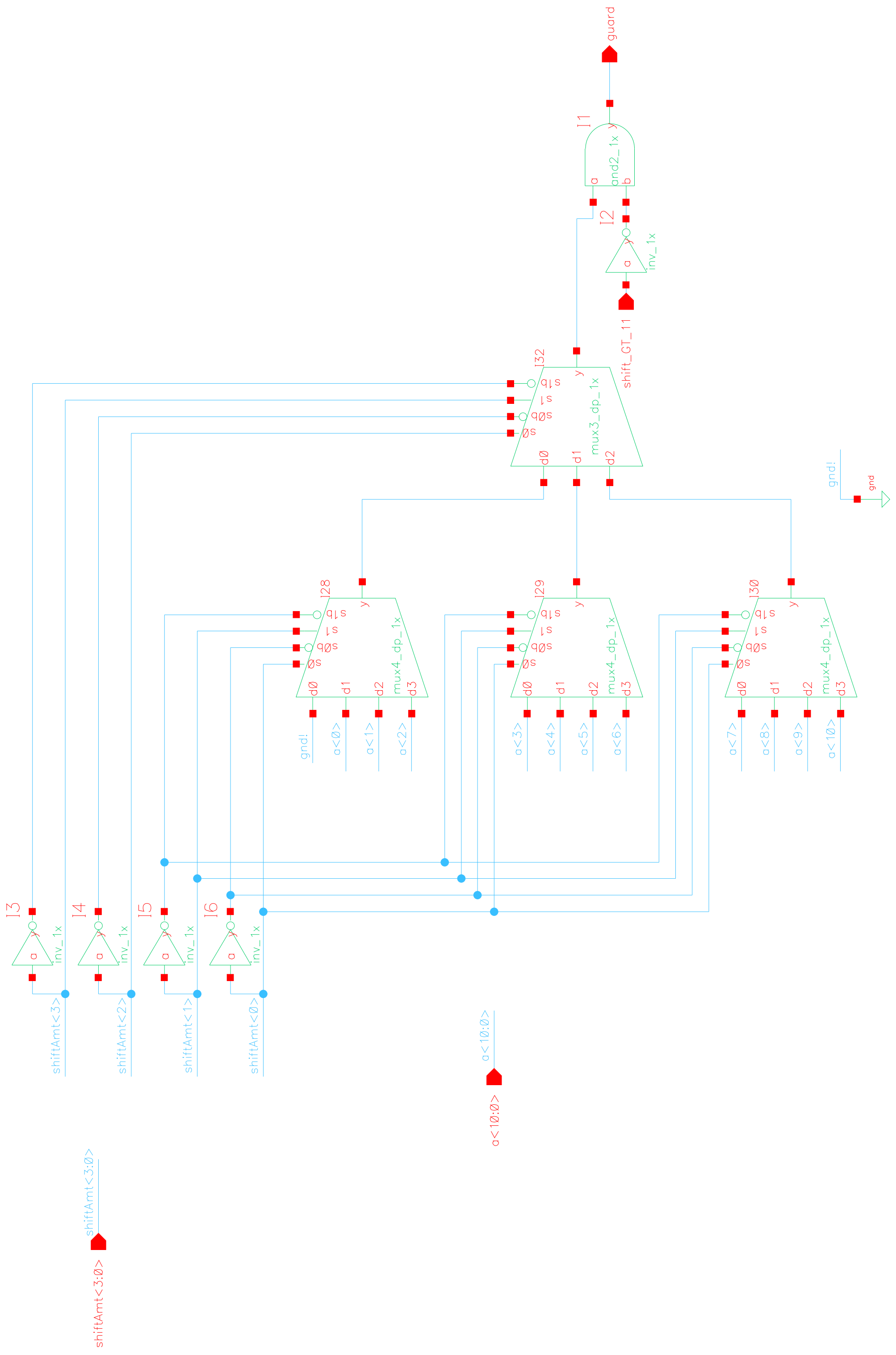


Figure 10: `guard` Schematic

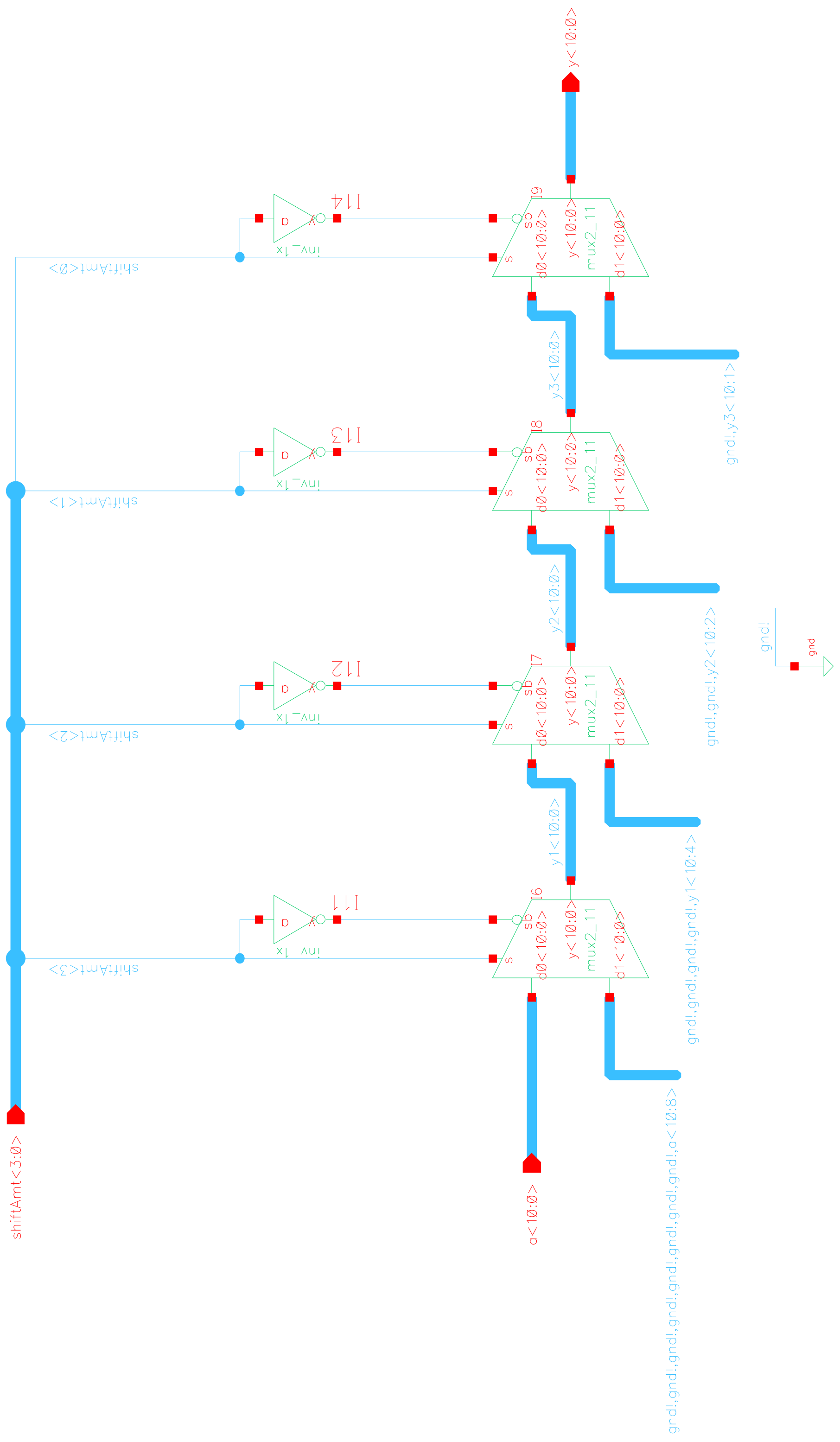


Figure 11: `srl_11` Schematic

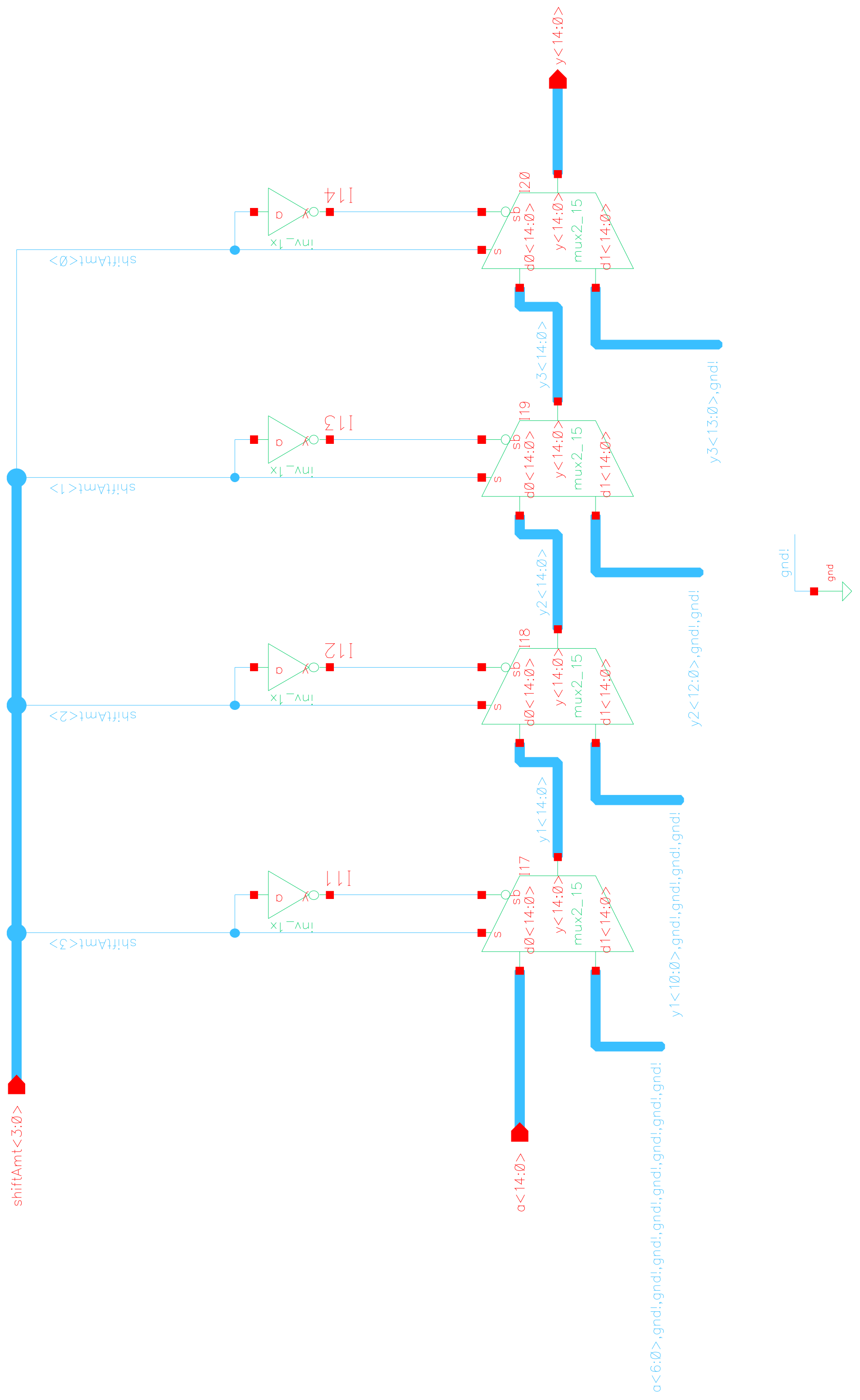


Figure 12: `sll_15` Schematic

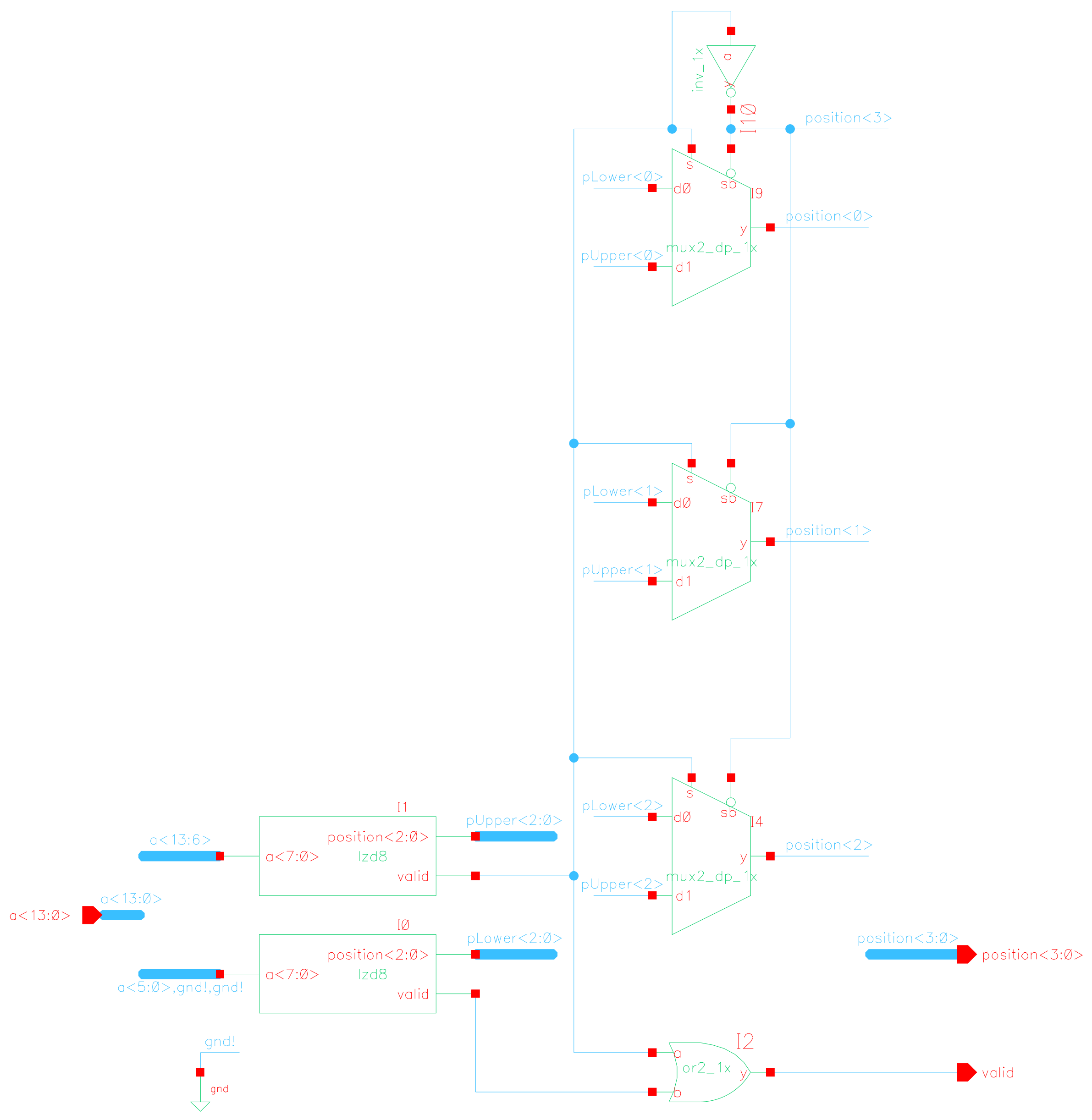


