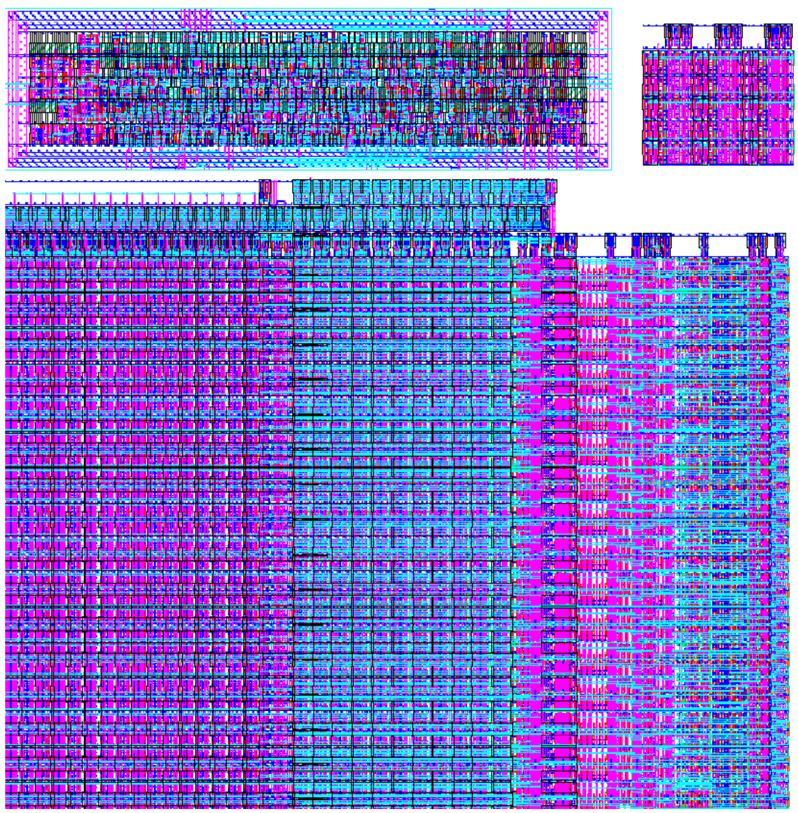
SHA-1 Cryptographic Hash Unit

Kyle Baran and Will Buik

E158

April 19, 2010



Introduction

The goal of this project was to create a functioning implementation of the SHA-1 cryptographic hash algorithm. The SHA-1 algorithm takes in a message of up to 2^{64}-1 bits and produces a unique 160-bit hash code. The hash code is computed through a series of nonlinear functions and rotations of temporary 32-bit values until the entire message has been encoded. This design was intended to be implemented on a 1.5x1.5 mm 40-pin MOSIS “TinyChip” fabricated in a 0.6 µm process.

Specifications

The SHA-1 cryptographic hash algorithm operates on 32-bit words, and processes 16 word long message blocks in an iterative process consisting of 80 rounds. The algorithm maintains a five word state that each round modifies to produce the final hash. To process each message block, the five word state is copied, each round modifies the copy, and it is then added to the original state. The algorithm extends the 16 word message into an 80 word message, but since each extended message word depends only on the previous 16, only 16 words of storage are needed to handle the message extension. The algorithm as implemented requires 16 words of storage for the message block, ten words of storage for the state, two words of storage for temporary values, and ten hard coded constant values. Further details of the SHA1 algorithm may be found in Appendix C.

Architecture

The core is broken down into three top level cells: a controller, counter bank, and datapath. The controller was synthesized from Verilog code, while the counter bank and datapath, both of which express a higher degree of regularity, were laid out by hand. The datapath contains two memory banks, a ROM, a single temporary register implemented as a flip flop, an ALU, and an input output direction selector. The counter bank consists of three five-bit counters and are used by the controller to keep track of rounds and memory locations. Finally, the controller directs the multiplexors and tristates in the datapath and interfaces the counters to coordinate the SHA-1 algorithm.

Since the SHA-1 algorithm operates on 32-bit words, it was decided that the inputs and outputs would also be 32 bits wide. Because there are only 40 pins on the TinyChip, it was determined that the input and output words would have to share the same pins. This was found to not be a problem since the design uses additional pins to control whether it is inputting or outputting data, making it impossible for data to be moving in both directions at the same time. Table 1 shows the inputs and outputs of the chip.

Pin Name	Direction	Width
vdd	Input	1
gnd	Input	1
io	Input/Output	32
ph1	Input	1
ph2	Input	1
Reset	Input	1

Block	Input	1
Hash	Output	1
Ready	Output	1

Table 1: Listing of SHA-1 Inputs and Outputs

VDD and GND are the power and ground lines for the chip. IO is the shared input and output pins. Ph1 and Ph2 are two-phase non-overlapping clocks. Reset is the global reset for all of the cells. Block signals that the chip should input a sixteen word message block on the subsequent sixteen cycles. Hash signals that the chip should output the five word hash on the subsequent five cycles. Ready indicates that the chip can accept a message block or output the hash. In addition, Ready indicates the direction of the input output pins. When Ready is high, the io pins are in input mode.

SRAM

The SRAM was responsible for storing the sixteen 32-bit words that make up the 512-bit message block. This cell contained an address buffer and 16 single-word SRAM cells.

Memory and Constant Bank

The memory and constant bank was responsible for storing the words of state as well as the constants used in the algorithm. It contained a ROM/RAM block made up of a 9-word mask ROM and an 11-element SRAM block; a flip-flop to control the data flow into the SRAM; and a 2-input mux to choose between the SRAM and ROM outputs. Though the SHA-1 algorithm uses 10 constants, one of those constants is zero, which does not need to be masked. The SRAM and ROM cells are interlaced to share the SRAM decoders.

ALU

The ALU was responsible for performing all of the calculations for the algorithm. It contained a "Shift A" cell that selected between unshifted, 5-bit left shifted, or 30-bit left shifted values of data coming from the temporary flip-flop, a "Shift B" cell that selected between unshifted and 1-bit left shifted values of data from either the SRAM or Memory and Constant Bank, an inverter, an inverting xnor cell, an inverting fulladder, a nand gate, and multiplexors to select the operation.

Flip-Flop

The flip-flop was used to hold temporary values outputted by the ALU. These values would later be passed back into the ALU for further processing.

I/O Selection Block

The I/O select block determined whether to pass the data coming out of the ALU back into the datapath or to the output pins.

The schematics and layouts of all custom blocks may be seen in Appendix B.

Floor Plan:

The final floor plan consumed almost twice as much area as the initial estimate, as seen in Figure 1. This was due primarily to the decision to implement the full SHA1 algorithm instead of the initially proposed version which would have acted on 8-bit data words. The final controller became larger because more states were required to handle the full algorithm and the datapath grew as it became 32 bits wide rather than 8. In addition, the initial estimate did not integrate the 16 Word Shift Register into the datapath and the final version of the chip separated the three counters from the synthesized controller.

The controller and counter bank both used standard muddlib10 cells, inverting every other row to share VDD and ground lines. These blocks had a pitch of 90 lambda and were six rows and five rows tall respectively. The datapath, however, could not use standard muddlib10 cells as they were too large to fit all 32 slices reasonably onto the chip. As such, reduced height cells were designed with to be six wiring tracks tall rather than eight. The datapath consisted of three zipper rows using muddlib10 cells at 110 lambda pitch, and 32 rows of reduced height cells at a 70 lambda pitch by inverting every other row. The ground plane for the first zipper row was shared with the 32^{nd} row of reduced height cells.

The third row of zippers was used for write address decoding and buffering, the second row of zippers was used for read address decoding and buffering, and the first row of zippers was general purpose. Except for the memories, all word slices had a single row of zipper logic. The memory banks required all three rows for address decoding, and had address line buffers hanging off to the side. These address line buffers fit over the other word slice zippers since they only used the first zipper row.

The sliceplans of the datapath, ALU, and IOSelect modules are shown in Figure 2. The pinout of the chip can be seen in Figure 3.

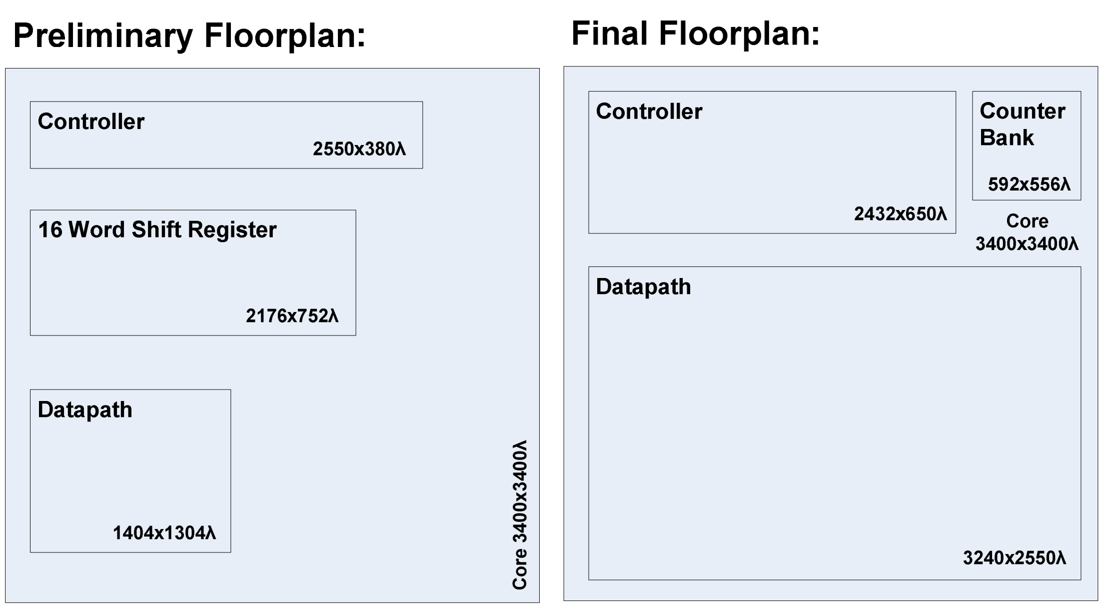


Figure 1: Preliminary and Final Floorplans.

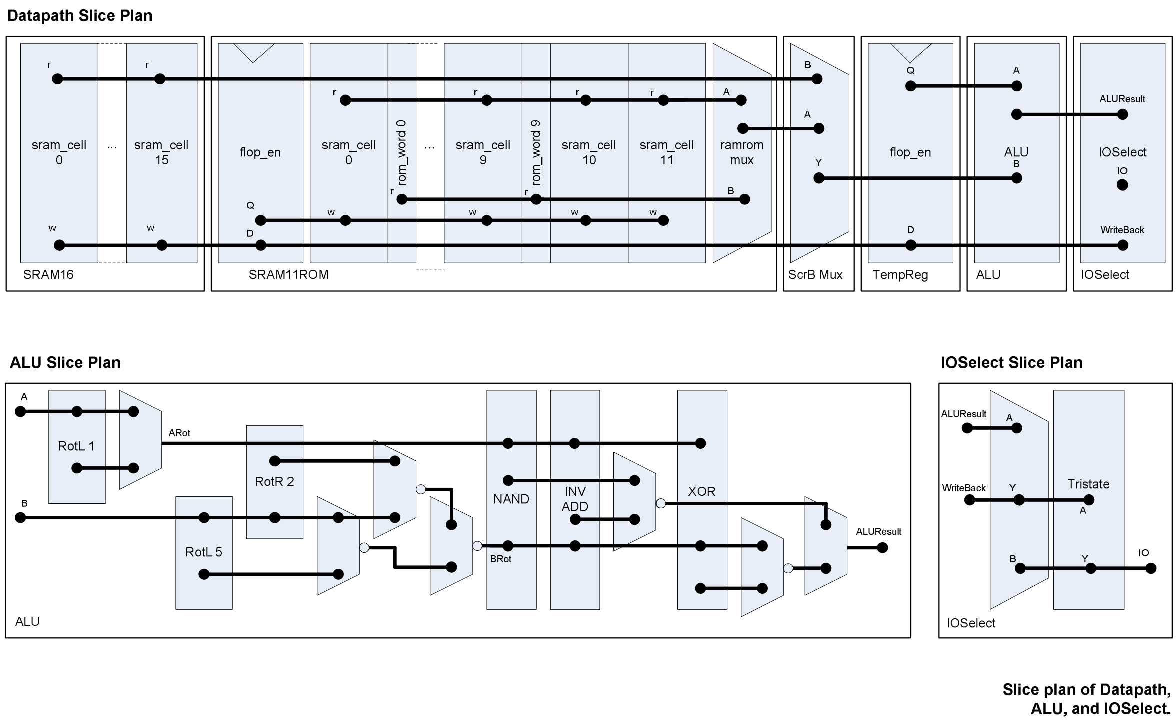


Figure 2: Slice Plans of the Datapath, ALU and IOSelect.

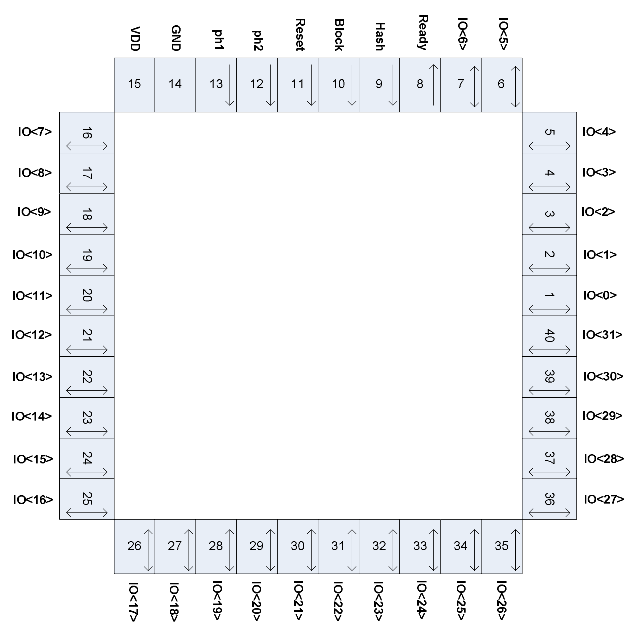


Figure 3: Pinout Diagram

Verification:

The design was first verified as a Verilog model. A test bench generator was written, the code for which is in Appendix D. This generator was verified against a known working implementation of the SHA-1 algorithm. A self checking test bench was designed to test the Verilog model. It tested only a single random message block, but due to the nature of the SHA-1 algorithm any small error propagates to the final hash, often drastically changing the result. The Verilog model was designed to be as close to the actual design as possible, and can be found in Appendix A.

Next, the schematics of each cell were designed using the Virtuoso Schematic Editor. Each was netlisted and plugged in individually to the Verilog model to verify functionality. The controller was synthesized from the Verilog model since it had little regularity and would not benefit from being constructed by hand. Once the design was completed up to the core schematic, the netlisted design could be tested alone with the same self checking test bench used to verify the Verilog model.

The only discrepancy in net listing was an intermittent error that caused the inverting 32 bit full adder to sometimes not netlist correctly. Sometimes invalid verilog code for this schematic was generated, that cut off before the proper end of the line. To handle this, a working copy of the inverting 32 bit full adder netlist was saved from one of the times it netlisted properly and used in the event that it did not.

Next, each cell was laid out using layout of the chip using the Virtuoso Layout Editor. The controller layout was synthesized using SOC Encounter. DRC and LVS were used to ensure that the layouts did not violate design rules and that logical netlist and transistor parameters matched the schematics.

All cells passed DRC, and all cells except the datapath passed LVS. The team ran out of time while wiring the datapath, and as a result it did not pass LVS. Since the datapath layout was not complete, the core layout was not auto routed or placed into a padframe. As such a CIF could not be created.

Had the datapath been completed, the core layout would have been auto routed and placed into a padframe. To verify the function of the chip, the top level chip schematic containing padframe and core would have been netlisted and verified against the same self checking test bench used for the core. Assuming the chip layout passed DRC and LVS, it would have been exported to a CIF, and then re-imported to verify that DRC and LVS still passed.

Post-Fabrication Test Plan:

Due to the tight spacing of elements on the chip, adding a self testing feature to the device was infeasible. As such, the chip must tested using its standard operating procedure. To do this, the chip must be restarted, a sample message loaded into its memory, and the hash value compared to a known value. A programmable microcontroller may be used to test the chip. Due to the iterative and cryptographic natures of the SHA1 algorithm, a single sample message is sufficient to test all facilities of the chip. Since each of the 80 rounds is designed to introduce a high degree of entropy into the algorithm's state, this single test message will likely hit all corner cases of the device. Additionally, any error that occurs during the progress of the algorithm will propagate to the final hash output, making failures simple to detect.

Errors in the controller or counters will cause the hash output to be incorrect if they cause the controller to improperly control the datapath. If the next state logic fails, the hash output will either be incorrect, or the chip will never reach the ready state to load the data or to output the hash. Errors in the datapath will cause the hash value to be incorrect.

By carefully selecting the test message, which consists of 16 32-bit words of input and five 32-bit words of expected output, stuck at failures for the input-output pins can be detected. The test input should ensure that every pin is asserted high and low, and should be chosen such that the hash produced causes the pins to be asserted both high and low when outputting the five result words. If any of the controller input or output pins are stuck, the chip will fail to output the hash.