Figure 13: **lzd14** Schematic

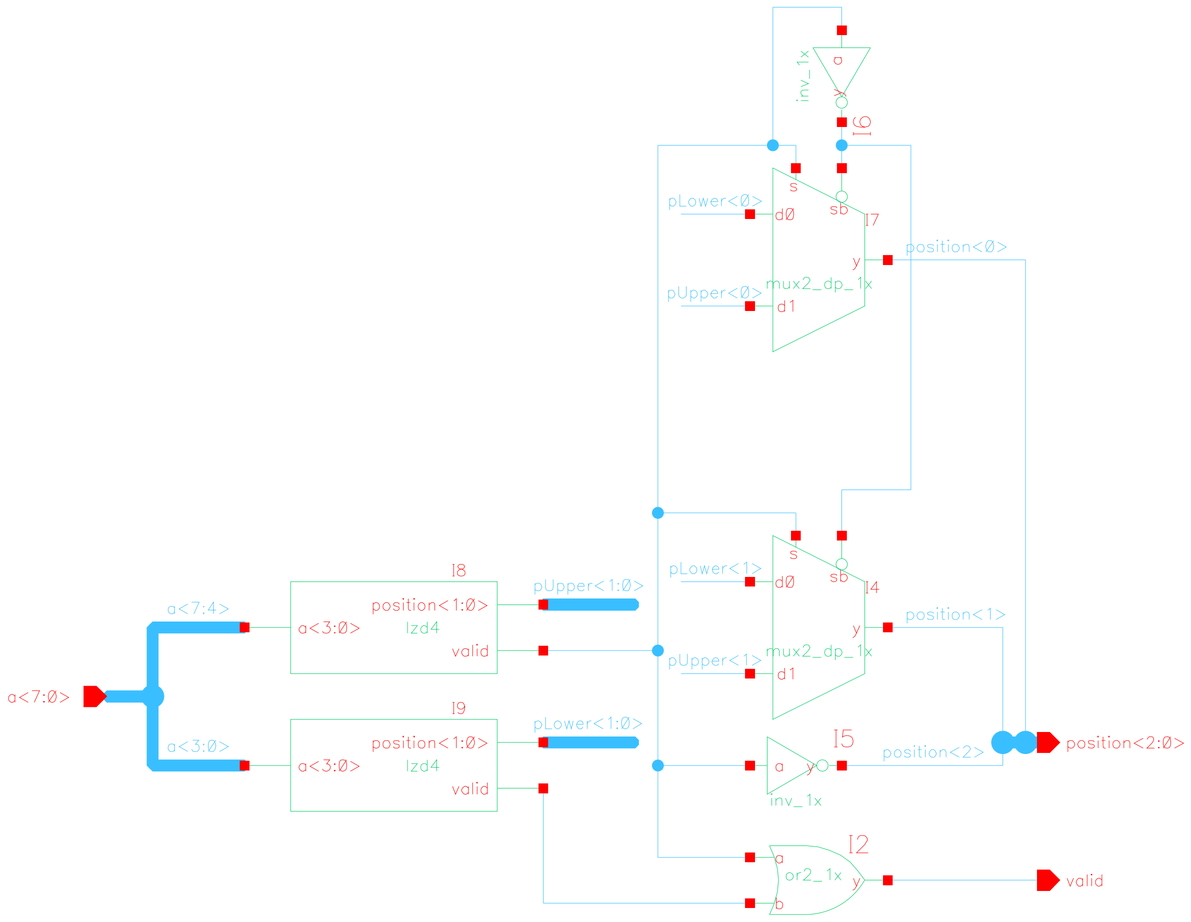


Figure 14: `lzd8` Schematic

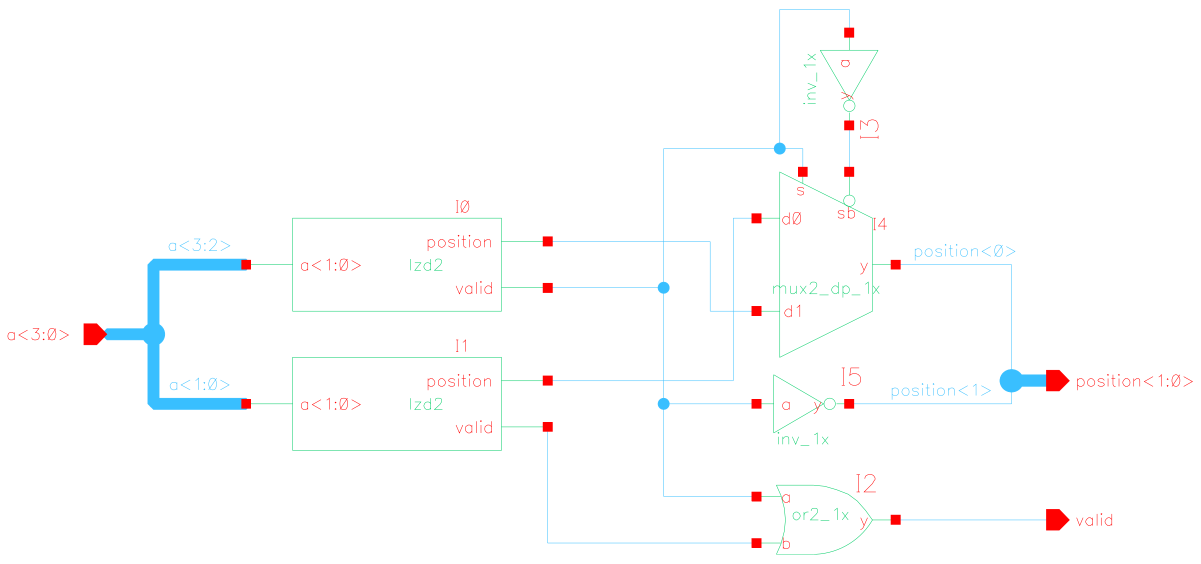


Figure 15: `lzd4` Schematic

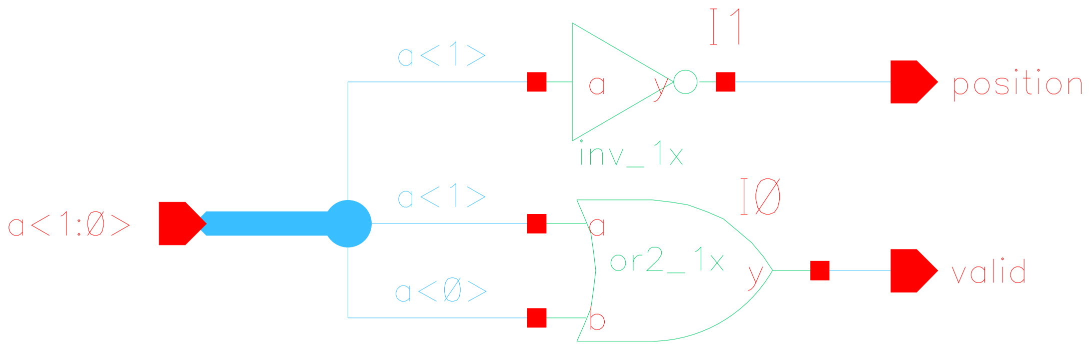


Figure 16: `lzd2` Schematic