Design Time

The team allocated the following person-hours to the project:

Project Proposal:	12 Hours
Verilog Model:	24 Hours
Schematics:	28 Hours
Layout:	80 Hours
Final Report:	14 Hours

File Locations:

Verilog Code:	`/home/wbuik/project2_final/sha1full.sv`
Test Vectors:	`/home/wbuik/project2_final/sha1full.sv`
Synthesis Results:	`/home/wbuik/project2_final/core_run1/`
Cadence Libraries:	`/home/wbuik/project2_final/sha1/` `/home/wbuik/project2_final/sha1_alu/`
CIF:	N/A *(Image on cover page is unrouted core plot not CIF chip plot.)*
PDF Core Plot:	`/home/wbuik/project2_final/core.pdf`
PDF of Report:	`/home/wbuik/project2_final/finalreport.pdf`

Appendix A (Verilog Code and Test Vectors)

```
typedef enum logic [4:0] {
    H0 = 5'b00000,
    H1 = 5'b00001,
    H2 = 5'b00010,
    H3 = 5'b00011,
    H4 = 5'b00100,
    A = 5'b00101,
    B = 5'b00110,
    C = 5'b00111,
    D = 5'b01000,
    E = 5'b01001,
    T2 = 5'b01010,

    AInit = 5'b10000,
    BInit = 5'b10001,
    CInit = 5'b10010,
    DInit = 5'b10011,
    EInit = 5'b10100,
    K1 = 5'b10101,
    K2 = 5'b10110,
    K3 = 5'b10111,
    K4 = 5'b11000,
    ZERO = 5'b11001
} regreadaddr;

typedef enum logic [3:0] {
    H0W = 4'b0000,
    H1W = 4'b0001,
    H2W = 4'b0010,
    H3W = 4'b0011,
    H4W = 4'b0100,
    AW = 4'b0101,
    BW = 4'b0110,
    CW = 4'b0111,
    DW = 4'b1000,
    EW = 4'b1001,
    T2W = 4'b1010
} regwriteaddr;

typedef enum logic [1:0] {
    NONE = 2'b00,
    THIRTY_B = 2'b10,
    FIVE_B = 2'b01,
    ONE_A = 2'b11
} alushift;

typedef enum logic [1:0] {
    PASS = 2'b00,
    ADD = 2'b10,
    XOR = 2'b01,
    AND = 2'b11
} aluop;

typedef enum logic {
    RegOut = 1'b0,
    WOut = 1'b1
} srcb;

module controller(
    // Clock Inputs
    input logic ph1, ph2, Reset,
    // Chip Inputs
    input logic Block, Hash,
    // Chip Outputs
    output logic Ready,
    // Datapath Control
    output logic [3:0] WAddr,
    output logic WEn,
    output regreadaddr RegReadAddr,
    output regwriteaddr RegWriteAddr,
    output logic RegWrite,
    output logic TempWrite,
    output srcb SrcB,
    output alushift ALUShift,
    output aluop ALUOp,
    output logic CounterAReset,
    output logic CounterAInc,
    input logic [4:0] CounterA,
```

```
output logic CounterBReset,
output logic CounterBInc,
input logic [1:0] CounterB,
output logic CounterCReset,
input logic [3:0] CounterC,
output logic [5:0] state);

// WAddr calculation
logic [1:0] WOutSel;
logic [3:0] WOffset0;
logic [3:0] WOffset1;
logic [3:0] WOffset2;
logic [3:0] WOffset3;
logic [3:0] WOffset;
assign WOffset0 = 0;
assign WOffset1 = 2;
assign WOffset2 = 8;
assign WOffset3 = 13;
mux4 #(4) woffsetmux(WOffset0, WOffset1, WOffset2, WOffset3, WOutSel, WOffset);
assign WAddr = CounterC + WOffset;

// Debugging
/*always @(posedge ph2)
begin
if(WEn == 1 || SrcB == WOut) begin
   $display("%d: %d, %b, %d", state, CounterC, WOutSel, WAddr);
end
end*/


// State
//logic [5:0] state;
logic [5:0] nextstate;
logic [5:0] stateplusone;
assign stateplusone = state + 1;

// State Register
flop #(6) statereg(ph1, ph2, nextstate, state);

// Next State Logic
always_comb
begin
    if(Reset == 1) nextstate = 0;
    else begin
    case(state)
        5:  // Ready
            if (Hash == 1) nextstate = 57;  // Ready -> Hash
            else if (Block == 1) nextstate = 6;  // Ready -> Read Block
            else nextstate = 5;  // Ready -> Ready

        6:  // Read Block
            if (CounterA == 15) nextstate = 7;  // Read Block -> Block Init
            else nextstate = 6;  // Read Block -> Read Block

        12:
            case (CounterB)
                0: nextstate = 13;  // Round Init -> RoundA
                1: nextstate = 18;  // Round Init -> RoundB
                2: nextstate = 22;  // Round Init -> RoundC
                3: nextstate = 29;  // Round Init -> RoundD
            endcase
        17:
            nextstate = 33; // RoundA -> Round Cleanup
        21:
            nextstate = 33; // RoundB -> Round Cleanup
        28:
            nextstate = 33; // RoundC -> Round Cleanup
        32:
            nextstate = 33; // RoundD -> Round Cleanup
        45:
            if (CounterA == 19) nextstate = 46; // MessageExtension -> Next Round Class
            else nextstate = 12; // MessageExtension -> Round Init

        46:
            if (CounterB == 3) nextstate = 47; // Next Round Class -> Block Cleanup
            else nextstate = 12; // Next Round Class -> Round Init
        56:
            nextstate = 5; // Block Cleanup -> Ready
        61:
            nextstate = 5; // Hash -> Ready
        default:
```

```
            nextstate = stateplusone;
    endcase
    end
end

// Output Logic
always_comb
begin
    Ready = 0;
    WOutSel = 2'b00;
    WEn = 0;
    RegReadAddr = H0;
    RegWriteAddr = H0W;
    RegWrite = 0;
    TempWrite = 0;
    SrcB = RegOut;
    ALUShift = NONE;
    ALUOp = PASS;
    CounterAReset = 0;
    CounterBReset = 0;
    CounterCReset = 0;
    CounterAInc = 0;
    CounterBInc = 0;

    case (state)
        // Init Chip
        0:  begin
            RegReadAddr = AInit;
            RegWriteAddr = H0W;
            RegWrite = 1;
            end
        1:  begin
            RegReadAddr = BInit;
            RegWriteAddr = H1W;
            RegWrite = 1;
            end
        2:  begin
            RegReadAddr = CInit;
            RegWriteAddr = H2W;
            RegWrite = 1;
            end
        3:  begin
            RegReadAddr = DInit;
            RegWriteAddr = H3W;
            RegWrite = 1;
            end
        4:  begin
            RegReadAddr = EInit;
            RegWriteAddr = H4W;
            RegWrite = 1;
            end

        // Ready
        5:  begin
            CounterAReset = 1;
            CounterCReset = 1;
            Ready = 1;
            end

        // Read Block
        6:  begin
            WEn = 1;
            CounterAInc = 1;
            Ready = 1;
            end

        // Block Init
        7:  begin
            RegReadAddr = H0;
            RegWriteAddr = AW;
            RegWrite = 1;
            end
        8:  begin
            RegReadAddr = H1;
            RegWriteAddr = BW;
            RegWrite = 1;
            end
        9:  begin
            RegReadAddr = H2;
            RegWriteAddr = CW;
            RegWrite = 1;
```

```
    end
10: begin
    RegReadAddr = H3;
    RegWriteAddr = DW;
    RegWrite = 1;
    end
11: begin
    RegReadAddr = H4;
    RegWriteAddr = EW;
    RegWrite = 1;
    CounterBReset = 1;
    CounterAReset = 1;
    end

// Round Init
//12:

// RoundA
13: begin
    RegReadAddr = D;
    TempWrite = 1;
    end
14: begin
    RegReadAddr = C;
    TempWrite = 1;
    ALUOp = XOR;
    end
15: begin
    RegReadAddr = B;
    TempWrite = 1;
    ALUOp = AND;
    end
16: begin
    RegReadAddr = D;
    TempWrite = 1;
    ALUOp = XOR;
    end
17: begin
    RegReadAddr = K1;
    TempWrite = 1;
    ALUOp = ADD;
    end

// RoundB
18: begin
    RegReadAddr = B;
    TempWrite = 1;
    end
19: begin
    RegReadAddr = C;
    TempWrite = 1;
    ALUOp = XOR;
    end
20: begin
    RegReadAddr = D;
    TempWrite = 1;
    ALUOp = XOR;
    end
21: begin
    RegReadAddr = K2;
    TempWrite = 1;
    ALUOp = ADD;
    end


// RoundC
22: begin
    RegReadAddr = C;
    TempWrite = 1;
    end
23: begin
    RegReadAddr = D;
    RegWriteAddr = T2W;
    RegWrite = 1;
    ALUOp = AND;
    end
24: begin
    RegReadAddr = C;
    TempWrite = 1;
    end
25: begin
```

```
    RegReadAddr = D;
    TempWrite = 1;
    ALUOp = XOR;
    end
26: begin
    RegReadAddr = B;
    TempWrite = 1;
    ALUOp = AND;
    end
27: begin
    RegReadAddr = T2;
    TempWrite = 1;
    ALUOp = XOR;
    end
28: begin
    RegReadAddr = K3;
    TempWrite = 1;
    ALUOp = ADD;
    end

// RoundD
29: begin
    RegReadAddr = B;
    TempWrite = 1;
    end
30: begin
    RegReadAddr = C;
    TempWrite = 1;
    ALUOp = XOR;
    end
31: begin
    RegReadAddr = D;
    TempWrite = 1;
    ALUOp = XOR;
    end
32: begin
    RegReadAddr = K4;
    TempWrite = 1;
    ALUOp = ADD;
    end

// Round Cleanup
33: begin
    RegReadAddr = E;
    TempWrite = 1;
    ALUOp = ADD;
    end
34: begin
    RegReadAddr = A;
    TempWrite = 1;
    ALUOp = ADD;
    ALUShift = FIVE_B;
    end
35: begin
    SrcB = WOut;
    TempWrite = 1;
    ALUOp = ADD;
    end
36: begin
    RegReadAddr = D;
    RegWriteAddr = EW;
    RegWrite = 1;
    ALUOp = PASS;
    end
37: begin
    RegReadAddr = C;
    RegWriteAddr = DW;
    RegWrite = 1;
    ALUOp = PASS;
    end
38: begin
    RegReadAddr = B;
    RegWriteAddr = CW;
    RegWrite = 1;
    ALUOp = PASS;
    ALUShift = THIRTY_B;
    end
39: begin
    RegReadAddr = A;
    RegWriteAddr = BW;
    RegWrite = 1;
```

```
    ALUOp = PASS;
    end
40: begin
    RegReadAddr = ZERO;
    RegWriteAddr = AW;
    RegWrite = 1;
    ALUOp = ADD;
    end

// Message Extension
41: begin
    SrcB = WOut;
    TempWrite = 1;
    end
42: begin
    WOutSel = 2'b01;
    SrcB = WOut;
    TempWrite = 1;
    ALUOp = XOR;
    end
43: begin
    WOutSel = 2'b10;
    SrcB = WOut;
    TempWrite = 1;
    ALUOp = XOR;
    end
44: begin
    WOutSel = 2'b11;
    SrcB = WOut;
    TempWrite = 1;
    ALUOp = XOR;
    end
45: begin
    RegReadAddr = ZERO;
    TempWrite = 1;
    ALUOp = ADD;
    ALUShift = ONE_A;
    WEn = 1;
    CounterAInc = 1;
    end

// Next Round Class
46: begin
    CounterBInc = 1;
    CounterAReset = 1;
    end

// Block Cleanup
47: begin
    RegReadAddr = A;
    TempWrite = 1;
    end
48: begin
    RegReadAddr = H0;
    RegWriteAddr = H0W;
    RegWrite = 1;
    ALUOp = ADD;
    end
49: begin
    RegReadAddr = B;
    TempWrite = 1;
    end
50: begin
    RegReadAddr = H1;
    RegWriteAddr = H1W;
    RegWrite = 1;
    ALUOp = ADD;
    end
51: begin
    RegReadAddr = C;
    TempWrite = 1;
    end
52: begin
    RegReadAddr = H2;
    RegWriteAddr = H2W;
    RegWrite = 1;
    ALUOp = ADD;
    end
53: begin
    RegReadAddr = D;
    TempWrite = 1;
```

```
            end
        54: begin
            RegReadAddr = H3;
            RegWriteAddr = H3W;
            RegWrite = 1;
            ALUOp = ADD;
            end
        55: begin
            RegReadAddr = E;
            TempWrite = 1;
            end
        56: begin
            RegReadAddr = H4;
            RegWriteAddr = H4W;
            RegWrite = 1;
            ALUOp = ADD;
            end

        // Output Hash
        57: RegReadAddr = H0;
        58: RegReadAddr = H1;
        59: RegReadAddr = H2;
        60: RegReadAddr = H3;
        61: RegReadAddr = H4;
      endcase
    end

endmodule


//
// Counter/Adder Bank
// Since the three counters never need to be reset or
// incremented at the same time, share the adder logic.
// Also share adder logic with WAddr calculation.
//
//module counterAdderBank
//   (input logic ph1, ph2,
//    input logic inc, reset,
//    input logic [1:0] counterSel
//    input logic [1:0] WOutSel,
//    output logic [4:0] counterOut);
//
//    input logic [4:0] ca, cb, cc, counter;
//    assign cb[4:2] = 0;
//    assign cc[4] = 0;
//
//    mux3 #(5) (ca, cb, cc, counterSel, counter);
//
//    assign counterOut = 1 + counter;
//
//   flopen #(5) counterareg(ph1, ph2, ~(counterSel[0] | counterSel[1]), counterOut, ca);
//    flopen #(2) counterbreg(ph1, ph2, counterSel[0], counterOut, cb[1:0]);
//    flopen #(4) countercreg(ph1, ph2, counterSel[1], counterOut, cc[3:0]);
//
//endmodule

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

     logic [WIDTH-1:0] yplusone;
     assign yplusone = y + 1;

     flopenr #(WIDTH) counterreg(ph1, ph2, reset, en, yplusone, y);
endmodule

module datapath
  #(parameter WIDTH = 32)
   (input logic ph1,
    input logic ph2,
    input logic [3:0] WAddr,
    input logic WEn,
    input logic [4:0] RegReadAddr,
    input logic [3:0] RegWriteAddr,
    input logic RegWriteEn,
    input logic TempWrite,
    input logic SrcB,
    input logic [1:0] ALUshift,
```

```
    input logic [1:0] ALUop,
    input logic IOSel,
    inout [WIDTH-1:0] io);

    logic [WIDTH-1:0] A;
    logic [WIDTH-1:0] B;
    logic [WIDTH-1:0] Wout;
    logic [WIDTH-1:0] RegOut;
    logic [WIDTH-1:0] ALUresult;
    logic [WIDTH-1:0] WriteData;

    //instantiate modules
    mux2 #(WIDTH) Bsource (RegOut, Wout, SrcB, B);
    flopen #(WIDTH) t1reg(ph1, ph2, TempWrite, WriteData, A);
    FinalAlu #(WIDTH) alu (A, B, ALUshift, ALUop, ALUresult);
    SRAM16 #(WIDTH) wmem (ph1, ph2, WAddr, WEn, WriteData, Wout);
    MemConstBank #(WIDTH) bank (RegReadAddr, RegWriteAddr, RegWriteEn, WriteData, ph1, ph2,
RegOut);
    ioselector #(WIDTH) ioselect (ALUresult, IOSel, WriteData, io);

endmodule


// Input Output Selector
module ioselector
  #(parameter WIDTH = 8)
   (input logic [WIDTH-1:0] ValueIn,
    input logic IOSel,
    output logic [WIDTH-1:0] ValueOut,
    inout [WIDTH-1:0] IO);

    mux2 #(WIDTH) wdmux (ValueIn, IO, IOSel, ValueOut);
    tristate  #(WIDTH) iotri (ValueIn, ~IOSel, IO);
endmodule


module tristate
  #(parameter WIDTH = 8)
   (input logic [WIDTH-1:0] in,
    input logic en,
    output logic [WIDTH-1:0] out);

    assign out = en ? in : {(WIDTH){1'bz}};
endmodule


// SRAM Module for Message State, 16 Words
module SRAM16
  #(parameter WIDTH = 8)
   (input ph1, input ph2,
    input [3:0] Addr,
    input WriteEn,
    input [WIDTH-1:0] DataIn,
    output logic [WIDTH-1:0] DataOut);

    logic [WIDTH-1:0] RAM [15:0];

    always @ ( * )
         if(ph2 & WriteEn) RAM[Addr] <= DataIn;

    assign DataOut = RAM[Addr];
endmodule

//This module stores words and provides words and constants as needed
//These are the addresses of the variables used in the algorithm
//They are only 4 bits since they are used by RegWriteAddr, which is also 4 bits
//When used by RegReadAddr as a 5-bit number, they will have a 0 padded onto the front
module MemConstBank
  #(parameter WIDTH = 8)
   (input [4:0] RegReadAddr,
    input [3:0] RegWriteAddr,
    input RegWriteEn,
    input [WIDTH-1:0] RegIn,
    input ph1, input ph2,
    output logic [WIDTH-1:0] RegOut);



    logic [WIDTH-1:0] ROM [9:0];
```

```
    assign ROM[0] = 32'h67452301;
    assign ROM[1] = 32'hEFCDAB89;
    assign ROM[2] = 32'h98BADCFE;
    assign ROM[3] = 32'h10325476;
    assign ROM[4] = 32'hC3D2E1F0;
    assign ROM[5] = 32'h5A827999;
    assign ROM[6] = 32'h6ED9EBA1;
    assign ROM[7] = 32'h8F1BBCDC;
    assign ROM[8] = 32'hCA62C1D6;
    assign ROM[9] = 32'b0;

    // Flop for ram sequencing
    logic [WIDTH-1:0] RegInSeq;
    flop #(WIDTH) seq(ph2, ph1, RegIn, RegInSeq);

    logic [WIDTH-1:0] RamData;
    SRAM11 #(WIDTH) ram(ph1, ph2, RegReadAddr[3:0], RegWriteAddr, RegWriteEn, RegInSeq, RamData);

    //always @(*)
    //    if (ph2 & RegWriteEn) RAM[RegWriteAddr] <= RegInSeq;

    assign RegOut = RegReadAddr[4] ? ROM[RegReadAddr[3:0]] : RamData;
endmodule


// SRAM Module for Message State, 16 Words
module SRAM11
  #(parameter WIDTH = 8)
   (input ph1, input ph2,
    input [3:0] RAddr,
    input [3:0] WAddr,
    input WriteEn,
    input [WIDTH-1:0] DataIn,
    output logic [WIDTH-1:0] DataOut);

    logic [WIDTH-1:0] RAM [10:0];

    always @(negedge ph2)
        if (WriteEn) RAM[WAddr] <= DataIn;

    assign DataOut = RAM[RAddr];
endmodule



/*This is the ALU used for the SHA algorithm
       It can left-circular shift the "A" input by 0, 5, or 6 places,
       and can left-circular shift the "B" input by 0 or 1 places
       It can AND, XOR, or sum the shifted A and B, or it can directly pass
       the shifted B
       This was implemented using NAND, XNOR, an inverted sum, and an inverter,
       with the output chosen by an inverting mux, to better reflect what the final
       chip will look like
*/
module FinalAlu
  #(parameter WIDTH = 8)
   (input [WIDTH-1:0] A, input [WIDTH-1:0] B,
    output [1:0] ALUrot, input [1:0] ALUop,
    output reg [WIDTH-1:0] Y);

    logic [WIDTH-1:0] Arot1;
    logic [WIDTH-1:0] Brot5;
    logic [WIDTH-1:0] BrotWMinus2;
    logic [WIDTH-1:0] Arot;
    logic [WIDTH-1:0] Brot;
    logic [WIDTH-1:0] OpNand;
    logic [WIDTH-1:0] OpXNOR;
    logic [WIDTH-1:0] OpAdd;
    logic [WIDTH-1:0] OpPass;
    logic [WIDTH-1:0] OpNandAdd;
    logic [WIDTH-1:0] OpXNORPass;

    // Rotation functionality for A and B
    rotl1 #(WIDTH) arot1(A, Arot1);
    rotl5 #(WIDTH) brot5(B, Brot5);
    rotr2 #(WIDTH) brot1(B, BrotWMinus2);
    mux2 #(WIDTH) arotmux(A, Arot1, ALUrot[0] & ALUrot[1], Arot);
    mux4 #(WIDTH) brotmux(B, Brot5, BrotWMinus2, B, ALUrot, Brot);

    //These are the functions that the ALU can perform on the inputs
    //They are inverted to better match how the final implementation will perform
```

```
    assign OpNand = ~(Arot & Brot);
    assign OpXNOR = ~(Arot ^ Brot);
    assign OpAdd  = ~(Arot + Brot);
    assign OpPass = ~Brot;

    // Output selector mux
    mux2inv #(WIDTH) outmux1(OpAdd, OpNand, ALUop[0], OpNandAdd);
    mux2inv #(WIDTH) outmux2(OpPass, OpXNOR, ALUop[0], OpXNORPass);
    mux2 #(WIDTH) outmux3(OpXNORPass, OpNandAdd, ALUop[1], Y);
endmodule

//
// Circular rotation modules
//
module rotl1
  #(parameter WIDTH = 8)
   (input logic [WIDTH-1:0] a,
    output logic [WIDTH-1:0] y);

    assign y[WIDTH-1:1] = a[WIDTH-2:0];
    assign y[0] = a[WIDTH-1];
endmodule

module rotl5
  #(parameter WIDTH = 8)
   (input logic [WIDTH-1:0] a,
    output logic [WIDTH-1:0] y);

    assign y[4:0] = a[WIDTH-1:WIDTH-5];
    assign y[WIDTH-1:5] = a[WIDTH-6:0];
endmodule

module rotr2
  #(parameter WIDTH = 8)
   (input logic [WIDTH-1:0] a,
    output logic [WIDTH-1:0] y);

    assign y[WIDTH-1:WIDTH-2] = a[1:0];
    assign y[WIDTH-3:0] = a[WIDTH-1:2];
endmodule


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

  logic [WIDTH-1:0] mid;

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


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

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

  assign resetval = 0;

  mux3 #(WIDTH) enrmux(q, d, resetval, {reset, en}, d2);
  flop #(WIDTH) f(ph1, ph2, d2, q);
endmodule

module flopen #(parameter WIDTH = 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) enrmux(q, d, en, d2);
  flop #(WIDTH) f(ph1, ph2, d2, q);
endmodule
```