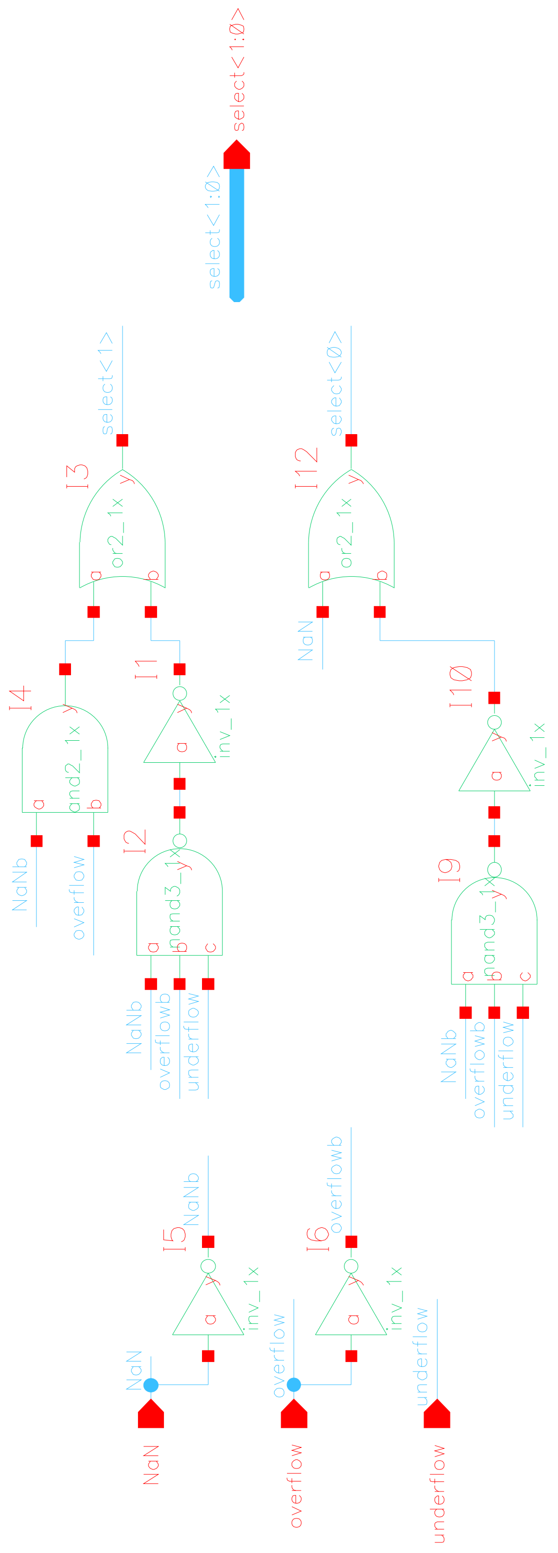


Figure 17: `outputSelect` Schematic

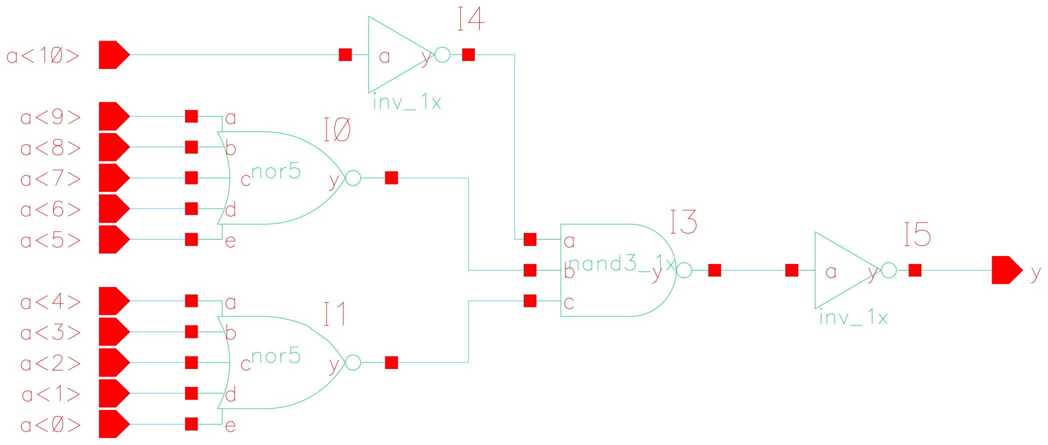


Figure 18: `nor11` Schematic

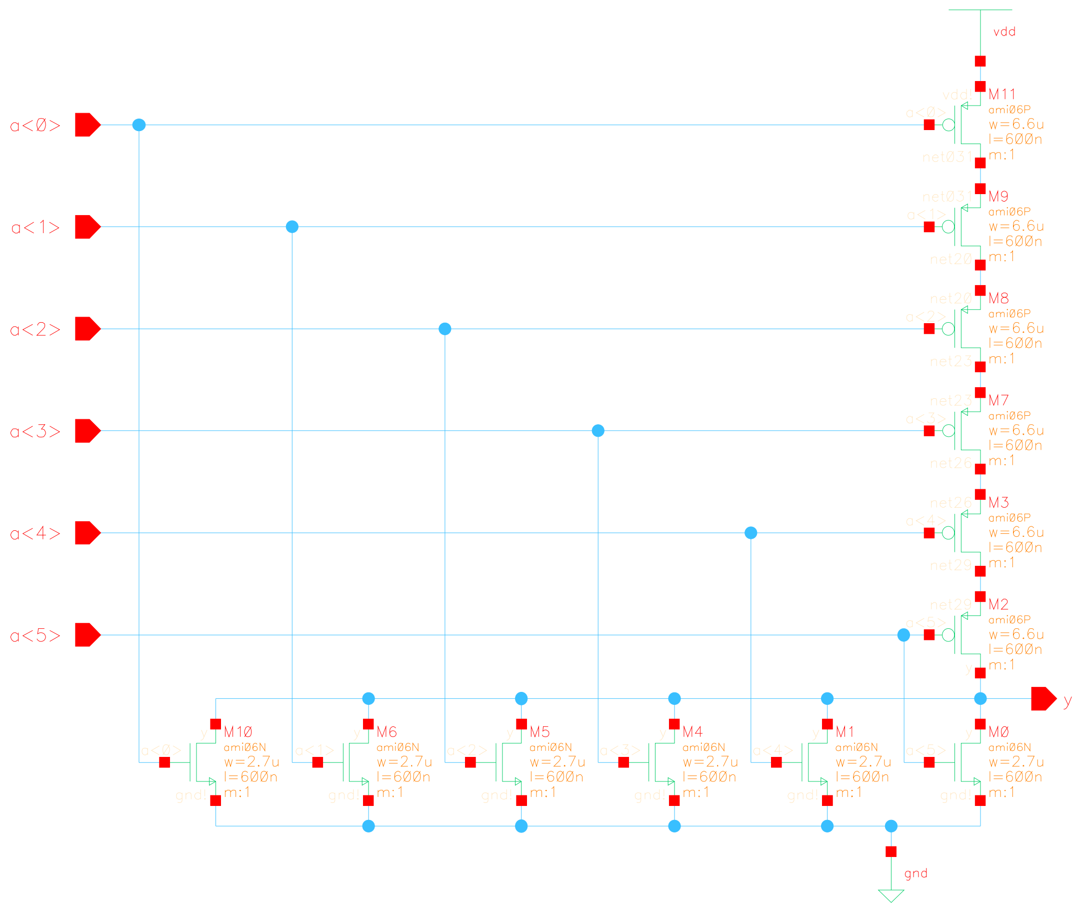


Figure 19: `nor6` Schematic

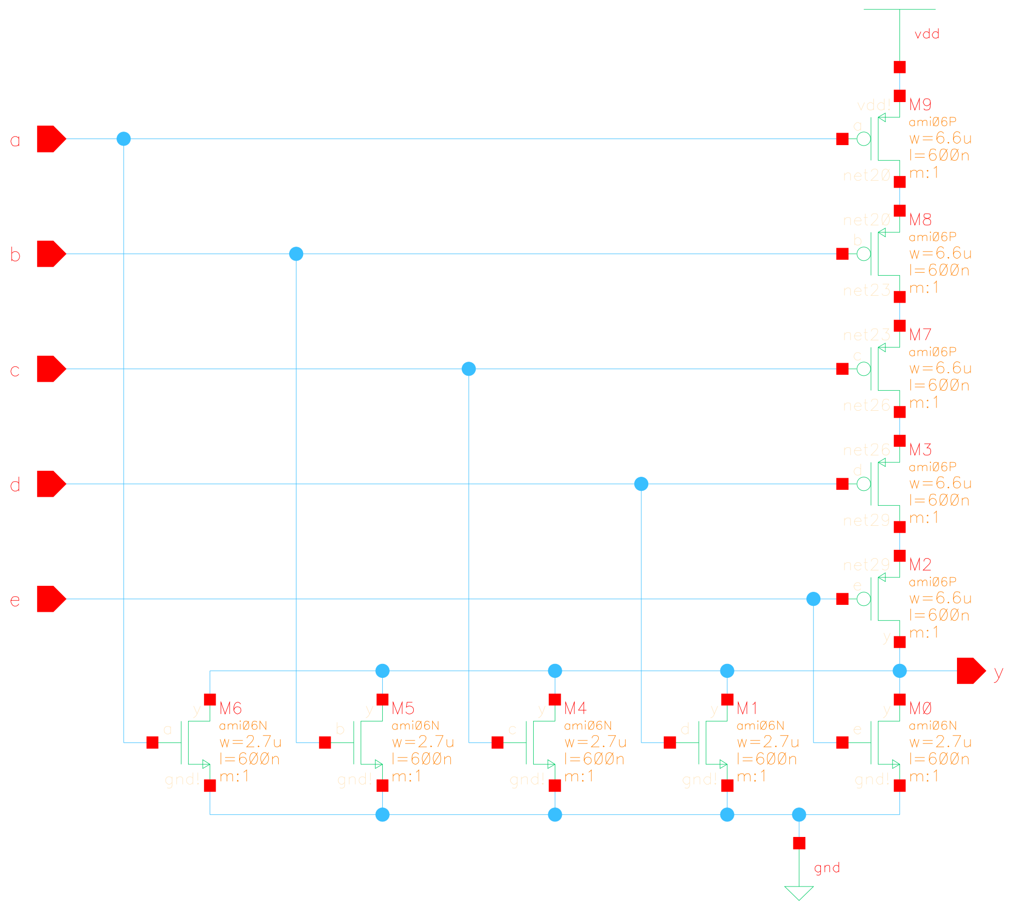