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

  always_latch
    if (ph) q <= d;
endmodule


//
// Muxes
//
module mux2inv #(parameter WIDTH = 8)
                (input logic [WIDTH-1:0] d0,
                 input logic [WIDTH-1:0] d1,
                 input logic s,
                 output logic [WIDTH-1:0] y);
    assign y = ~(s ? d1: d0);
endmodule

module mux2 #(parameter WIDTH = 8)
                (input logic [WIDTH-1:0] d0,
                 input logic [WIDTH-1:0] d1,
                 input logic s,
                 output logic [WIDTH-1:0] y);
    assign y = (s ? d1: d0);
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 SHA1 #(parameter WIDTH = 32)
             (input ph1, ph2, Reset, Block, Hash,
              inout [WIDTH-1:0] io,
              output Ready);

    // Datapath Control
    logic [3:0] WAddr;
    logic WEn;
    regreadaddr RegReadAddr;
    regwriteaddr RegWriteAddr;
    logic RegWrite;
    logic TempWrite;
    srcb SrcB;
    alushift ALUShift;
    aluop ALUOp;

    // Counter Control
    logic CounterAReset;
    logic CounterBReset;
    logic CounterCReset;
    logic CounterAInc;
    ///logic CounterBInc;
    logic [4:0] CounterA;
    logic [1:0] CounterB;
```

```
    logic [3:0] CounterC;

    // Debug Out
    logic [5:0] state;

    controller ctrl(
        ph1, ph2, Reset, Block, Hash, Ready, WAddr,
        WEn, RegReadAddr, RegWriteAddr, RegWrite, TempWrite,
        SrcB, ALUShift, ALUOp, CounterAReset, CounterAInc, CounterA,
        CounterBReset, CounterBInc, CounterB, CounterCReset, CounterC,
        state);

    counter #(5) counter5(ph1, ph2, CounterAReset, CounterAInc, CounterA);
    counter #(2) counter2(ph1, ph2, CounterBReset, CounterBInc, CounterB);
    counter #(4) counter4(ph1, ph2, CounterCReset, WEn, CounterC);

    datapath #(WIDTH) dp(ph1, ph2, WAddr, WEn, RegReadAddr, RegWriteAddr, RegWrite,
                   TempWrite, SrcB, ALUShift, ALUOp, Ready, io);

endmodule

module testbench();
    // Clock
    logic ph1, ph2;

    // Chip Inputs
    logic Block, Hash, Reset;
    logic [31:0] DataIn;
    // Chip Outputs
    logic Ready;
    // ChipIO
    tri [31:0] DataIO;
    assign DataIO = Ready ? DataIn: 32'bz;
    SHA1 #(32) dut(ph1, ph2, Reset, Block, Hash, DataIO, Ready);

    // TV
    logic [2:0] testState;
    logic [10:0] vectornum;
    logic [31:0] TestVectors[100:0];
    logic [31:0] HashOut[4:0];

    assign TestVectors[0] = 32'h8A921FC4;
    assign TestVectors[1] = 32'h452C45D2;
    assign TestVectors[2] = 32'hABC243FE;
    assign TestVectors[3] = 32'hEC429CBD;
    assign TestVectors[4] = 32'h452C45D2;
    assign TestVectors[5] = 32'hEC429CBD;
    assign TestVectors[6] = 32'h452C45D2;
    assign TestVectors[7] = 32'hEC429CBD;
    assign TestVectors[8] = 32'h8A921FC4;
    assign TestVectors[9] = 32'h452C45D2;
    assign TestVectors[10] = 32'hABC243FE;
    assign TestVectors[11] = 32'h452C45D2;
    assign TestVectors[12] = 32'hEC429CBD;
    assign TestVectors[13] = 32'h452C45D2;
    assign TestVectors[14] = 32'h8A921FC4;
    assign TestVectors[15] = 32'h452C45D2;

    assign TestVectors[16] = 32'hFC258E41;
    assign TestVectors[17] = 32'hDFE90802;
    assign TestVectors[18] = 32'h64C65A1F;
    assign TestVectors[19] = 32'hDCB36023;
    assign TestVectors[20] = 32'h9FAEA24E;


    // generate clock to sequence tests
    always
    begin
        ph1 <= 0; ph2 <= 0; #1;
        ph1 <= 1; # 4;
        ph1 <= 0; #1;
        ph2 <= 1; # 4;
    end

    // Init chip
    initial begin
        //$readmemb("ctrl.tv", testvectors); // load test vectors
        testState = 0;
        Reset = 1; #17; Reset = 0; // come out of reset before cycle 2
        Block = 0;
        Hash = 0;
```

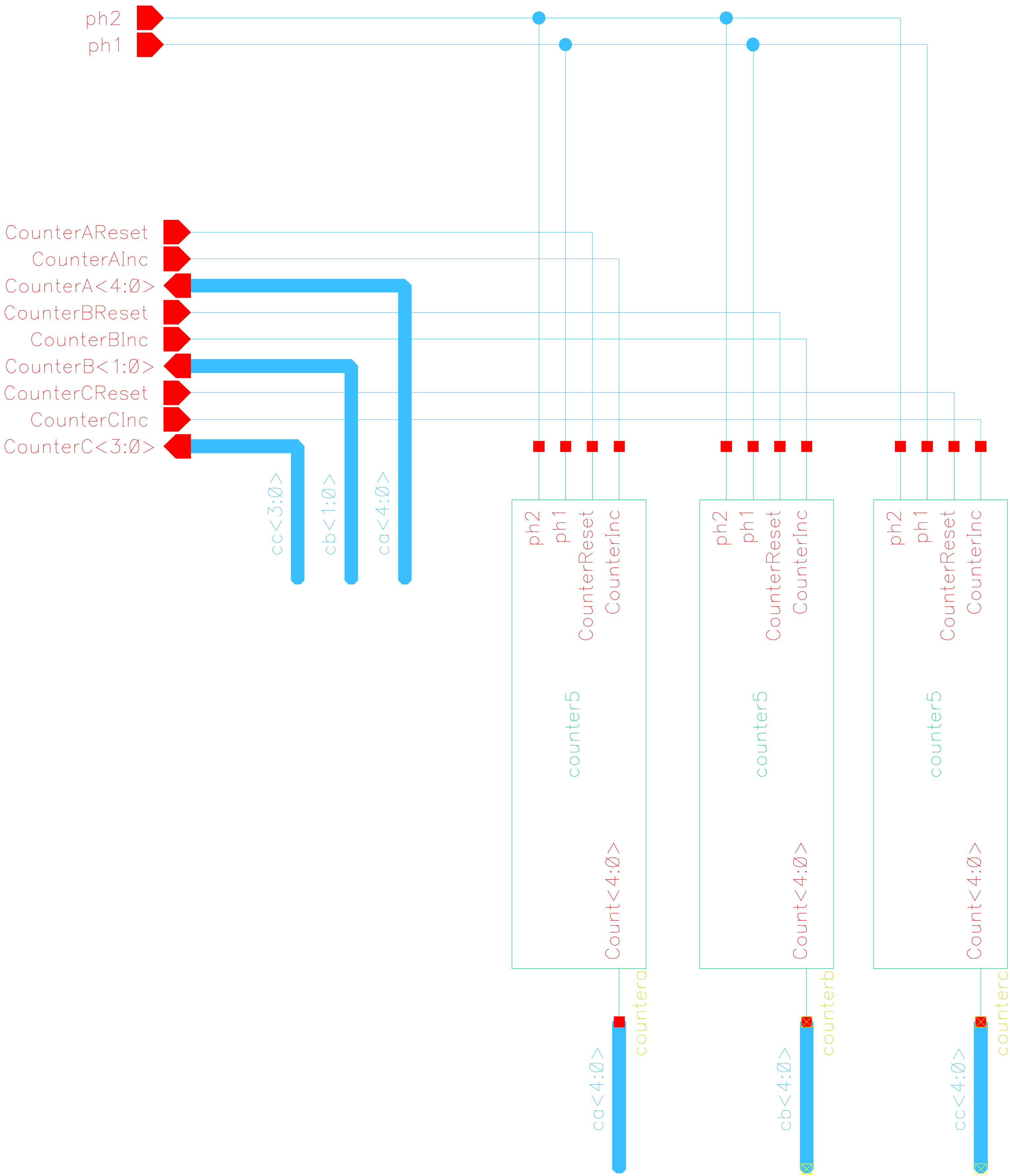
```
        DataIn = 0;
    end

    // Check results on ph2
    always @(posedge ph2) begin
        if(Hash && Ready) begin
            // Verify Hash
            if(HashOut[0] !== TestVectors[16] ||
               HashOut[1] !== TestVectors[17] ||
               HashOut[2] !== TestVectors[18] ||
               HashOut[3] !== TestVectors[19] ||
               HashOut[4] !== TestVectors[20]) begin
                $display("ERROR:");
                $display("EXPECTED: %H %H %H %H %H", TestVectors[16], TestVectors[17],
                         TestVectors[18], TestVectors[19], TestVectors[20]);
            end

            $display("HASHOUT:  %H %H %H %H %H", HashOut[0], HashOut[1],
                     HashOut[2], HashOut[3], HashOut[4]);
            $finish;
        end

        if(testState == 0) begin // wait for init
            if(Ready == 1) begin
                testState = 1;
                vectornum = 0;
                Block = 1;
            end
        end
        else if(testState == 1) begin // input msg block
            if(vectornum > 15) begin
                testState = 2;
            end
            vectornum += 1;
        end
        else if(testState == 2) begin // wait for hash result
            if(Ready == 1) begin
                Block = 0;
                Hash = 1;
                testState = 3;
                vectornum = 0;
            end
        end
        else if(testState == 3) begin // output hash data
            HashOut[vectornum] = DataIO;
            vectornum += 1;
            //$display("%h", DataIO);
        end
    end

    // Load data on ph1
    always @(posedge ph1) begin
        if(testState == 1) DataIn = TestVectors[vectornum];
    end

endmodule
```

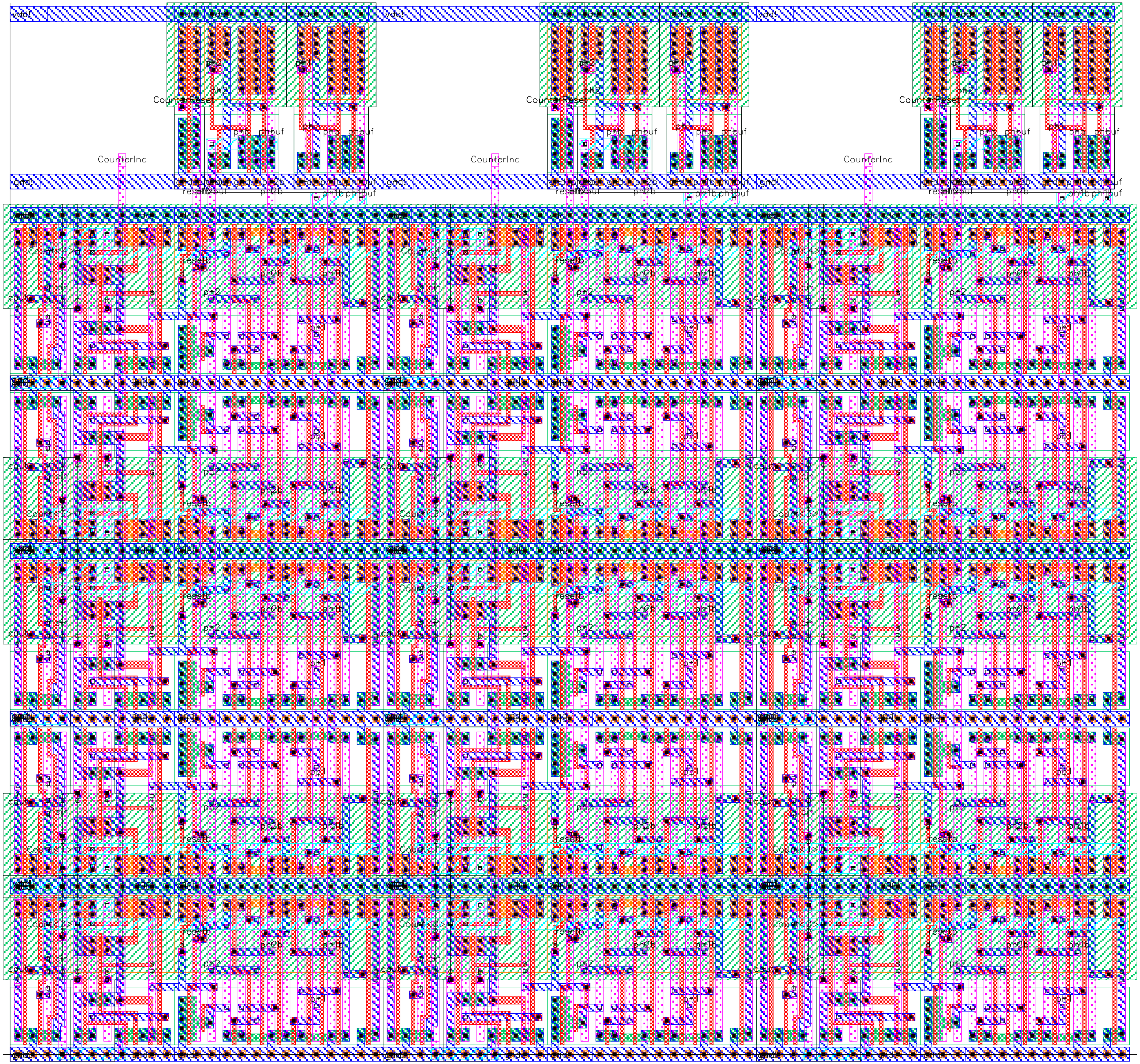
Appendix B (Schematics and Layout of Custom Cells)

Library	Cell Name	Notes
sha1	controller_fsm_synth	Layout only, schematic synthesized.
sha1	core	Layout not routed.
sha1	counter_bank	
sha1	counter_cell	
sha1	counter_zipper	
sha1	counter5	
sha1	datapath	Layout does not LVS.
sha1	flopen_dp	
sha1	flopen32_dp	
sha1	flopr	
sha1	halfadder_dp	
sha1	inv_dp	
sha1	ioselect	
sha1	memconstbank	
sha1	mux2_32	
sha1	mux2_dp	
sha1	nand4_1x	
sha1	rom_bitcond	
sha1	rom_bitcond32	
sha1	rom_one	
sha1	rom_word0	
sha1	rom_word1	
sha1	rom_word2	
sha1	rom_word3	
sha1	rom_word4	
sha1	rom_word5	
sha1	rom_word6	
sha1	rom_word7	
sha1	rom_word8	
sha1	rom_zero	Layout only, schematic empty.
sha1	sram_cell	
sha1	sram11_addrbuf	
sha1	sram11_word	
sha1	sram11_zipper	
sha1	sram11rom	
sha1	sram16	
sha1	sram16_addrbuf	
sha1	sram16_word	
sha1	sram16_zipper	
sha1	tristate	
sha1	xor2	
sha1_alu	alu	