Figure 20: `nor5` Schematic

Custom Layouts

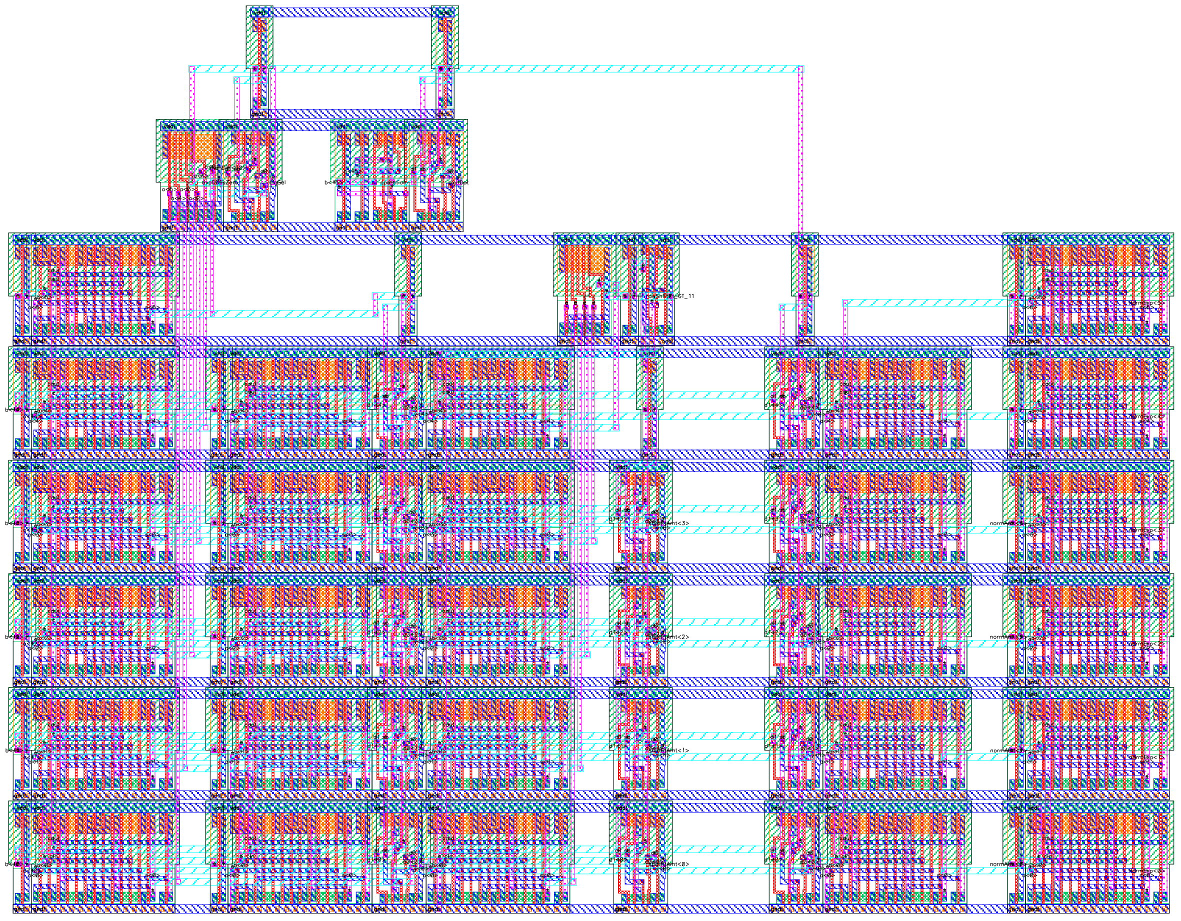


Figure 21: `datapath_exponent` Layout

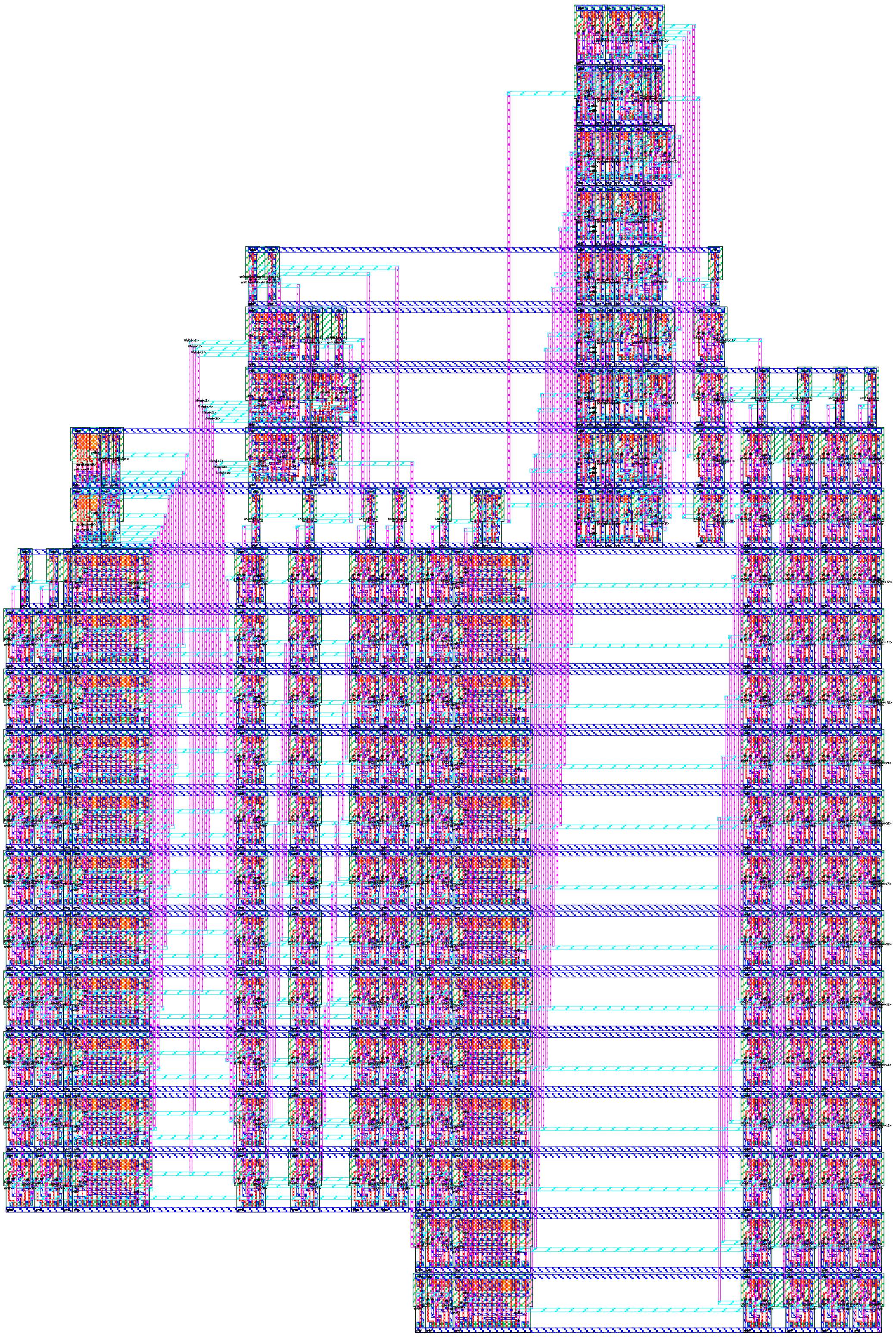


Figure 22: `datapath_mantissa` Layout

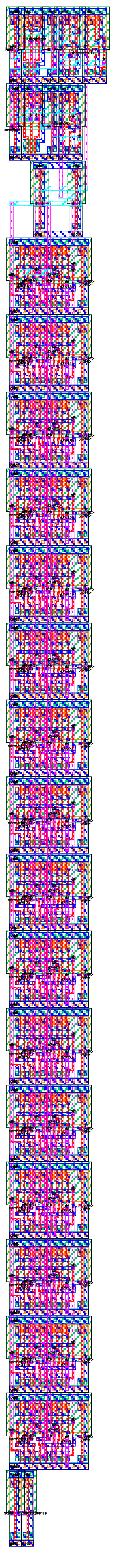