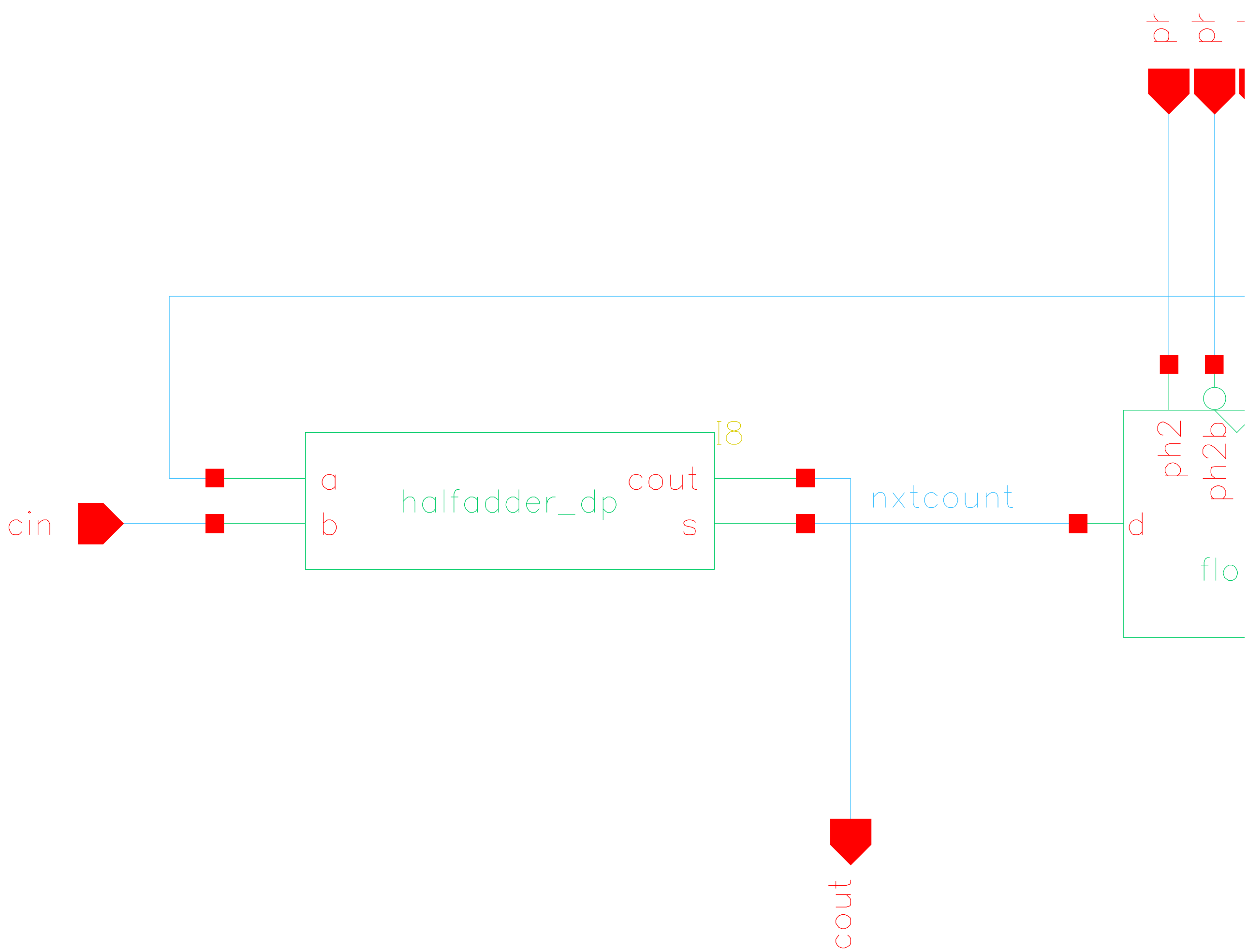
sha1_alu	invadder_32
sha1_alu	invfulladder
sha1_alu	invmux2_32
sha1_alu	invmux2_dp
sha1_alu	nand2_1x
sha1_alu	nand2_32x
sha1_alu	shift_a
sha1_alu	shift_b
sha1_alu	xnor2
sha1_alu	xnor2_32x

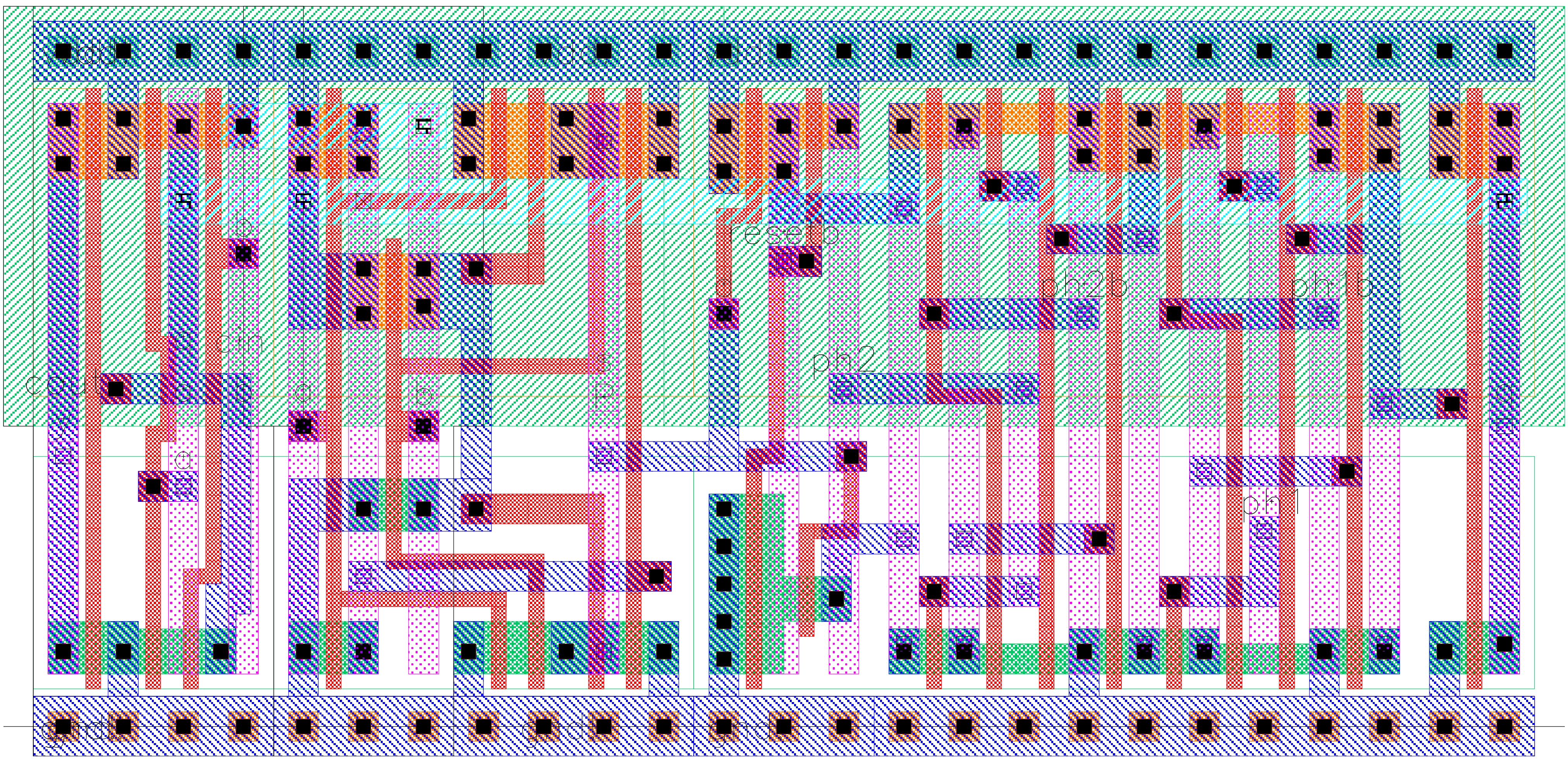
sha1.counter_bank



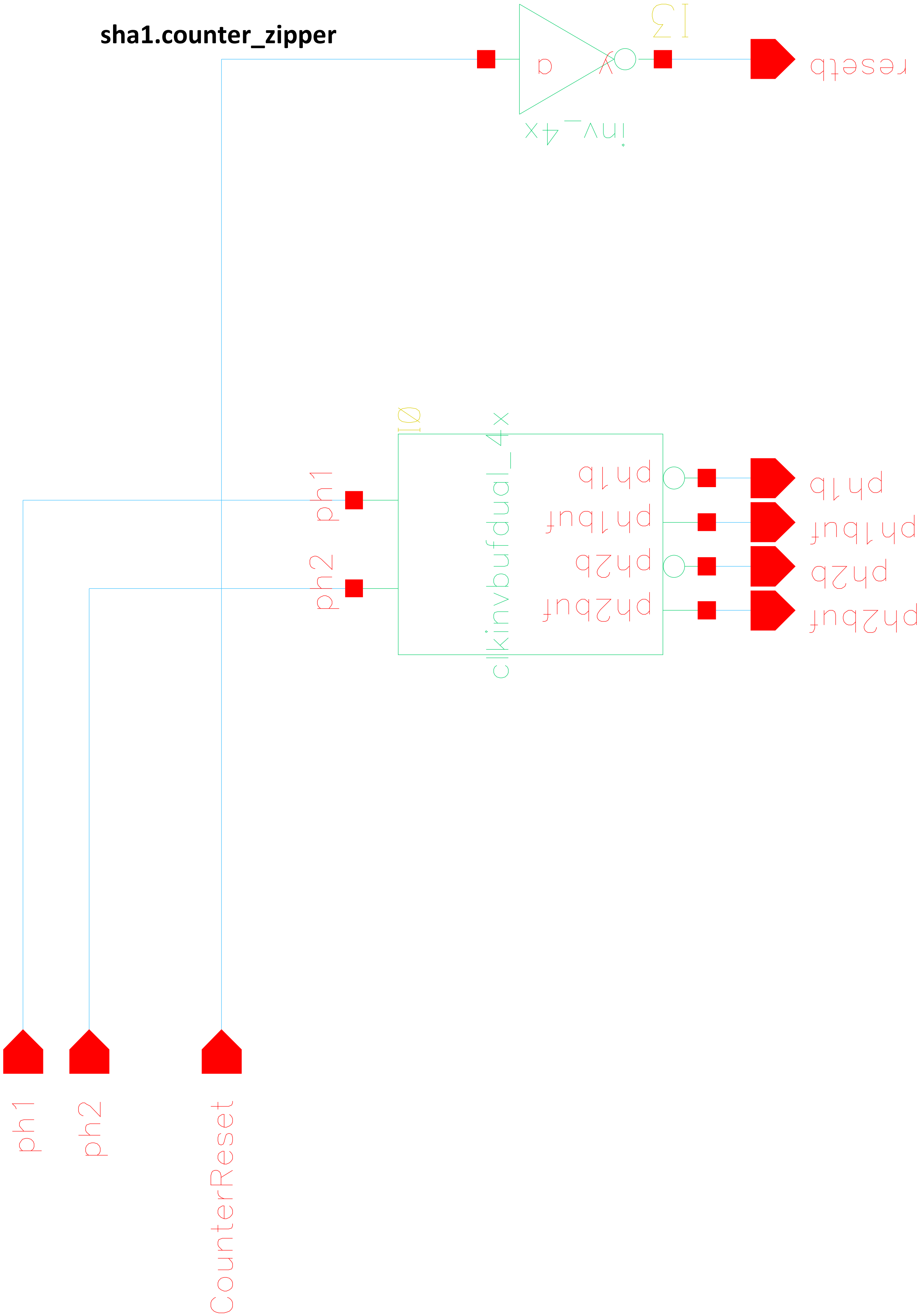

CounterReset
CounterInc

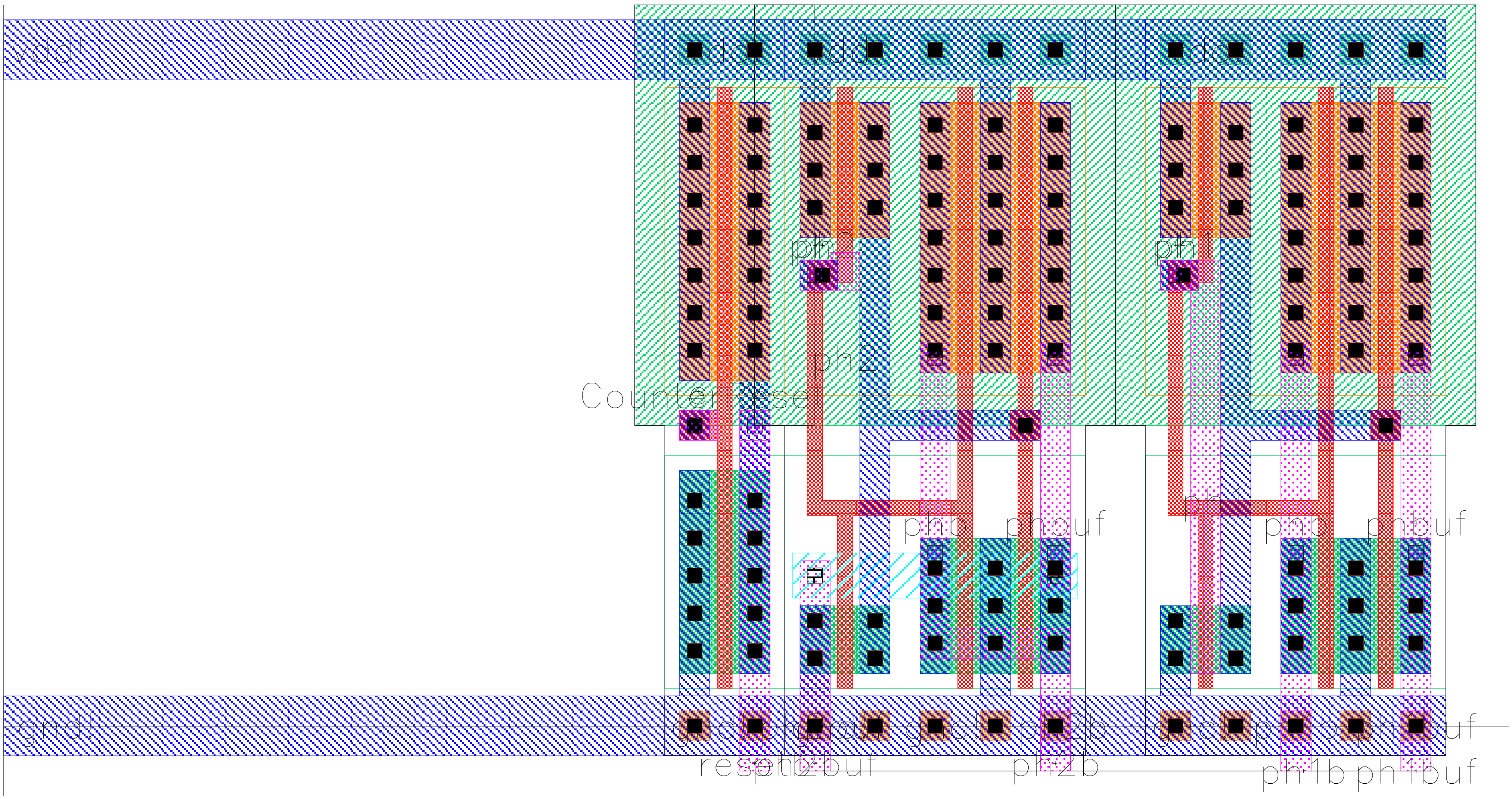
sha1.counter_cell



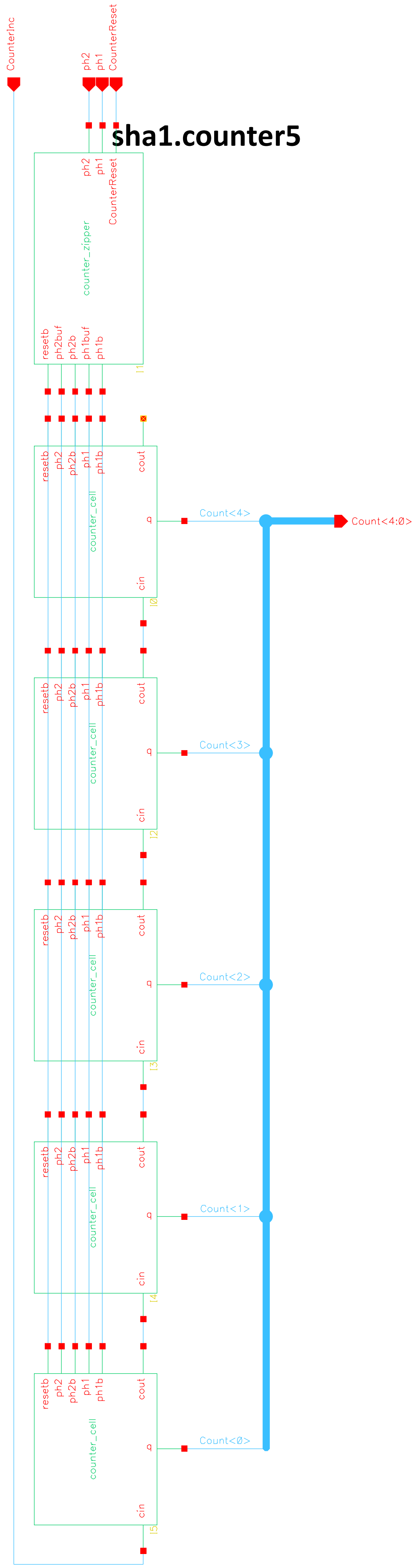

vdd
cout
cin
a
b
s
resetb
d
ph2
ph2b
ph1b
ph1
q
gnd

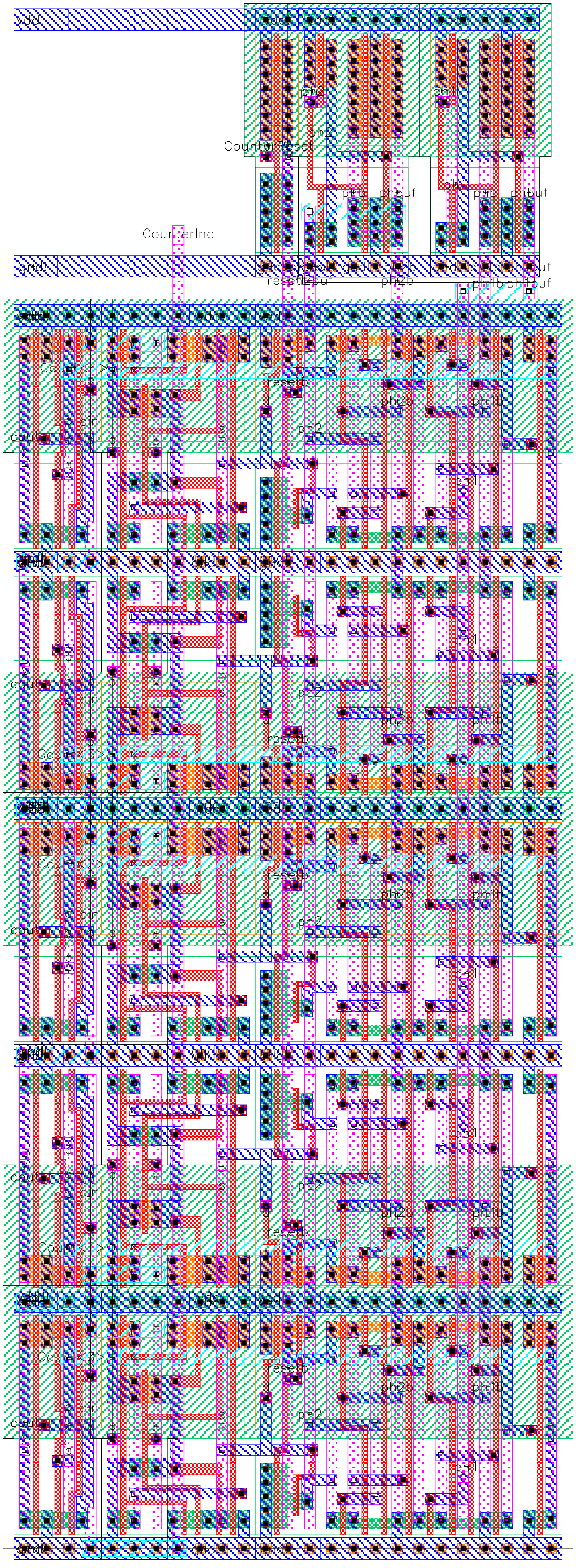
sha1.counter_zipper




vdd!
gnd!
CounterReset
ph1
ph2
phb
phbuf
ph2b
ph1b
ph1buf

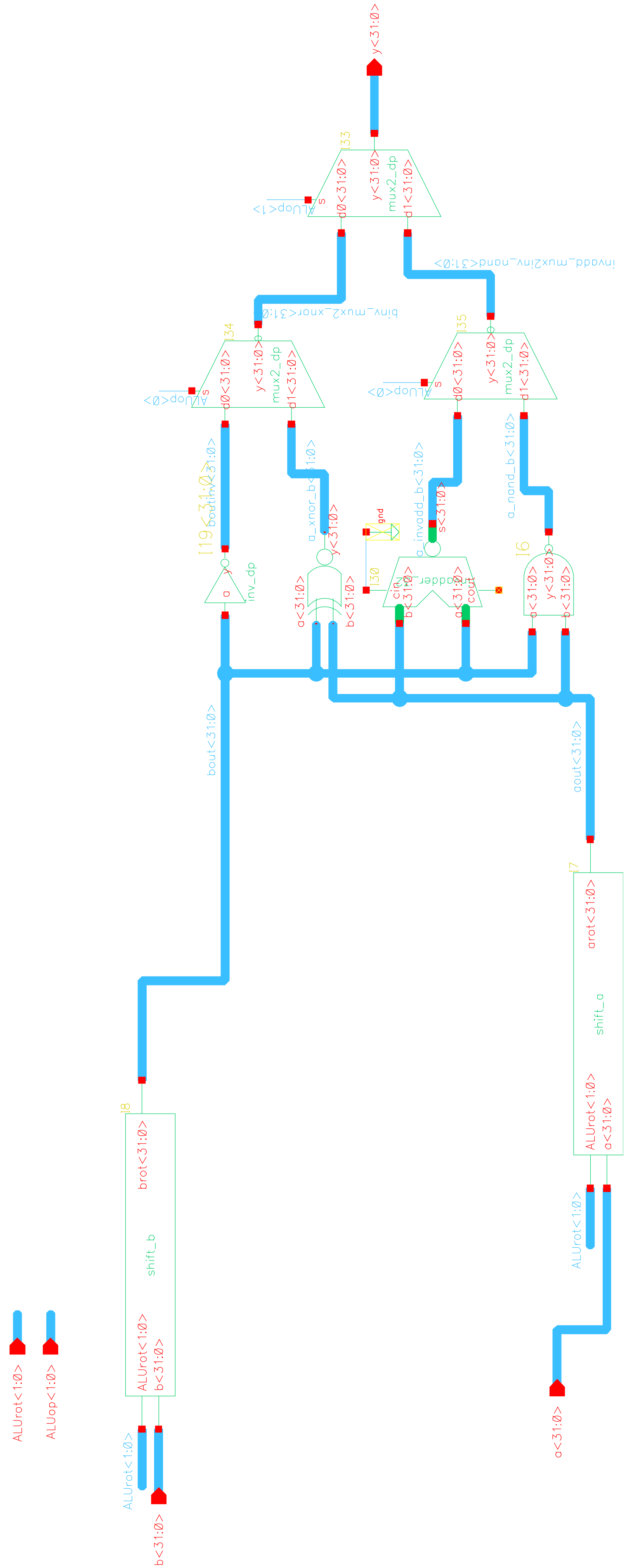
sha1.counter5



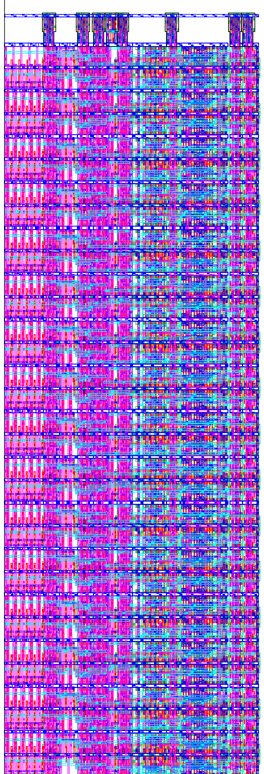


vdd!
CounterReset
CounterInc
gnd!
ph2
ph1
ph2b
ph1b
resetb
phibuf

sha1_alu.alu

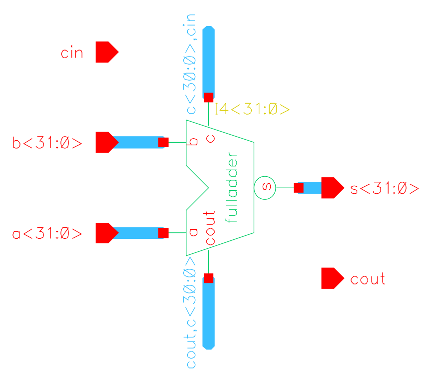


y<31:0>
mux2_dp
ALUop<1>
ALUop<0>
invadd_mux2inv_nand<31:0>