Figure 23: `datapath_output` Layout

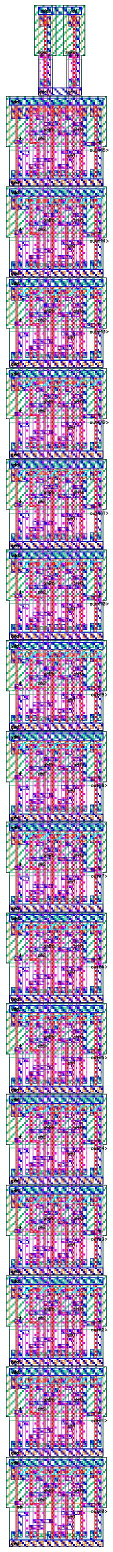


Figure 24: `scanchain` Layout

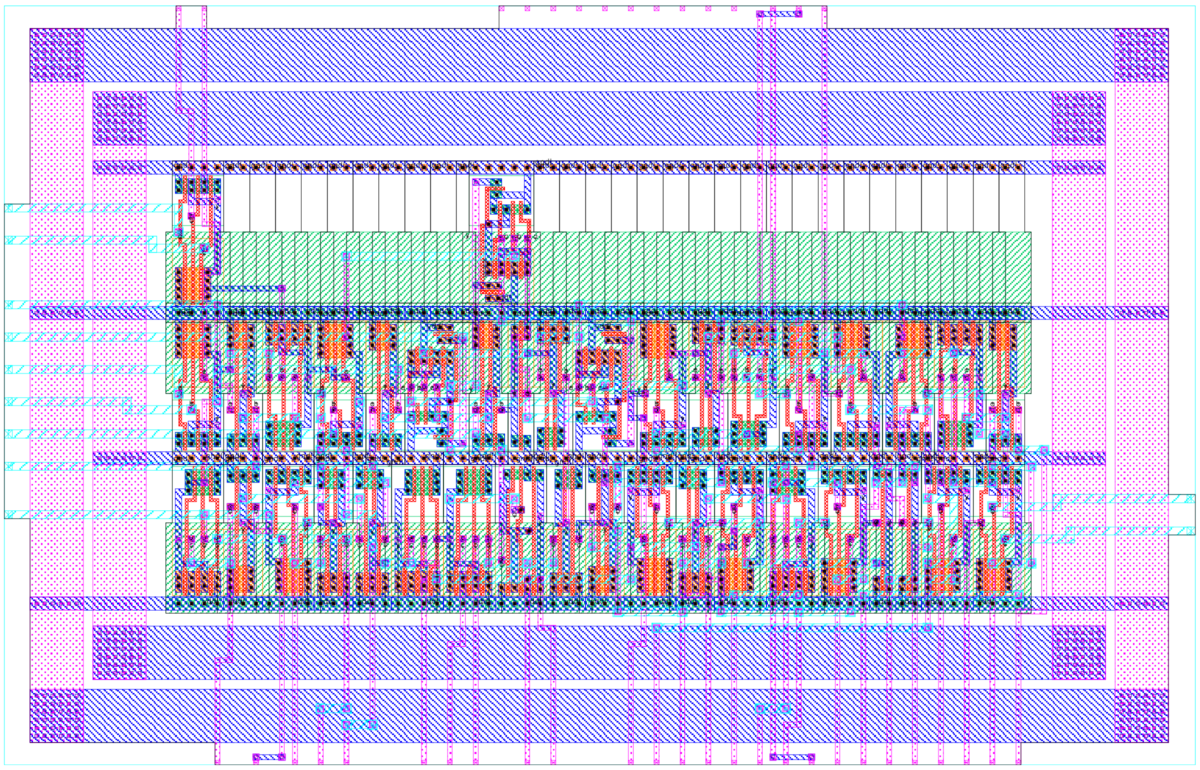


Figure 25: `control` Layout

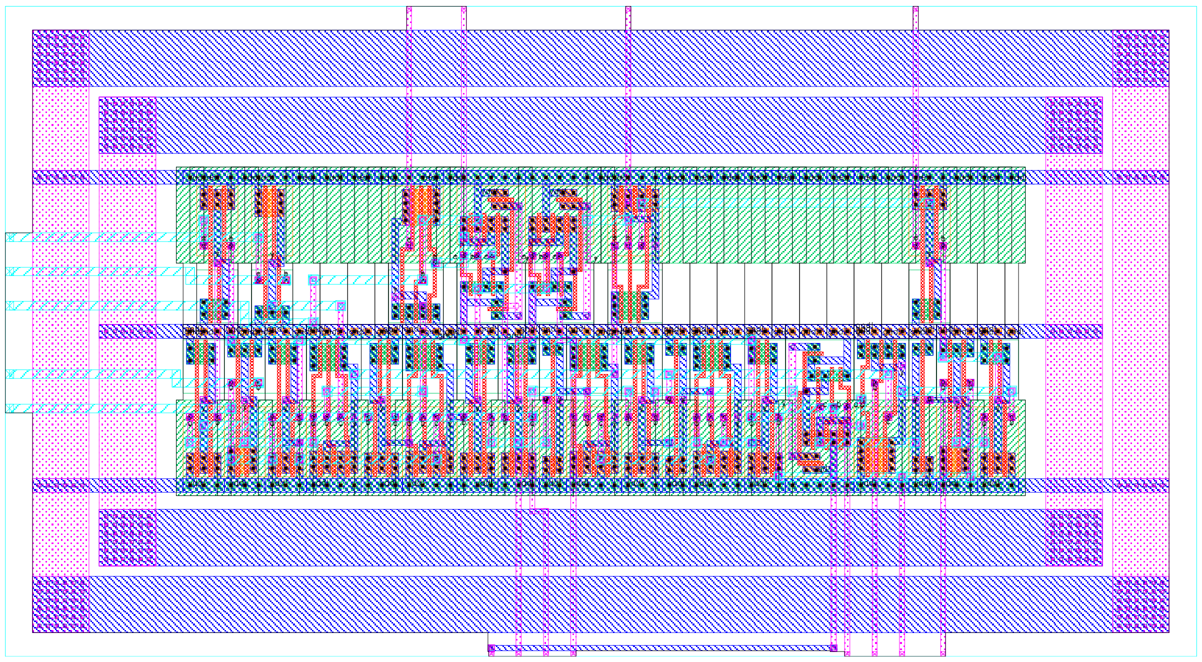


Figure 26: `sticky` Layout