binv_mux2_xnor<31:0>
d0<31:0>
d1<31:0>
inv_dp
adder
gnd
a_xnor_b<31:0>
a_invadd_b<31:0>
a_nand_b<31:0>
boutinv<31:0>
bout<31:0>
aout<31:0>
arot<31:0>
shift_a
brot<31:0>
shift_b
ALUrot<1:0>
ALUop<1:0>
a<31:0>
b<31:0>



sha1_alu.invadder_32

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED
Apr 14 02:47:32 2010

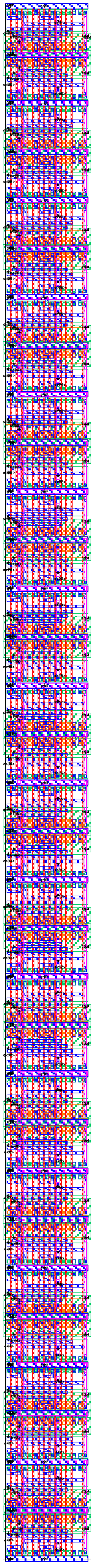
BY
wbuik

SIZE A

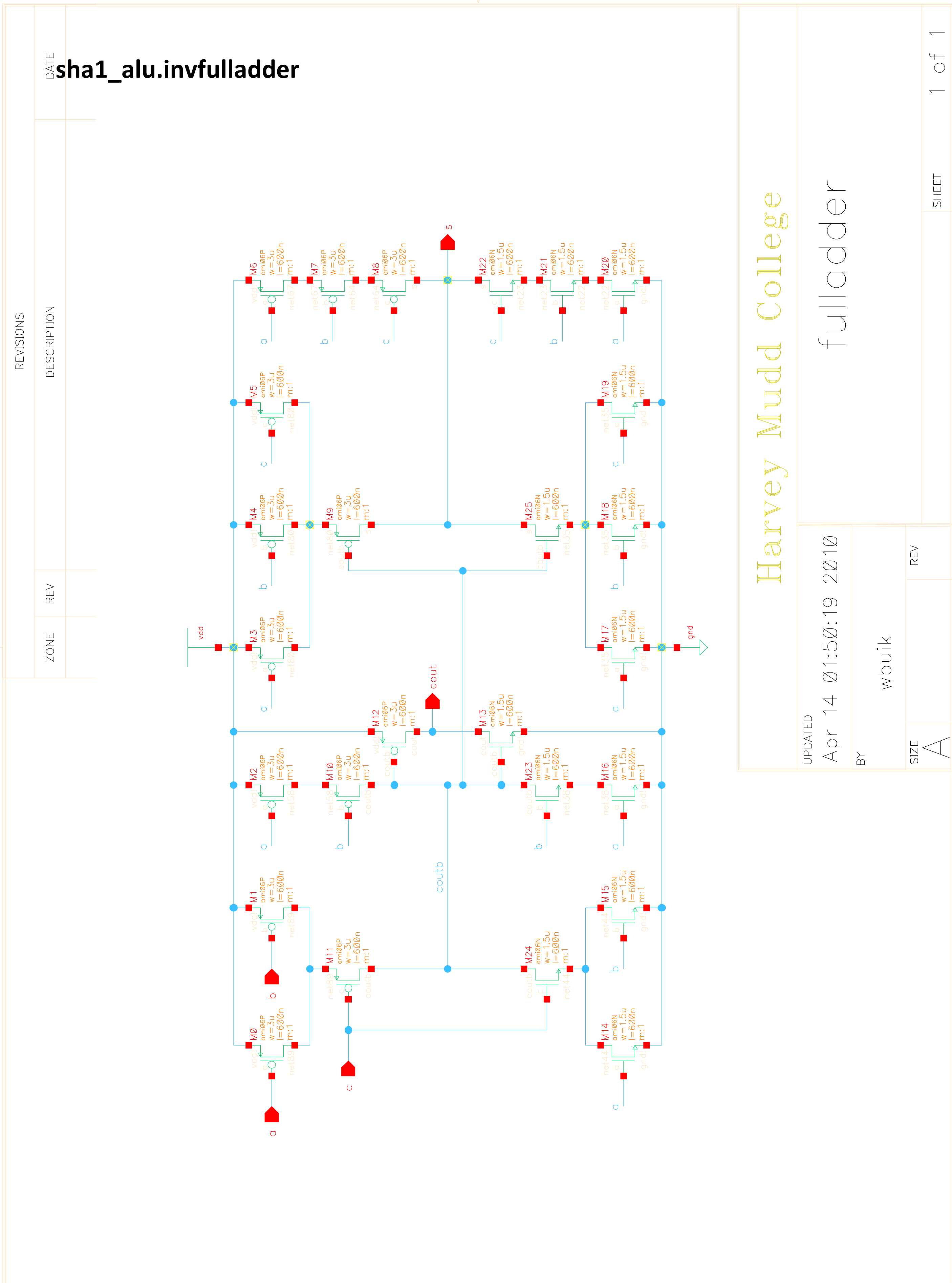
REV

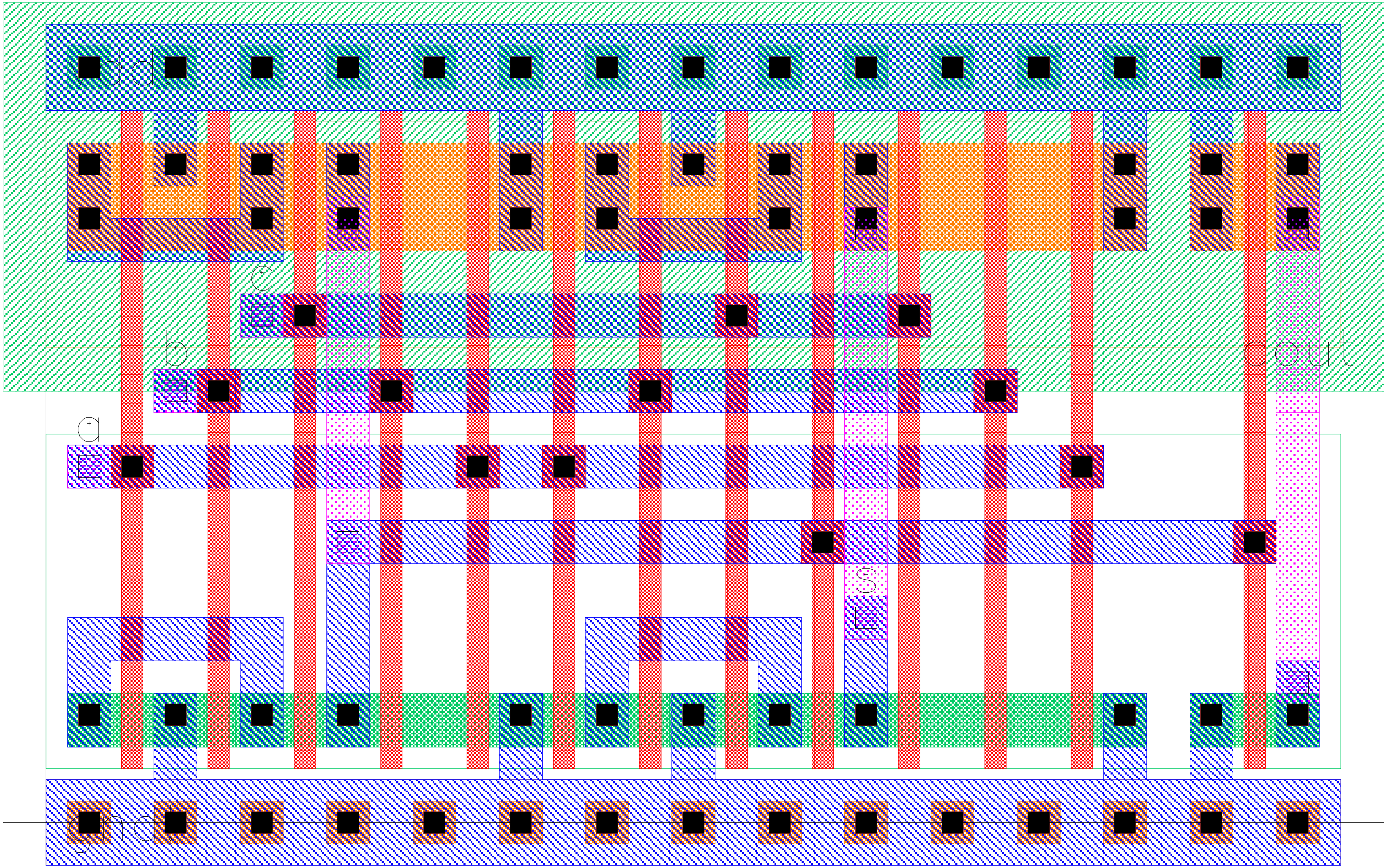
adder_8

SHEET 1 of 1

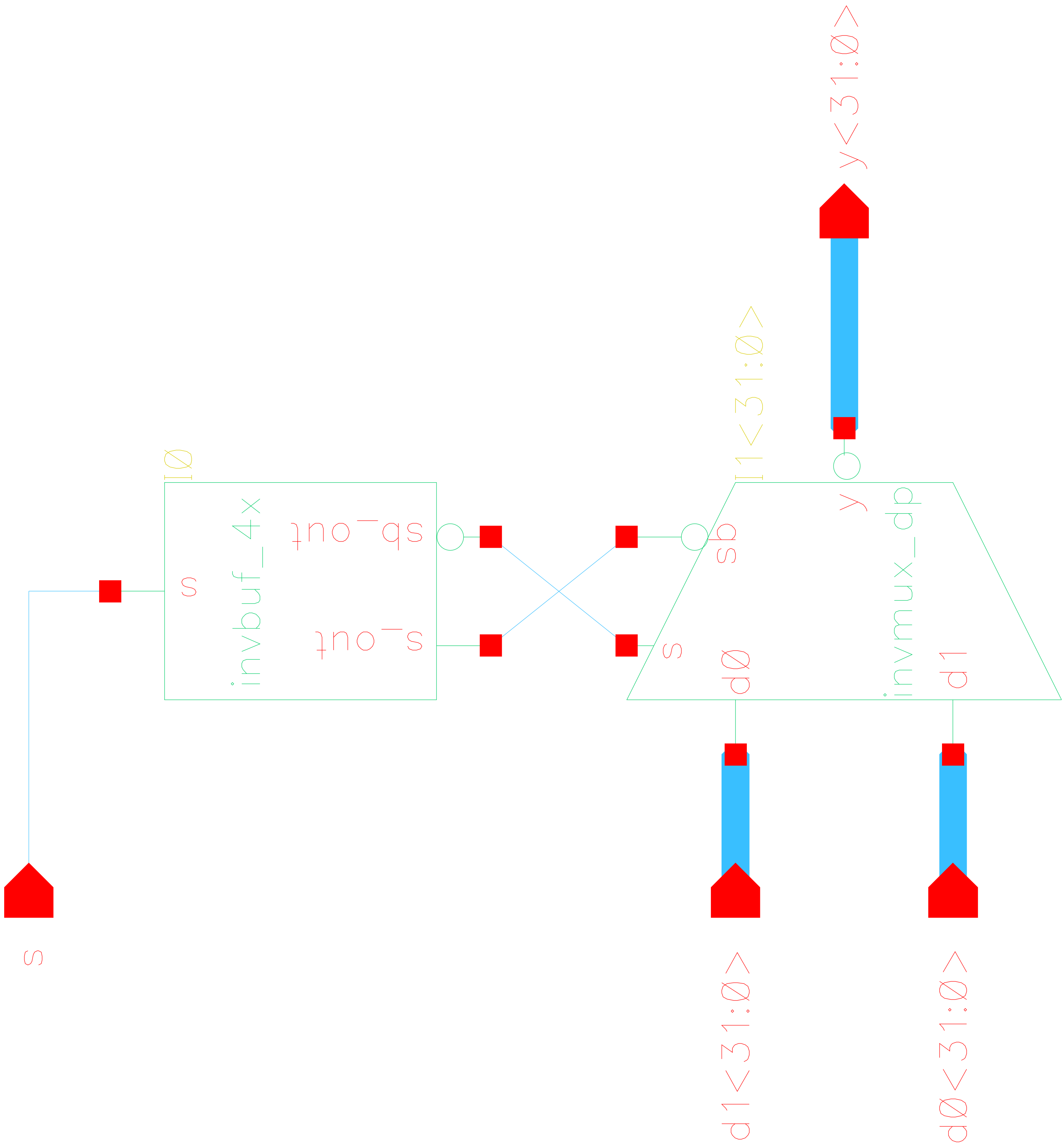


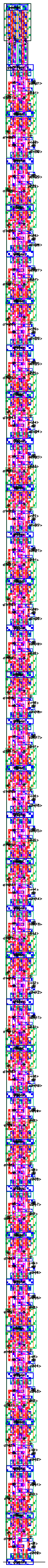
sha1_alu.invfulladder




c
b
a
s
cout

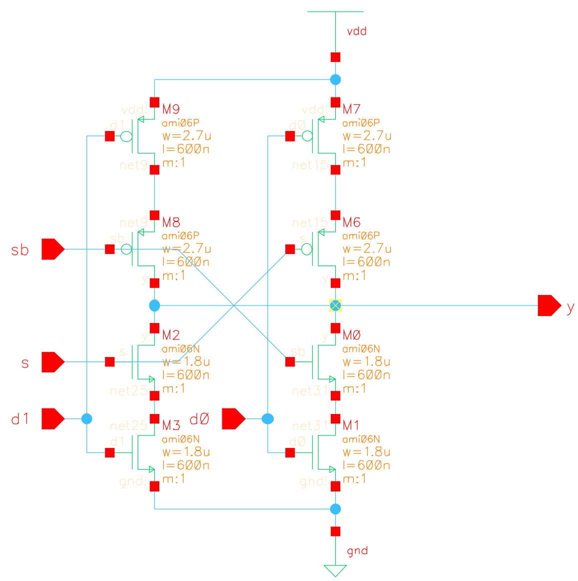
sha1_alu.invmux2_32





sha1_alu.invmux2_dp

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED Mar 29 23:52:23 2010

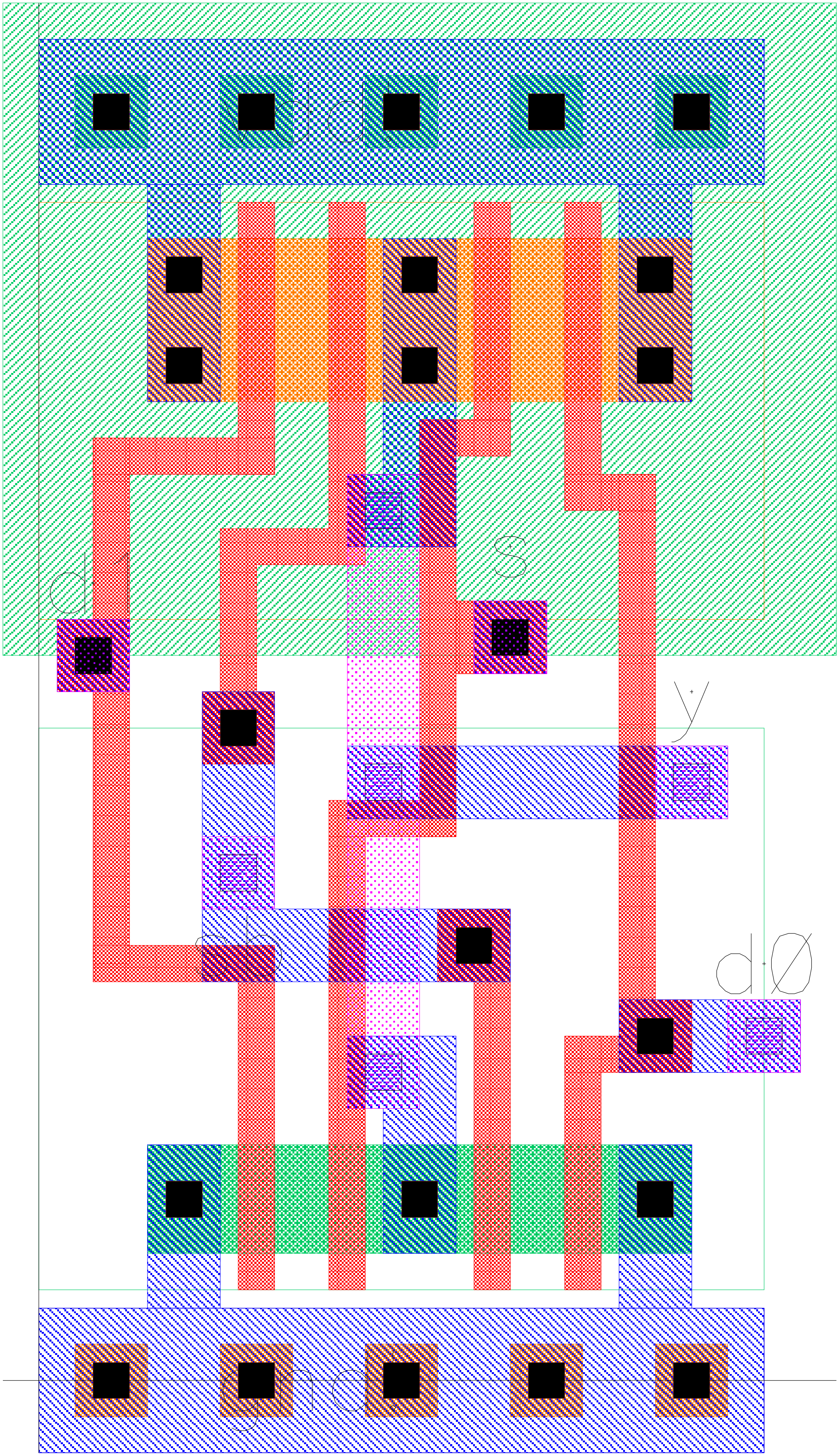
BY wbuik

mux2_dp_1x

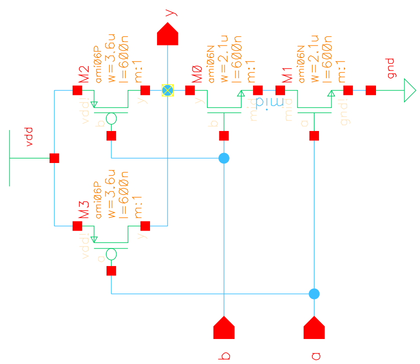
SIZE A

REV

SHEET 1 of 1


s
y
d0

sha1_alu.nand2_1x