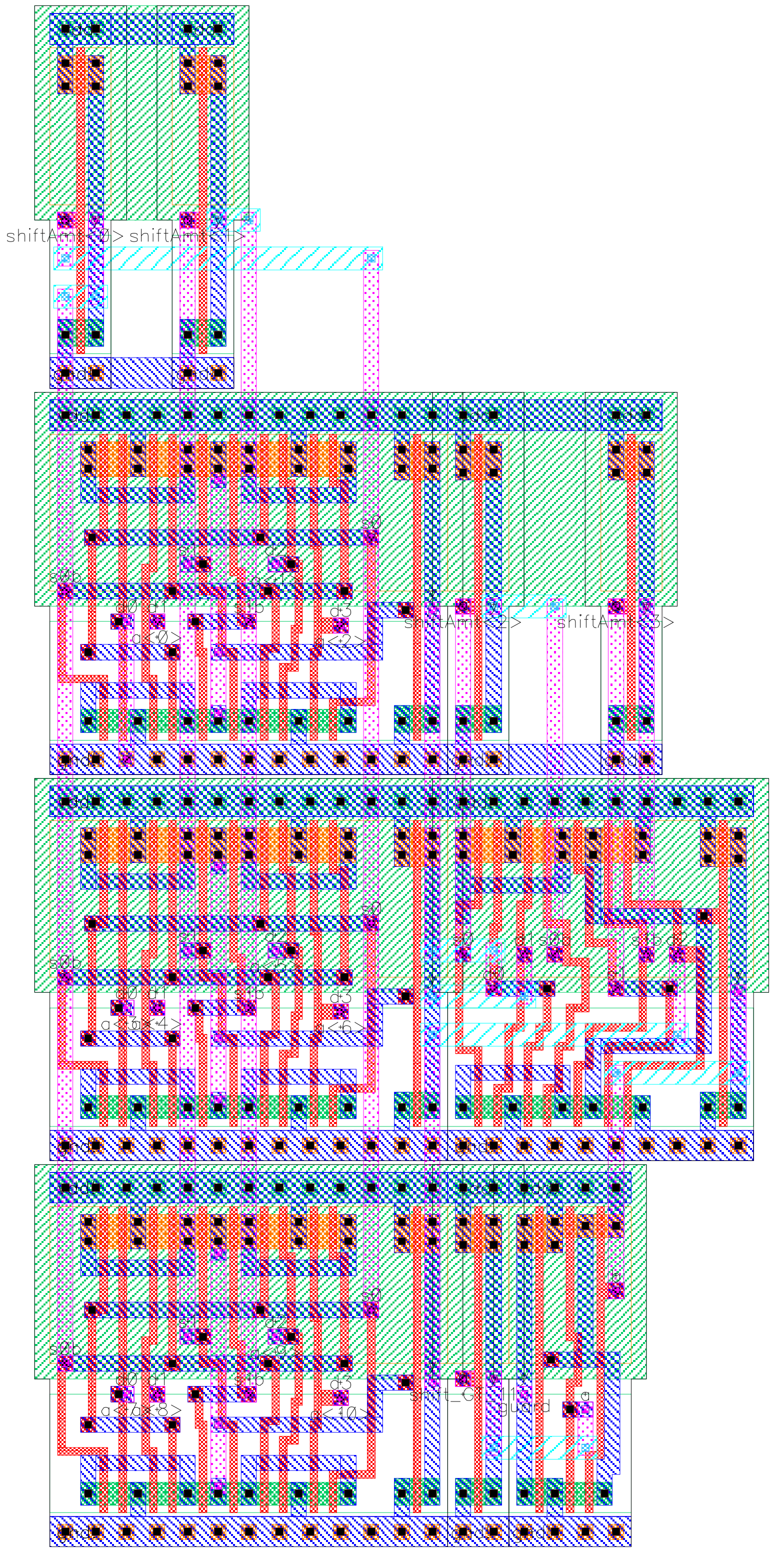


Figure 27: `guard` Layout

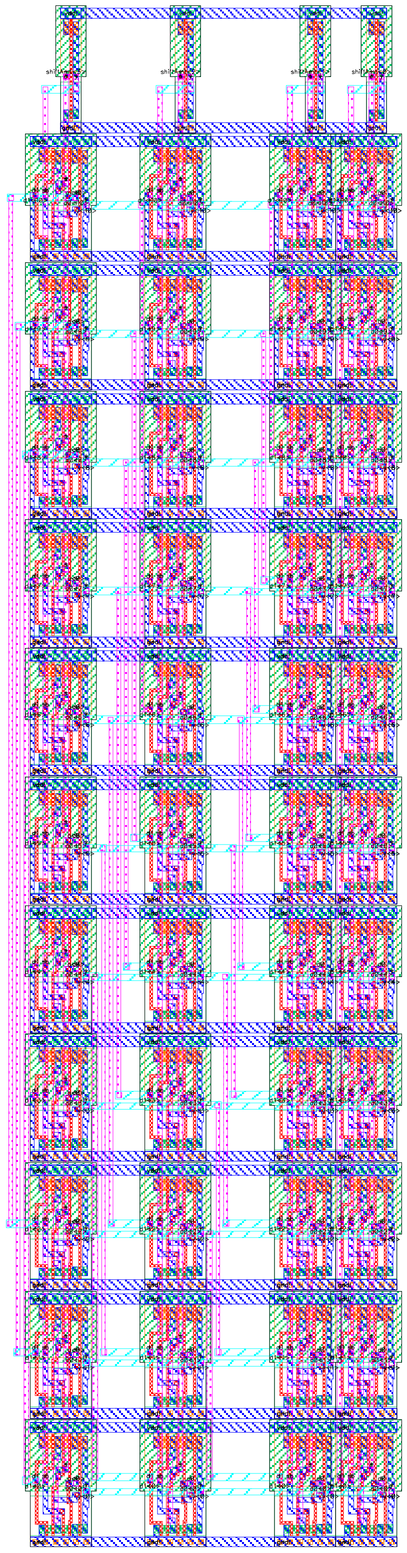


Figure 28: `srl11` Layout

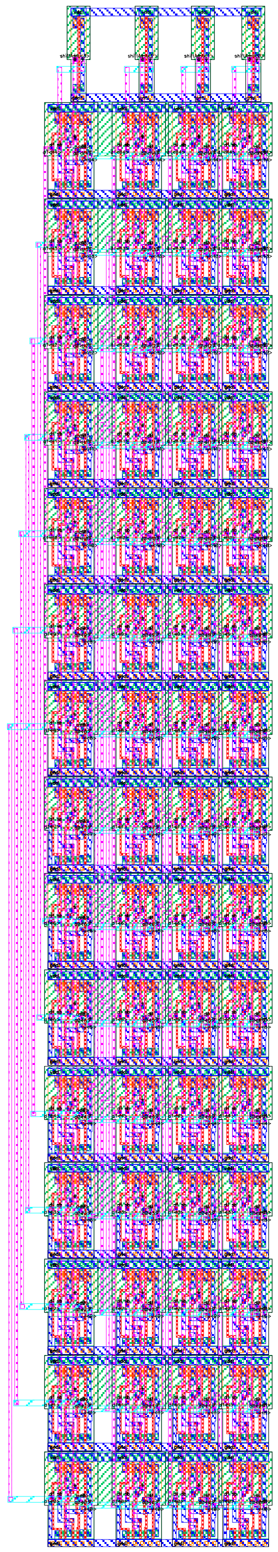


Figure 29: `sll15` Layout

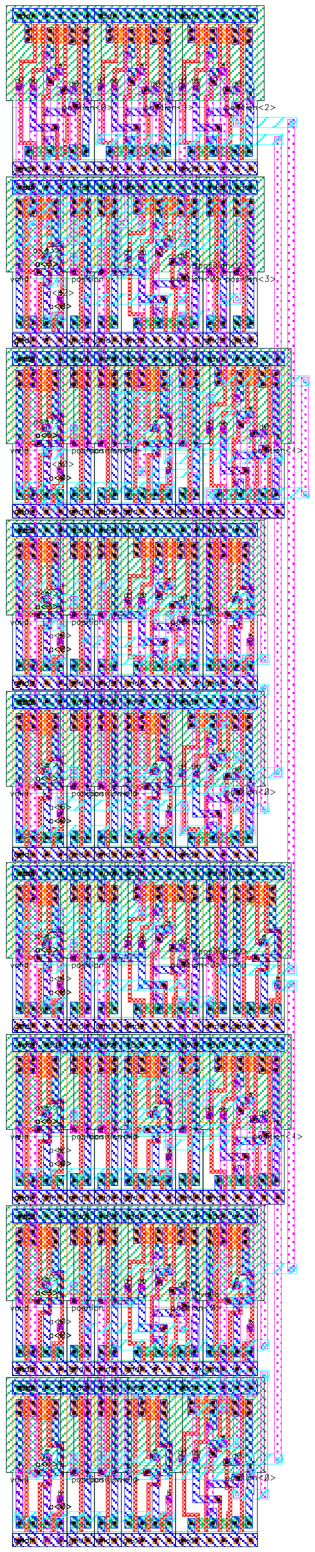