REVISIONS

ZONE	REV	DESCRIPTION	DATE

Harvey Mudd College

nand2_1x

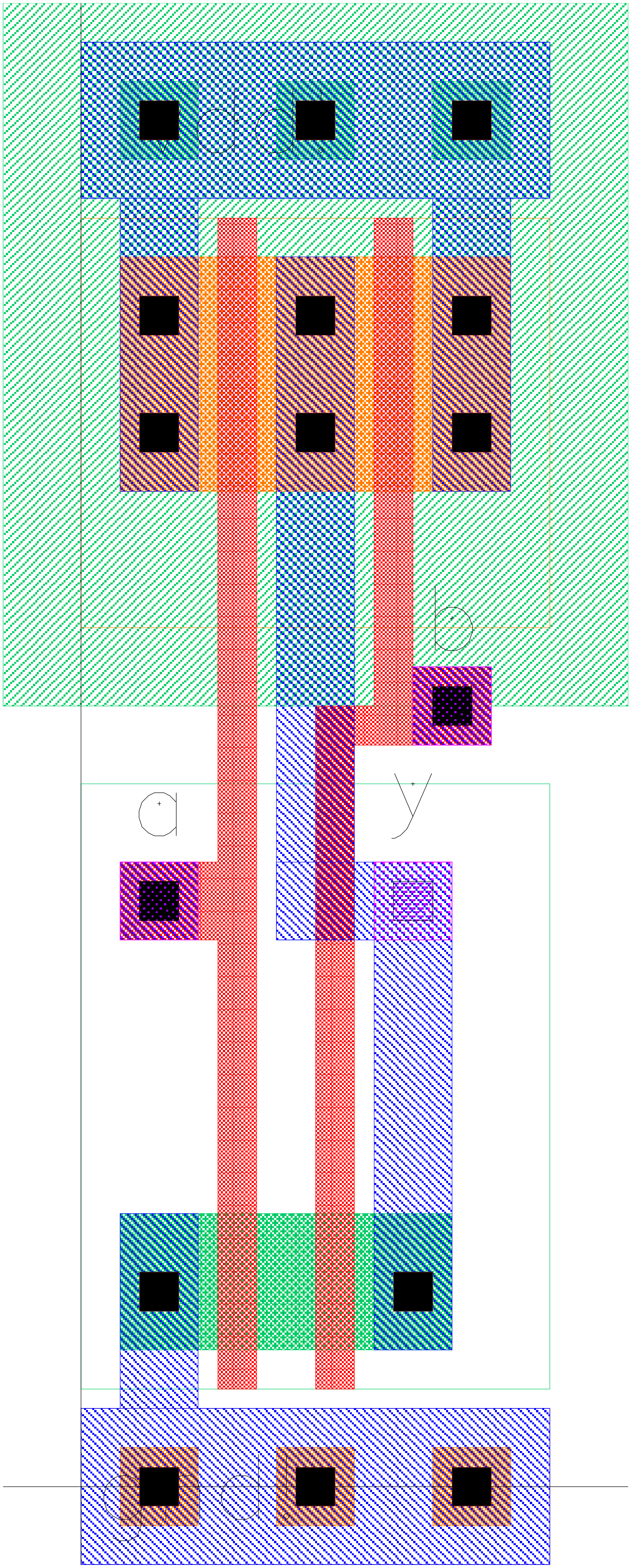
SHEET 1 of 1

UPDATED Apr 14 02:07:56 2010

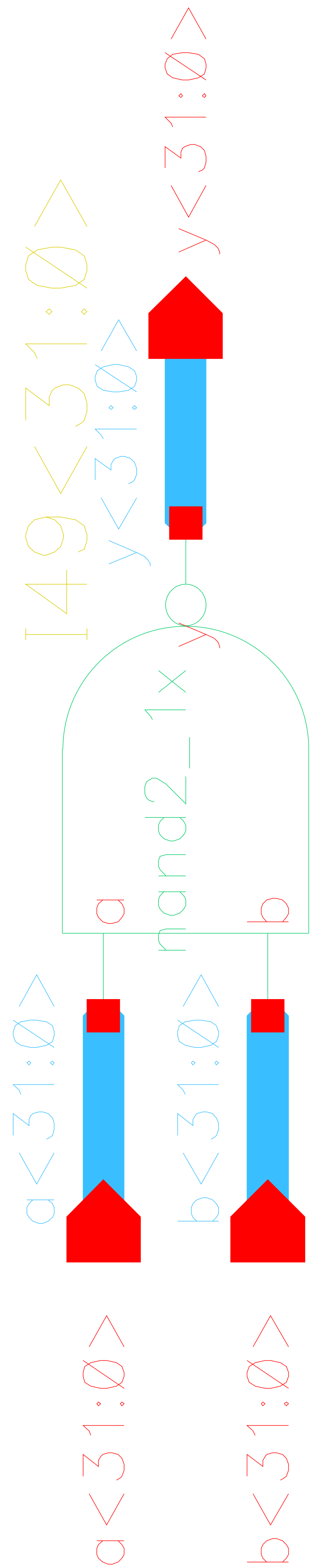
BY wbuik

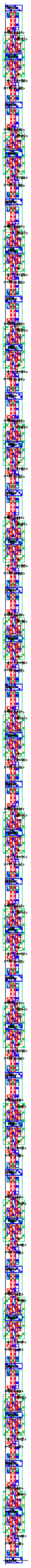
SIZE A

REV

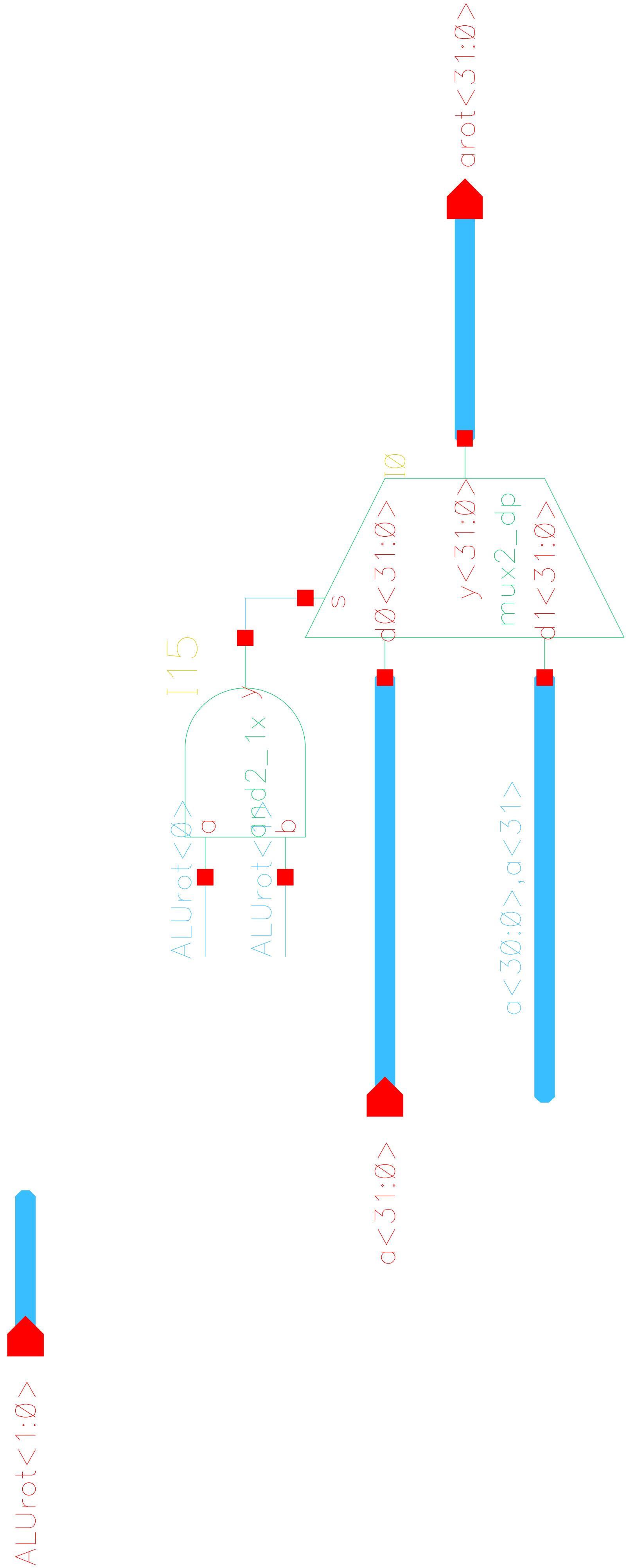

vdd
b
a
y
gnd

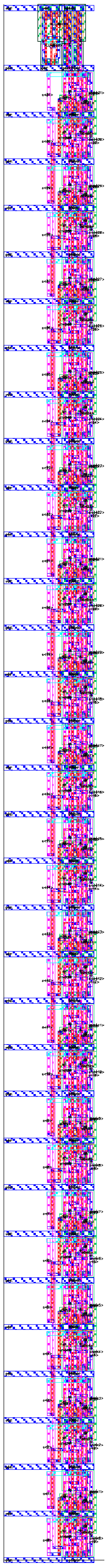
sha1_alu.nand2_32x



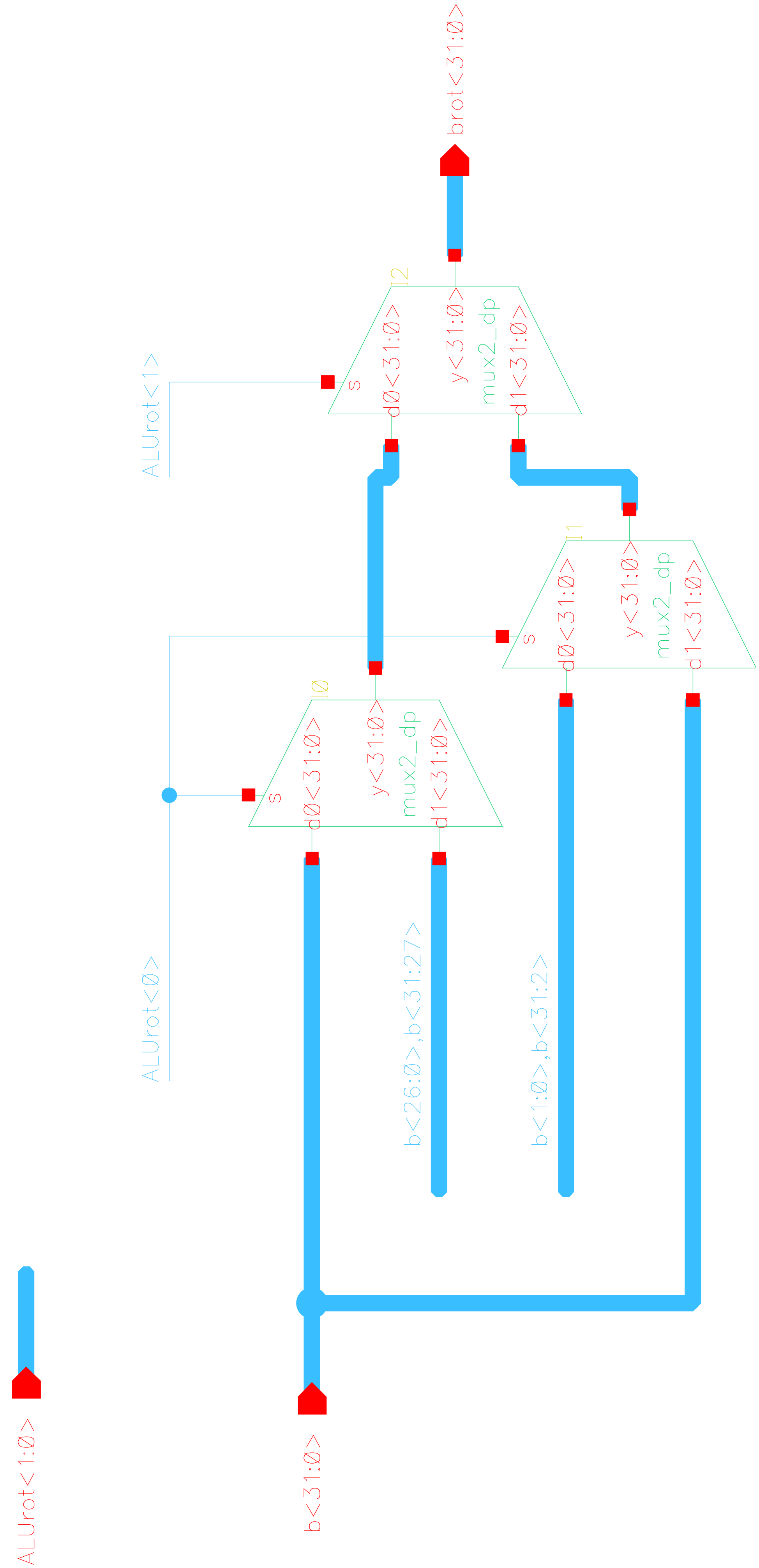


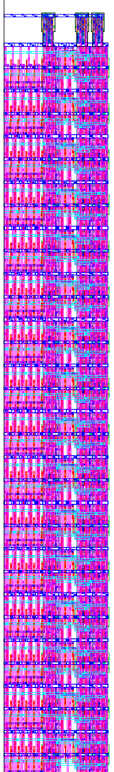
sha1_alu.shift_a



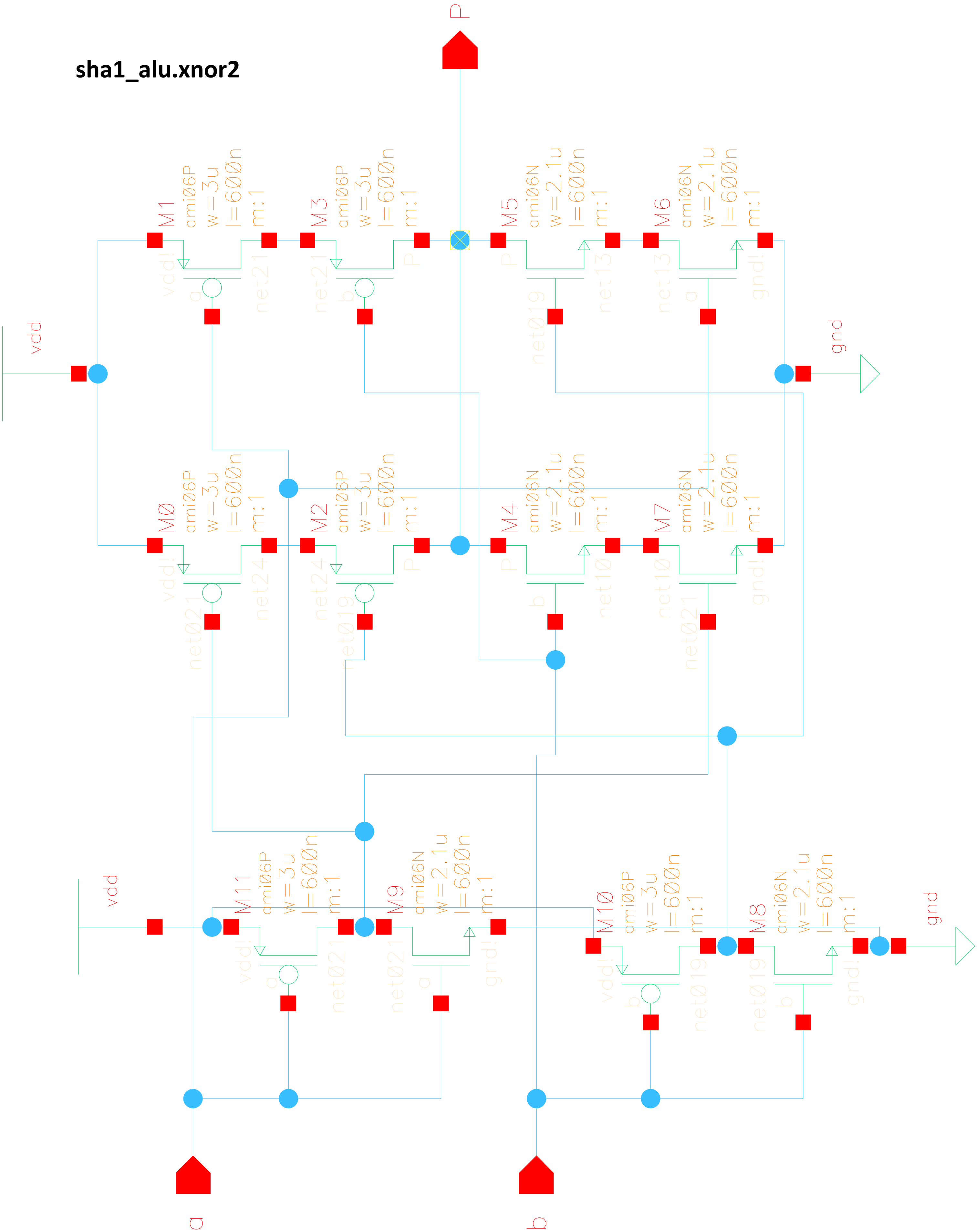


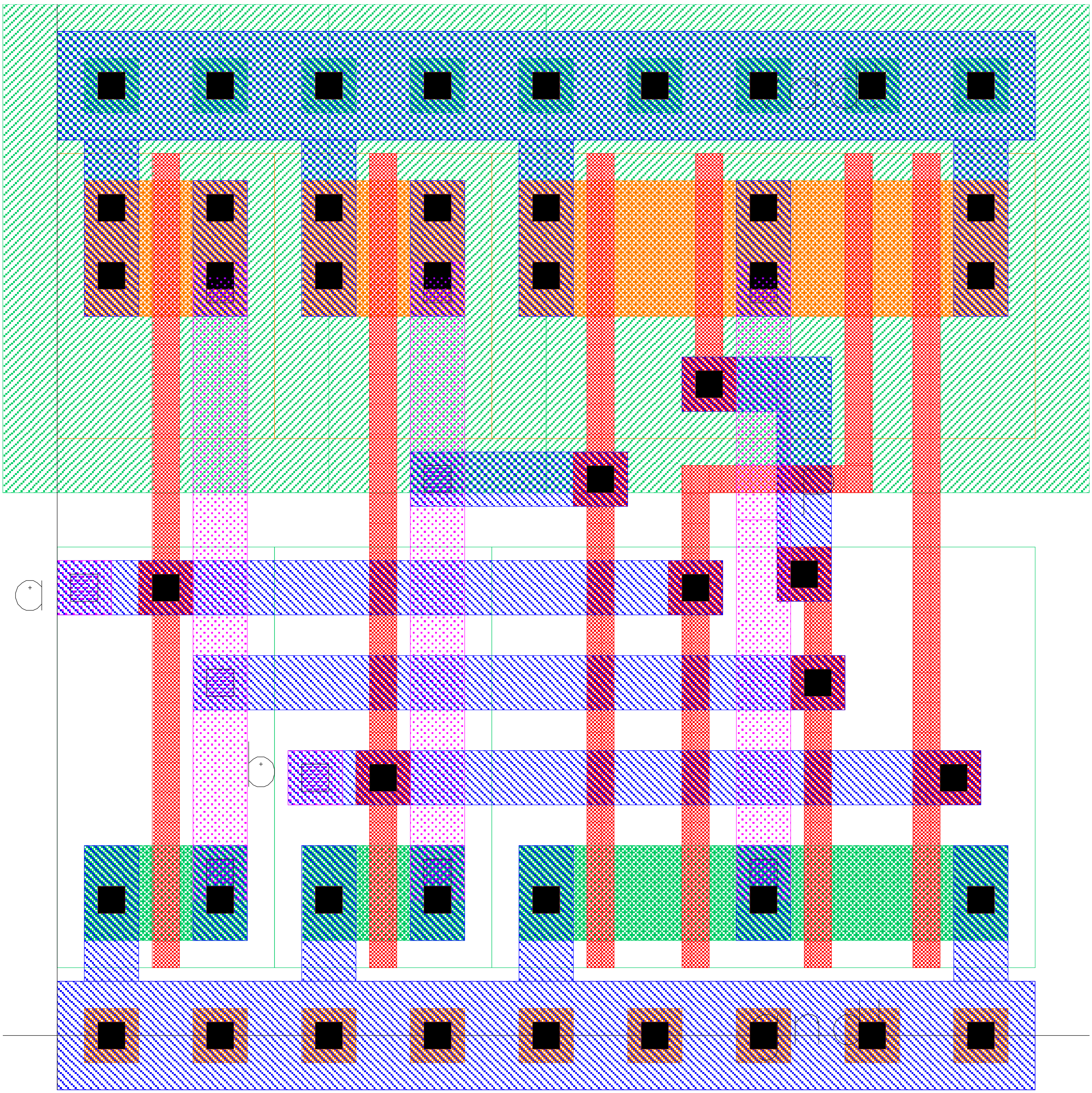
sha1_alu.shift_b



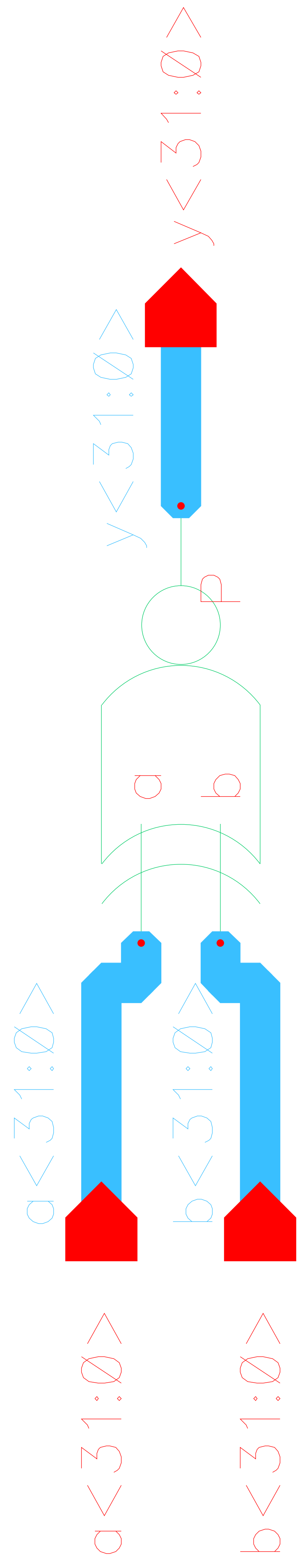


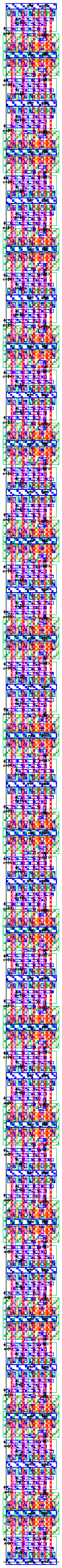
sha1_alu.xnor2



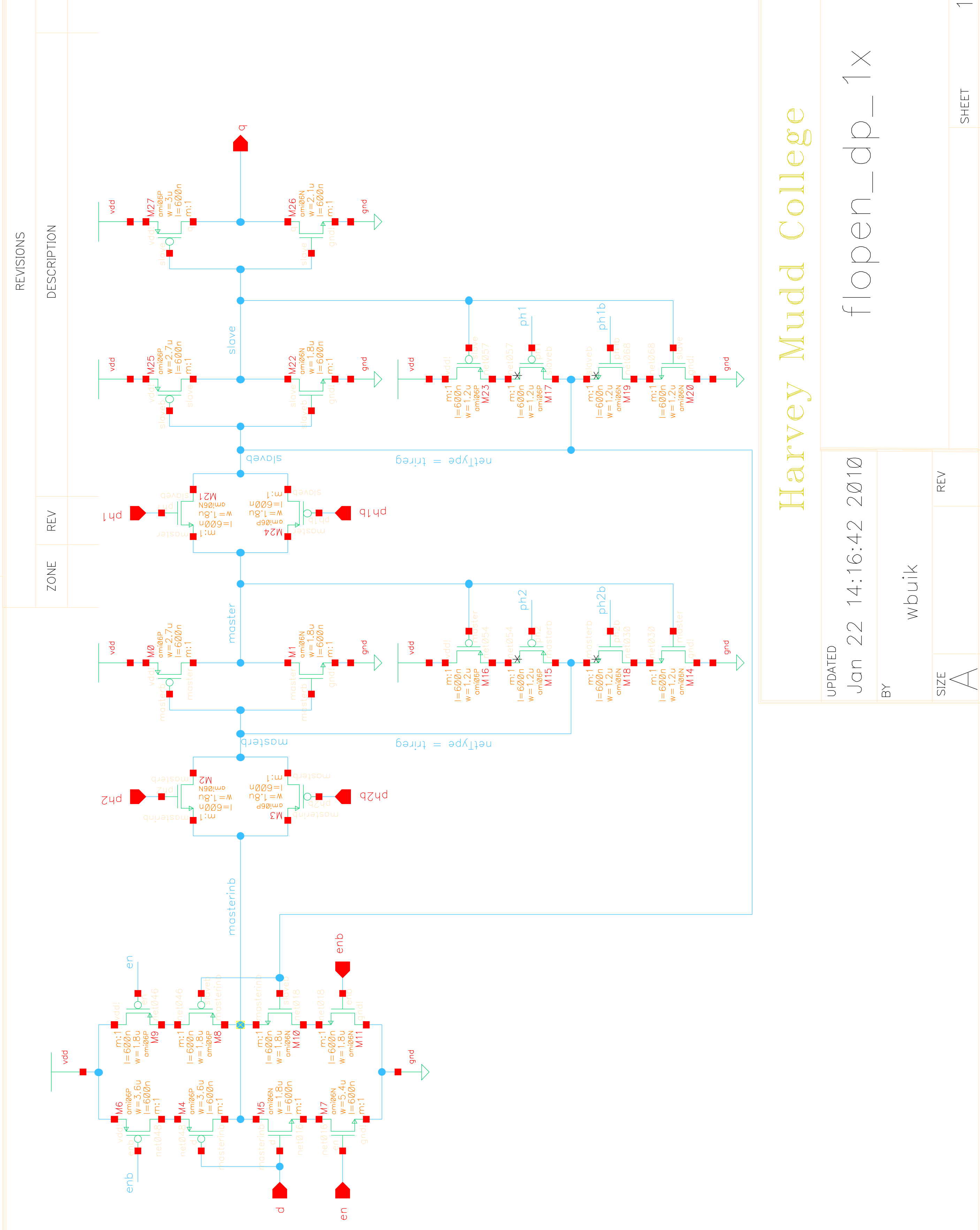


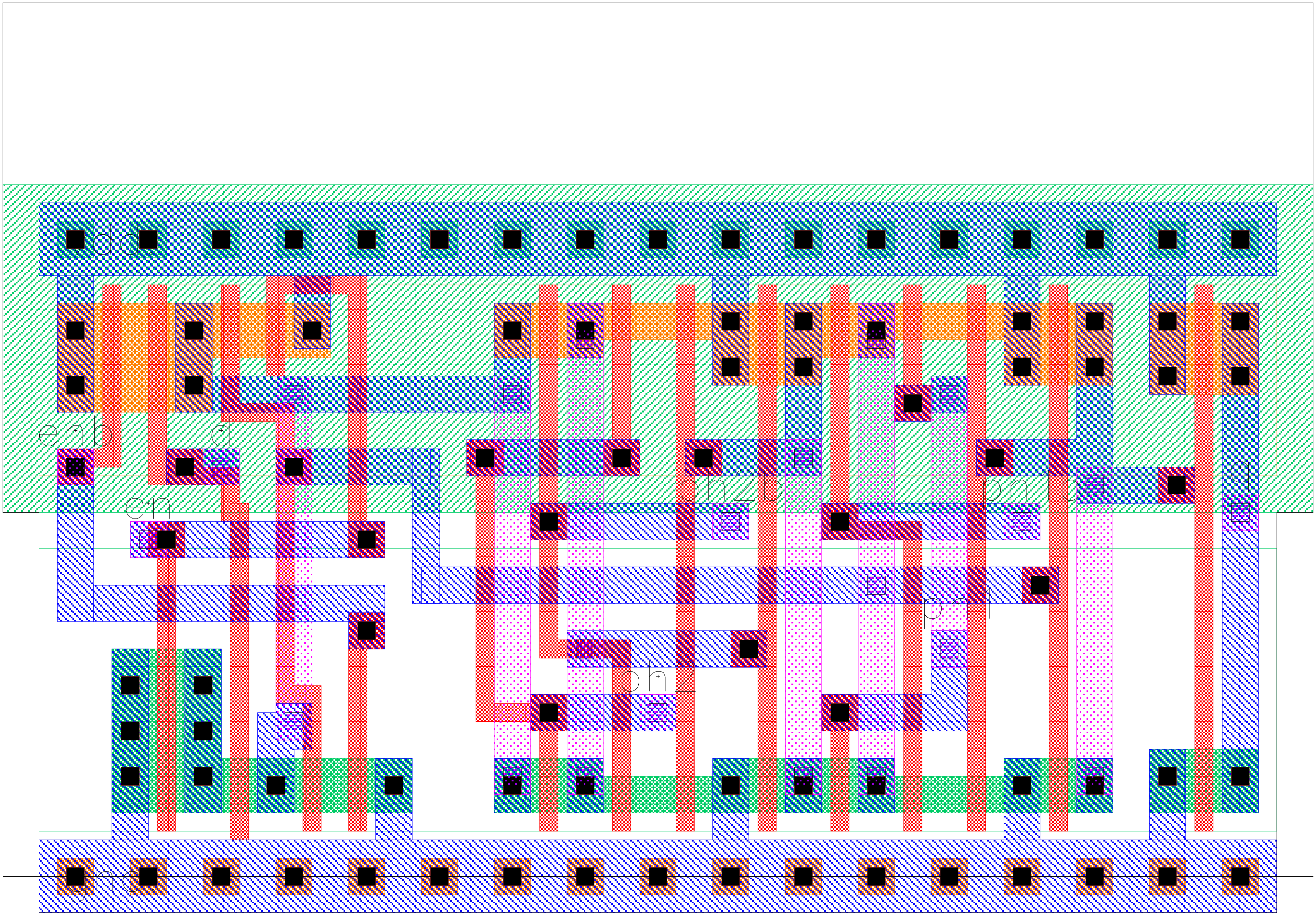
sha1_alu.xnor2_32x



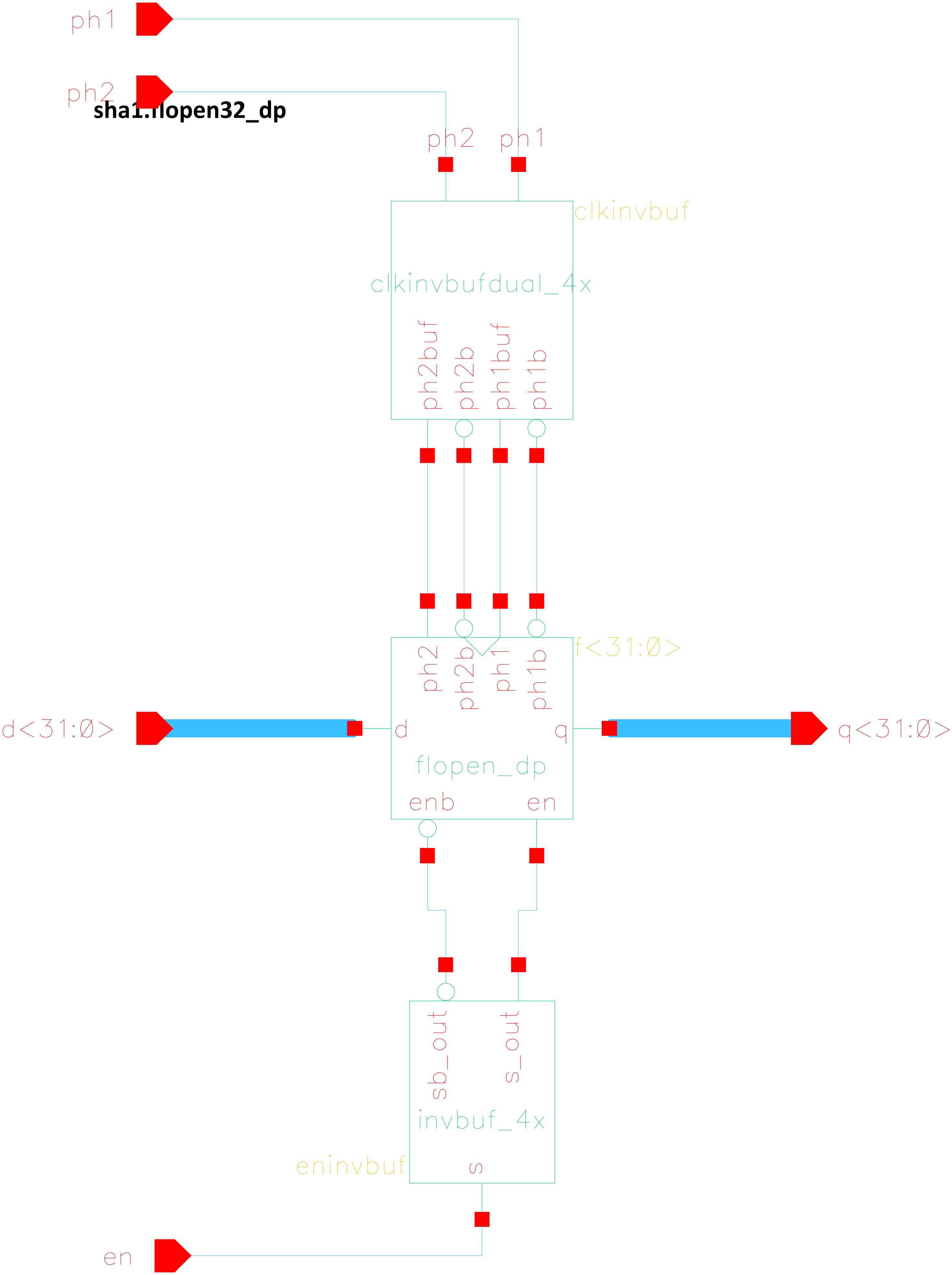


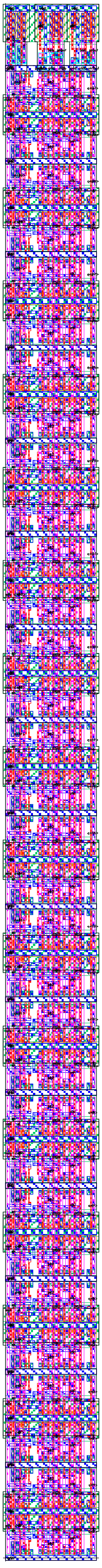
sha1.flopen_dp



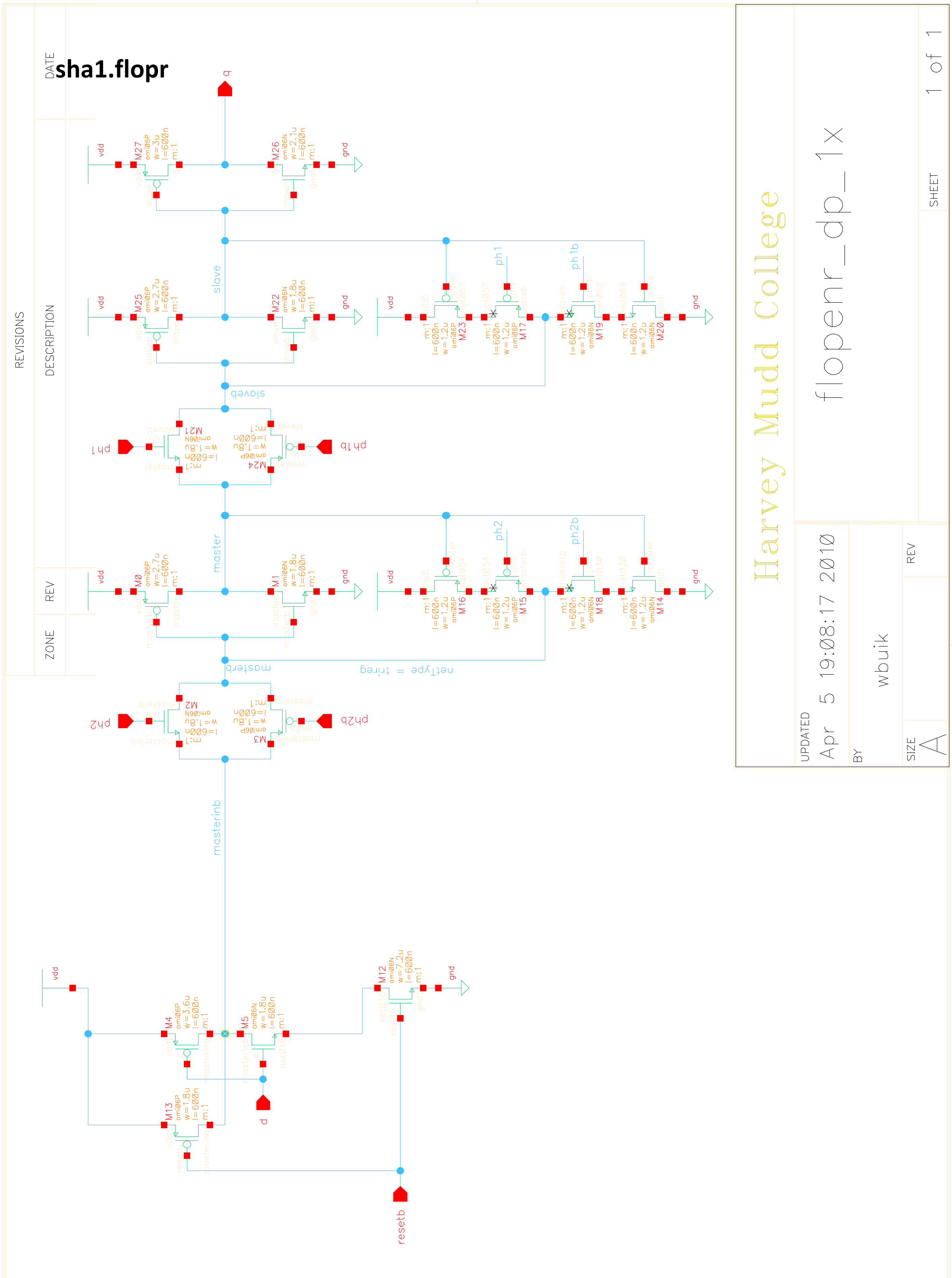