Figure 30: `lzd14` Layout

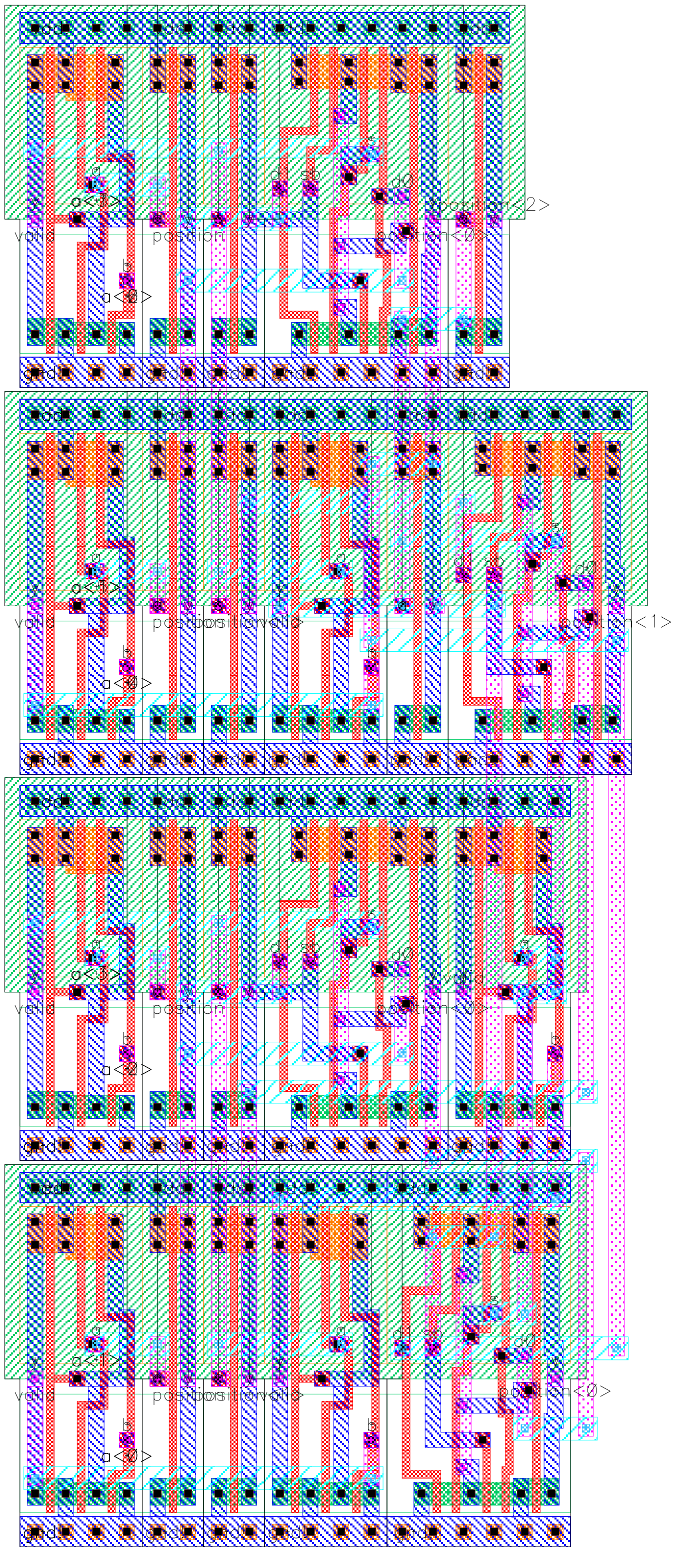


Figure 31: `lzd8` Layout

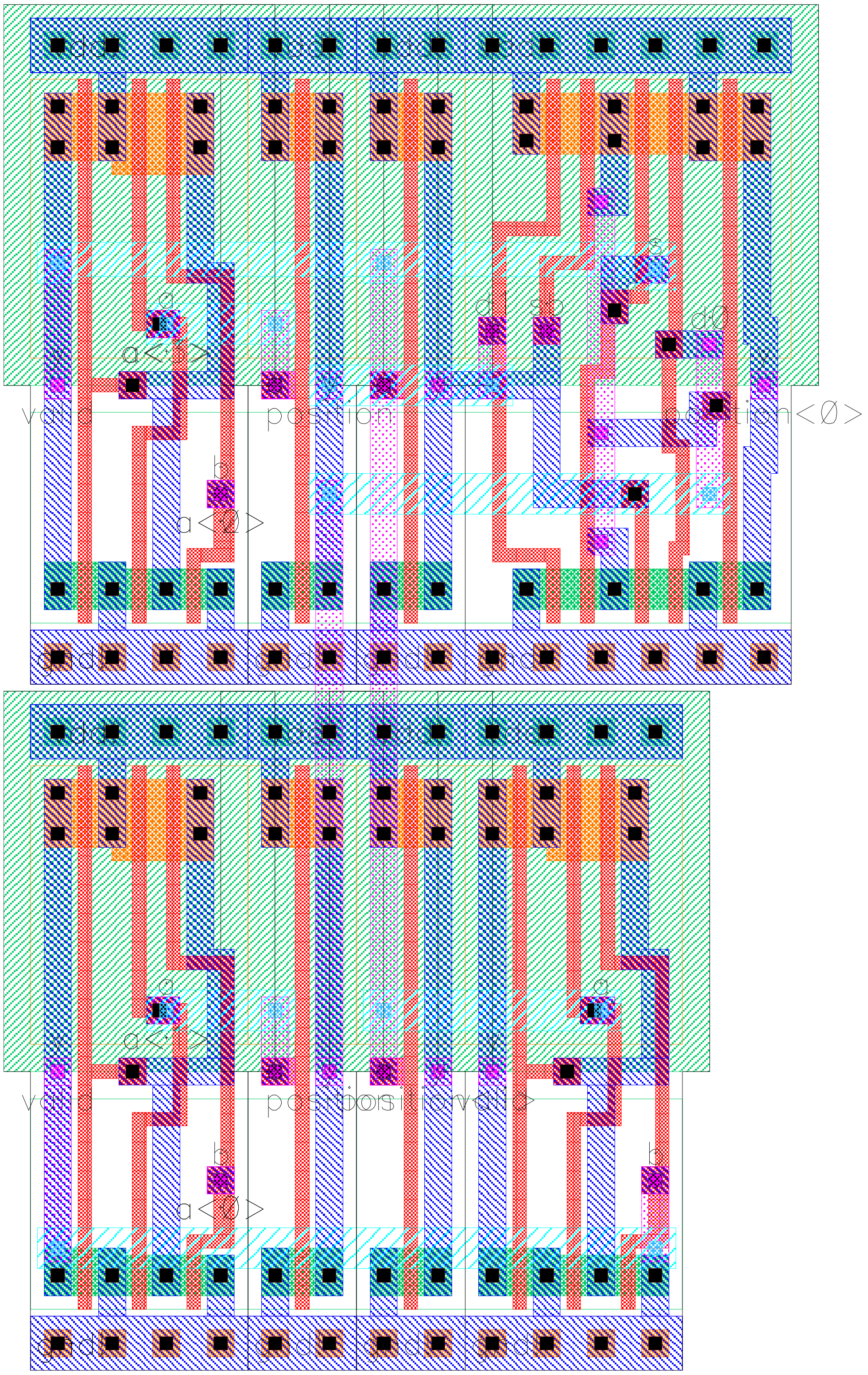


Figure 32: `lzd4` Layout

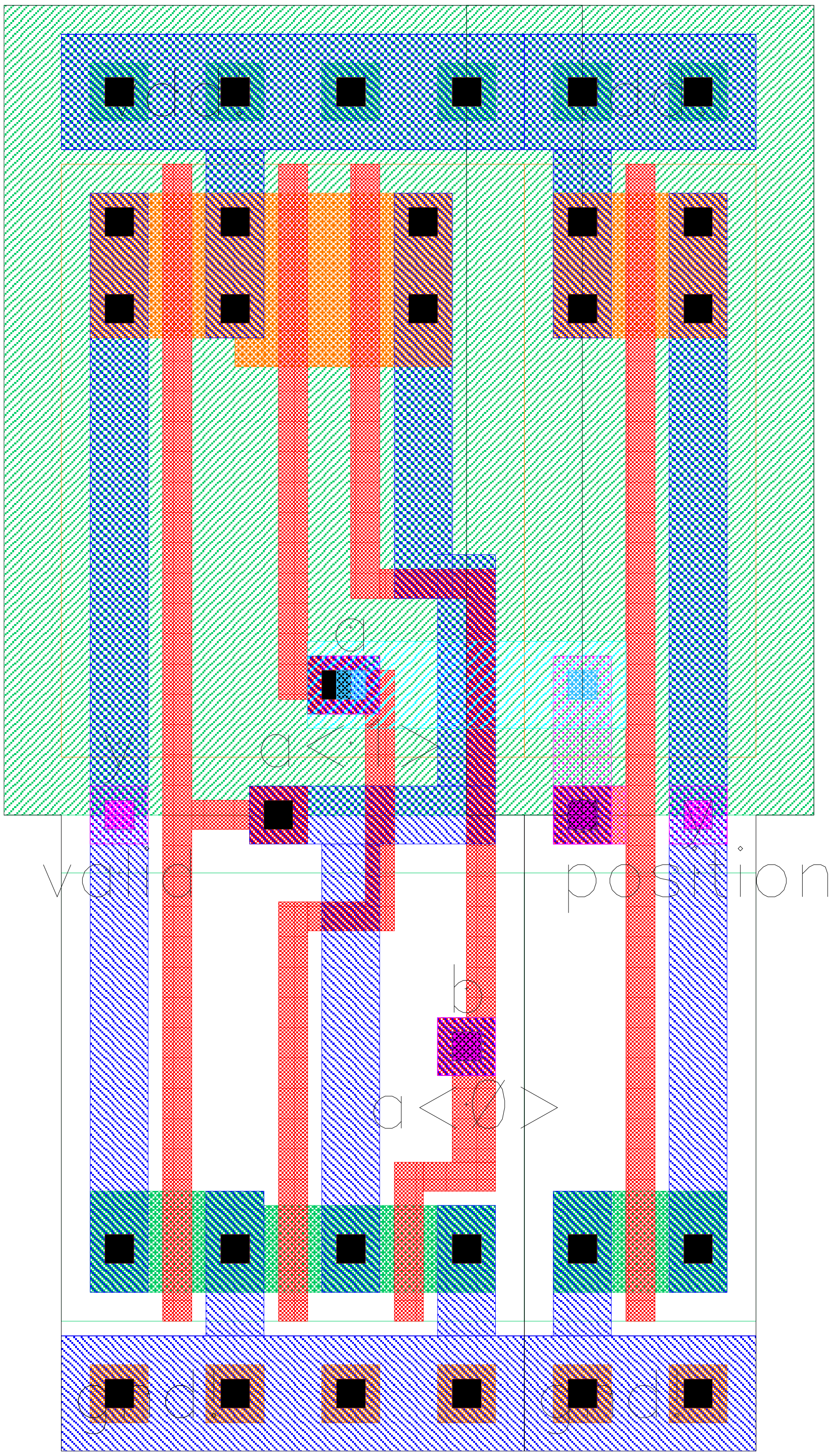


Figure 33: `lzd2` Layout

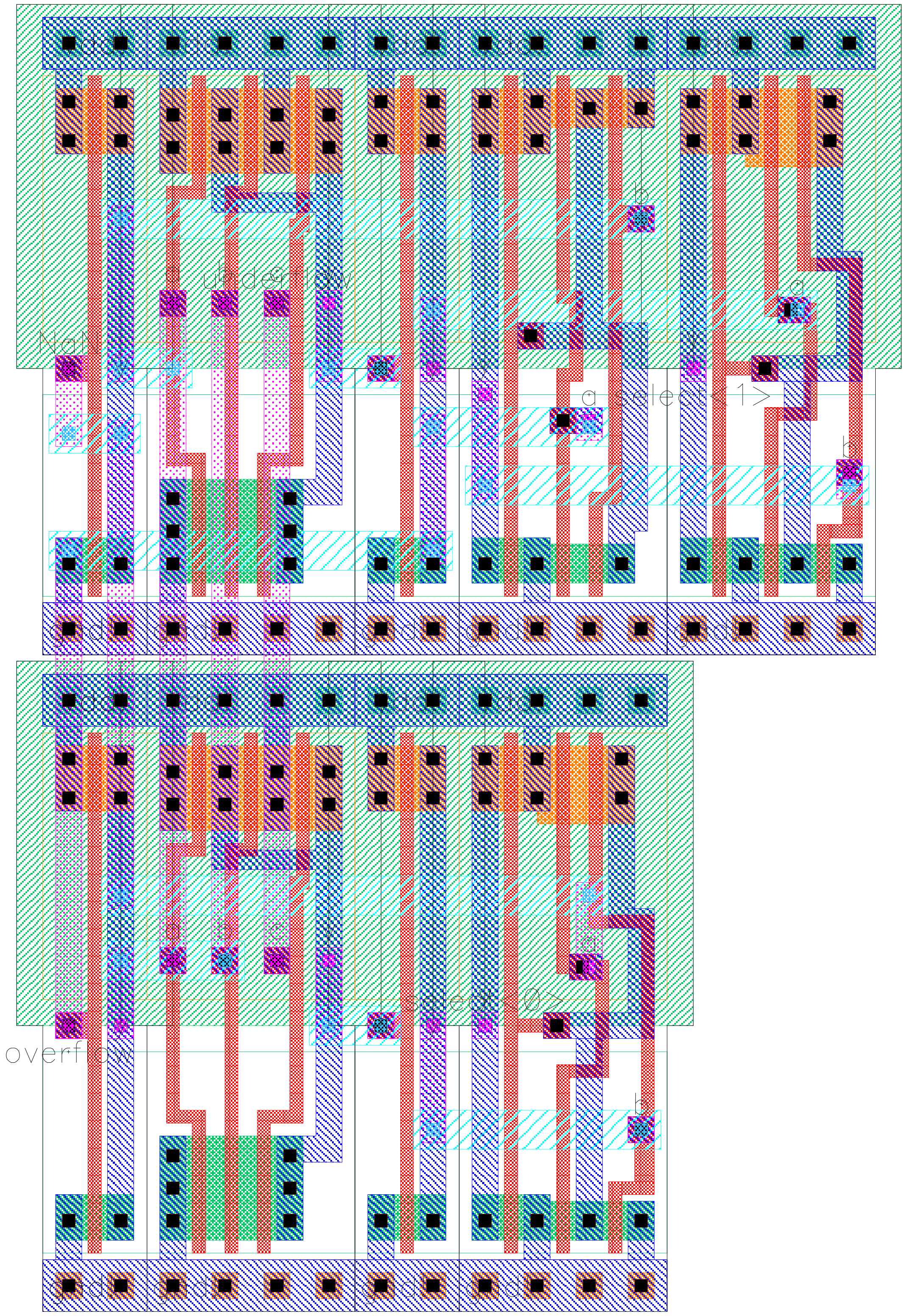


Figure 34: `outputSelect` Layout

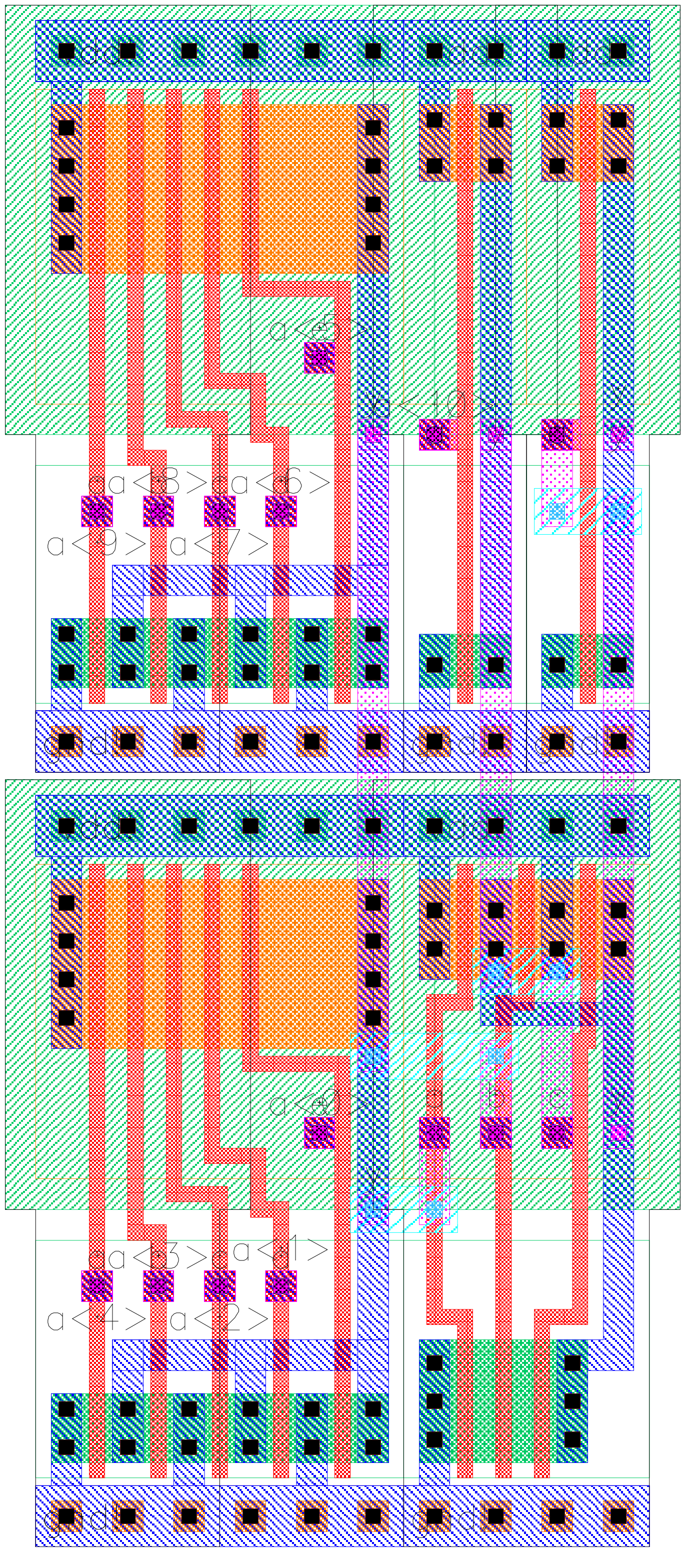


Figure 35: `nor11` Layout

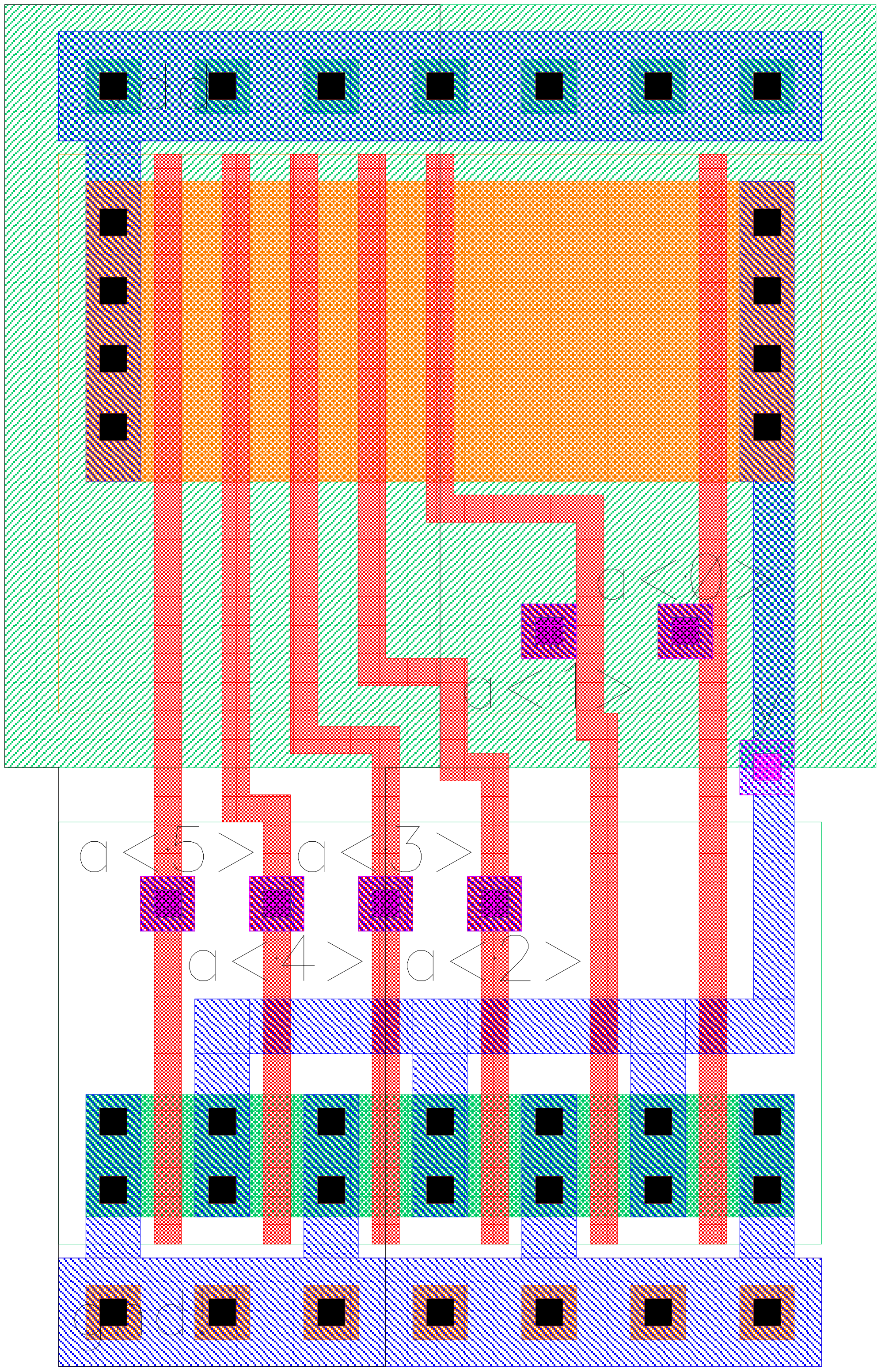


Figure 36: `nor6` Layout

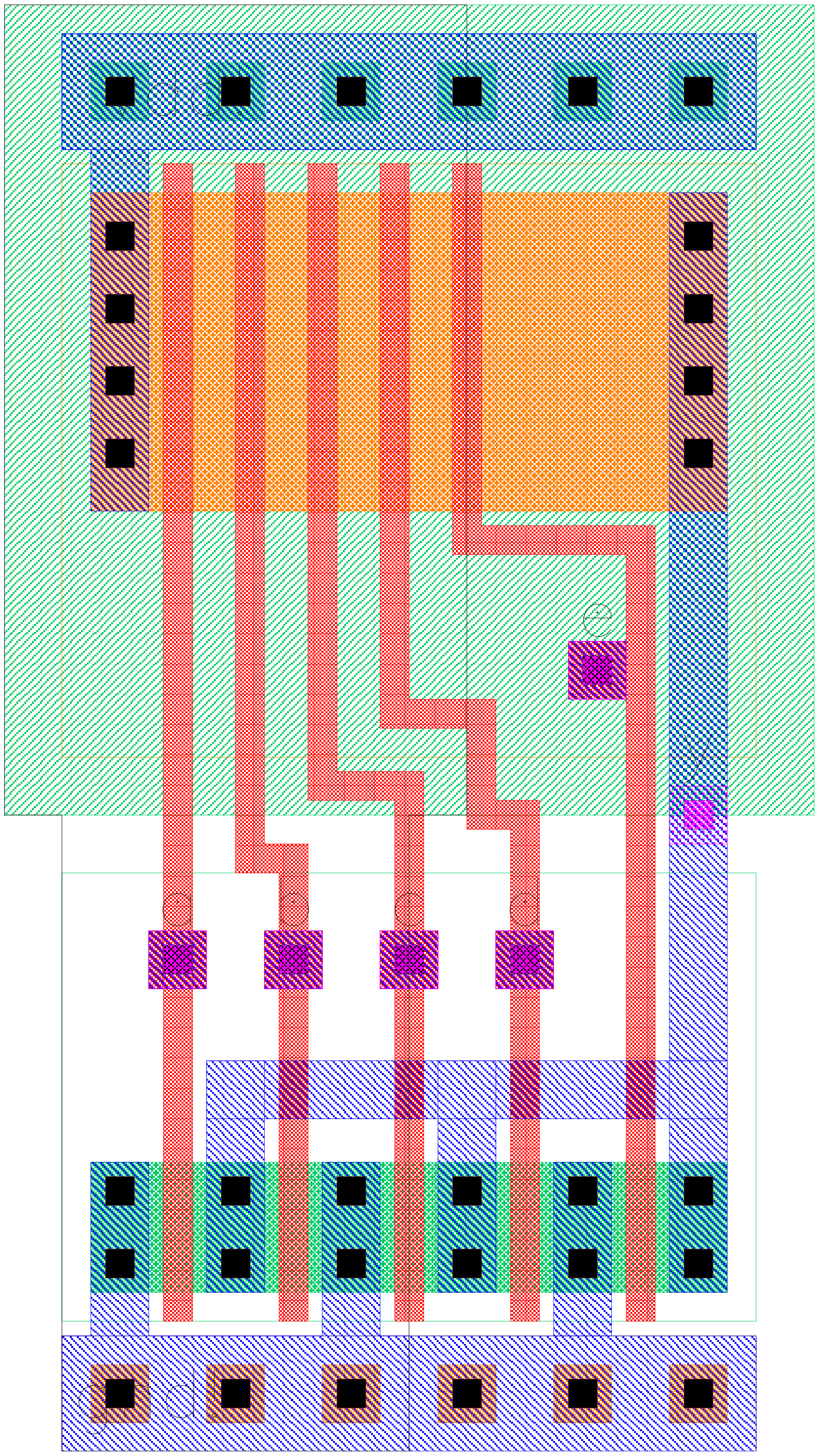


Figure 37: `nor5` Layout