enb
d
en
ph2b
ph1b
ph1
ph2

sha1.flopen32_dp

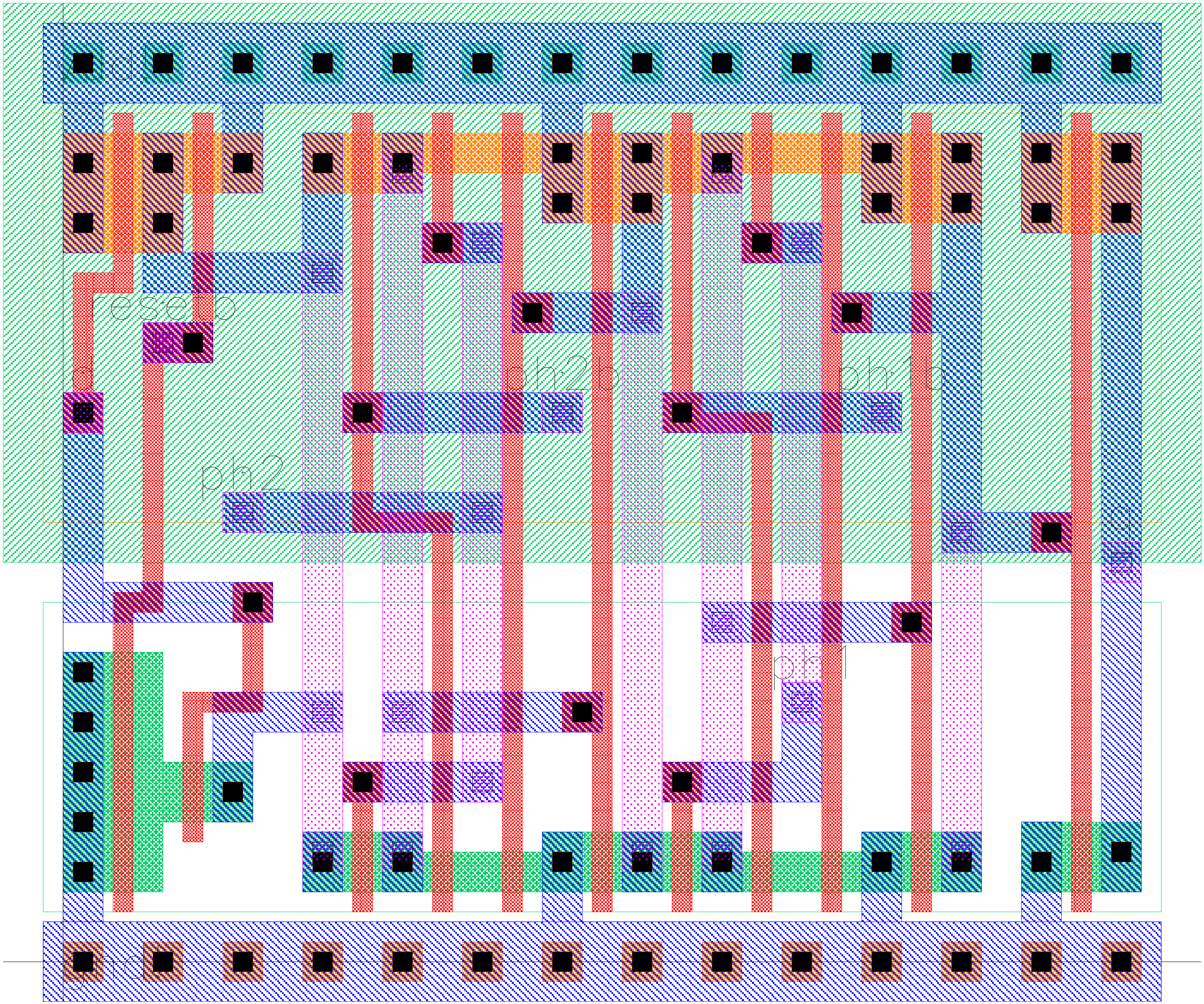




sha1.flopr

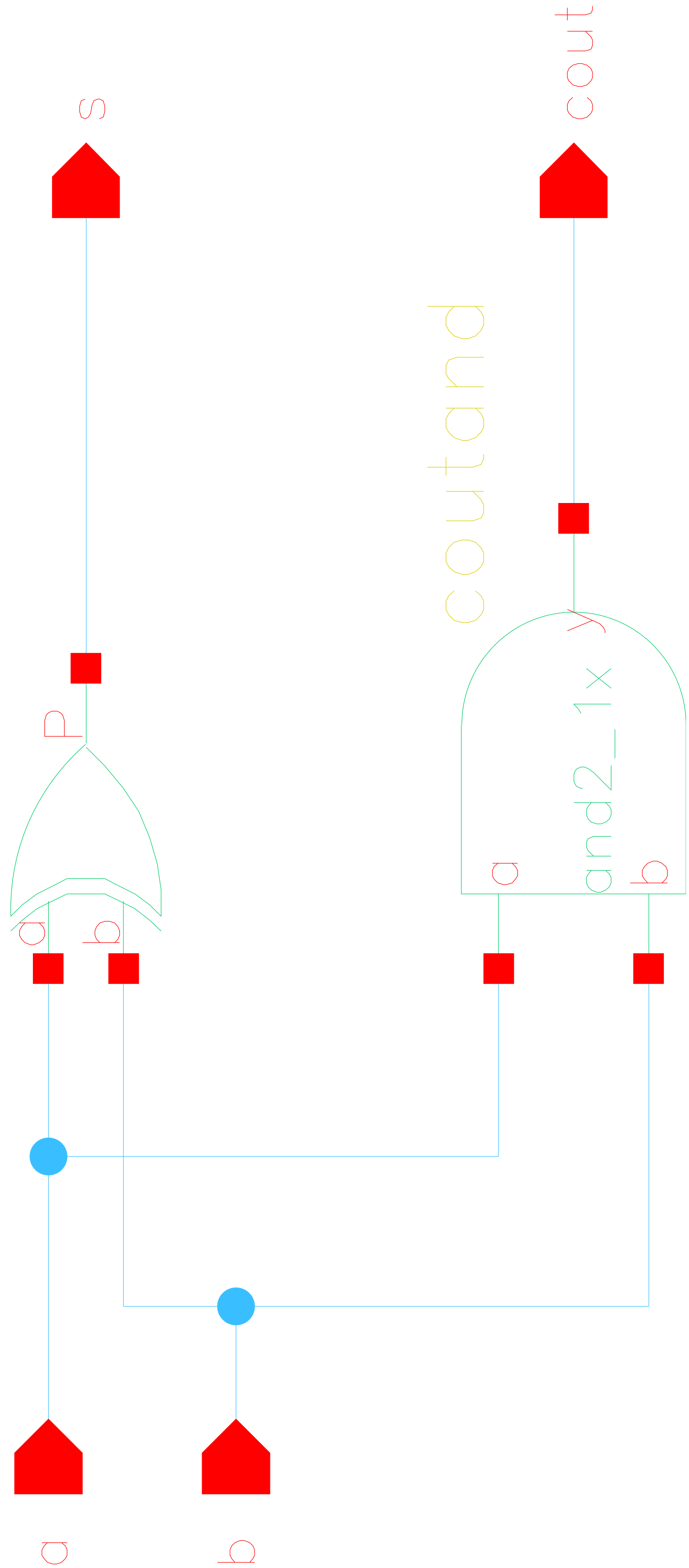


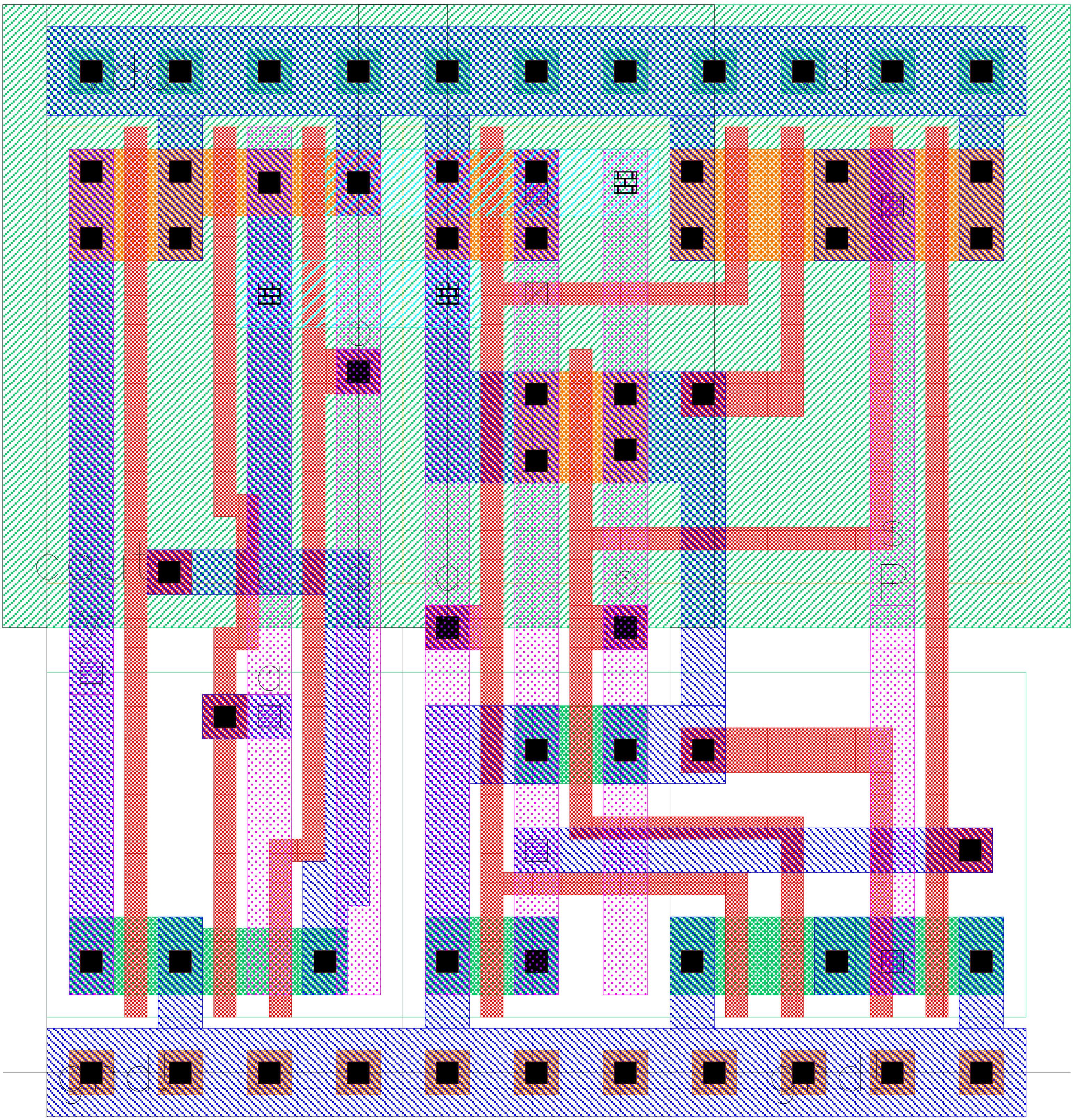
REVISIONS
ZONE
REV
DESCRIPTION
DATE
q
vdd
gnd
slave
slaveb
master
masterb
masterinb
netType = triReg
ph1
ph1b
ph2
ph2b
d
resetb
M0
M1
M2
M3
M4
M5
M12
M13
M14
M15
M16
M17
M18
M19
M20
M21
M22
M23
M24
M25
M26
M27
Harvey Mudd College
flopenr_dp_1x
UPDATED
Apr 5 19:08:17 2010
BY
wbuik
SIZE
A
REV
SHEET
1 of 1



reset_b
d
ph2
ph2b
ph1b
ph1
q

sha1.halfadder_dp

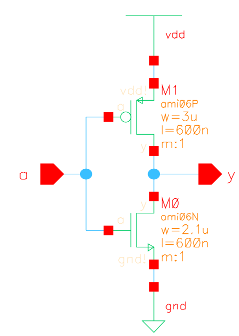




sha1.inv_dp

REVISIONS

ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED
Mar 29 23:52:00 2010

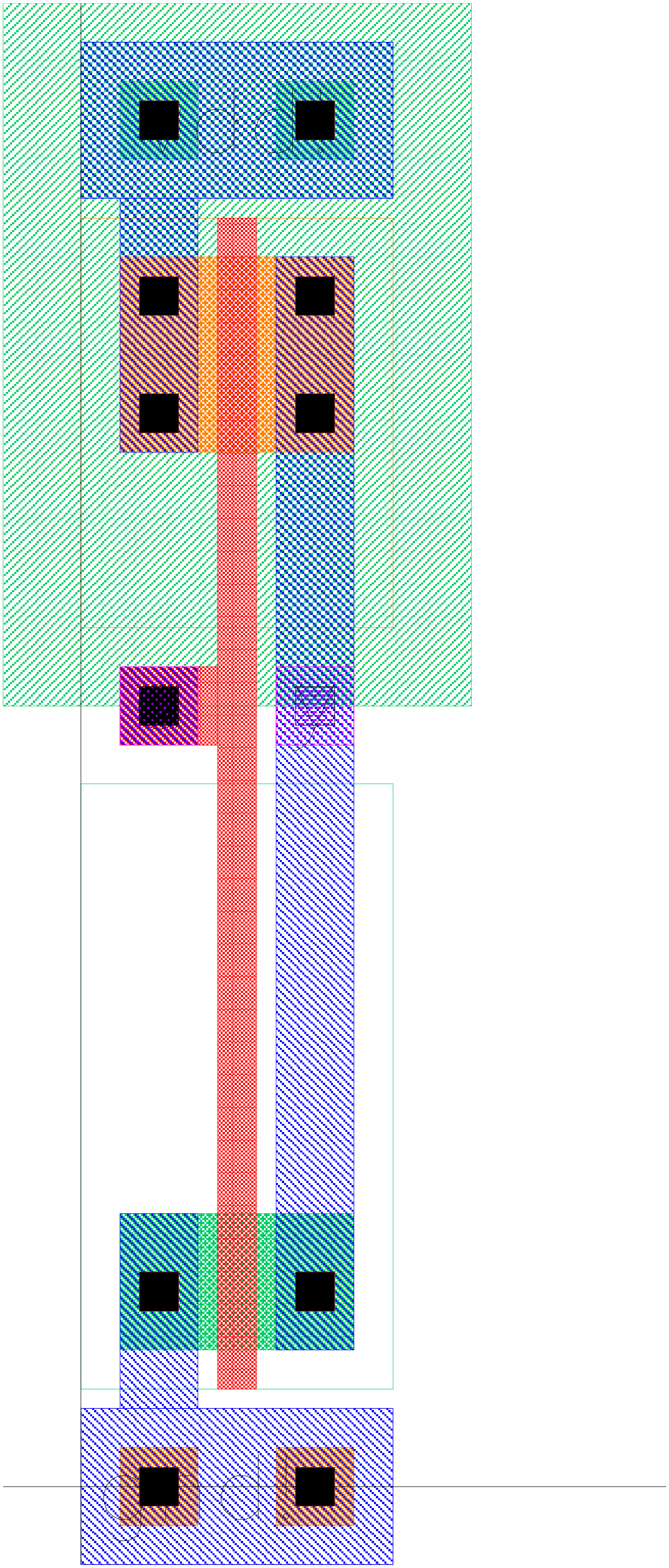
inv_1x

BY
wbuik

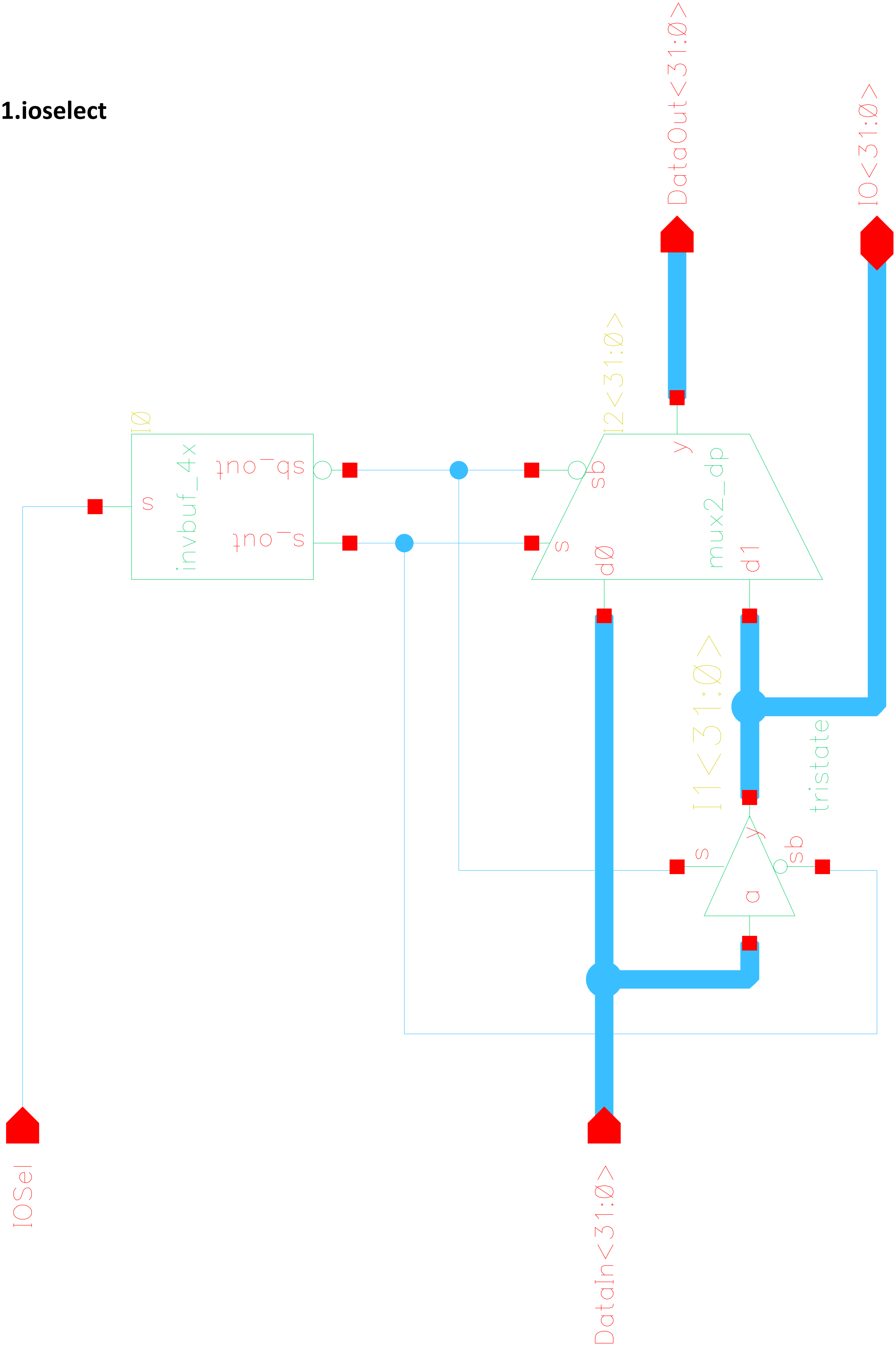
SIZE
A

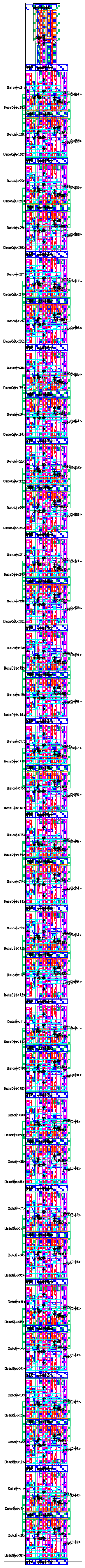
REV

SHEET
1 of 1

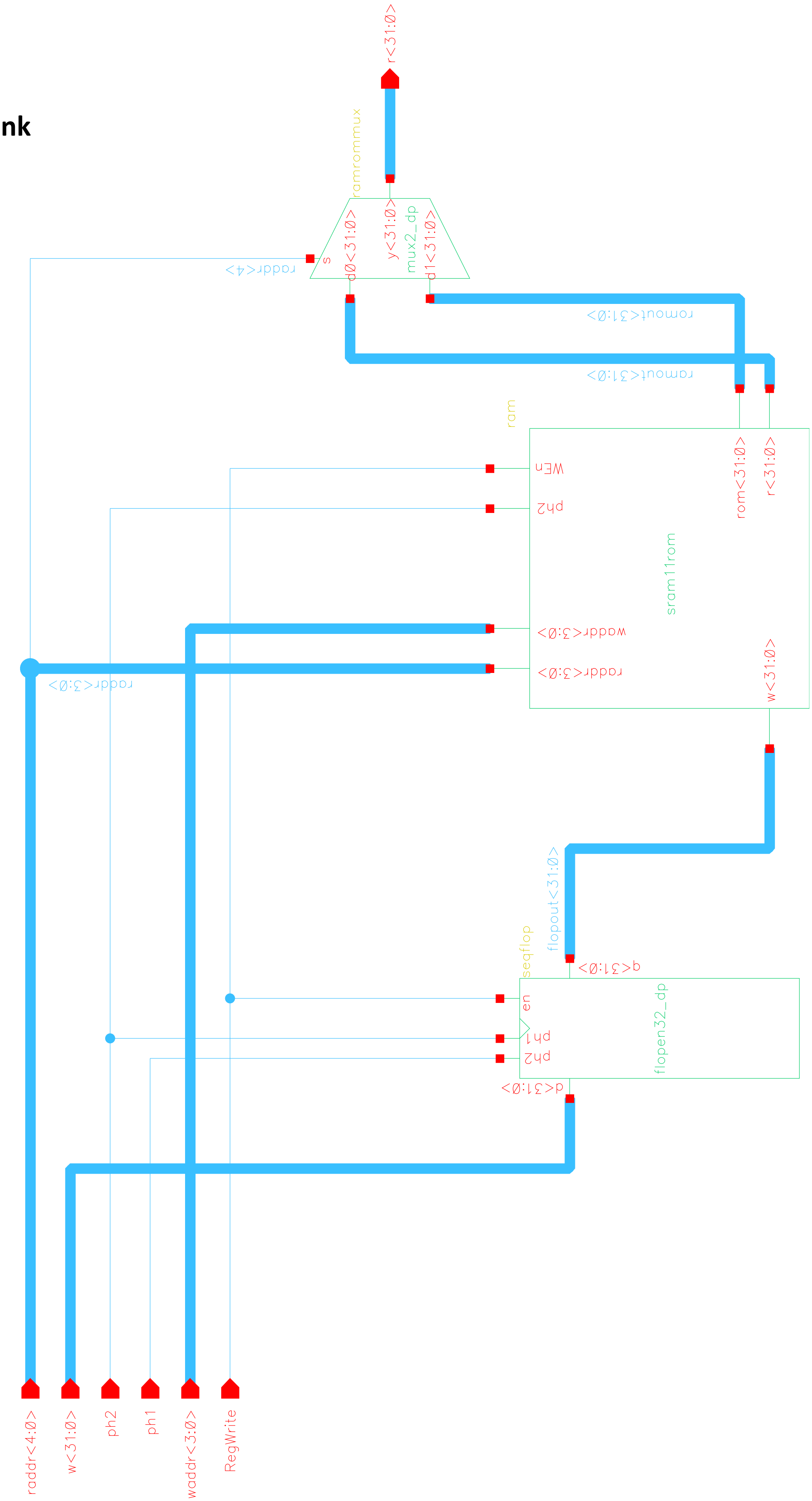

vdd!
y
gnd!

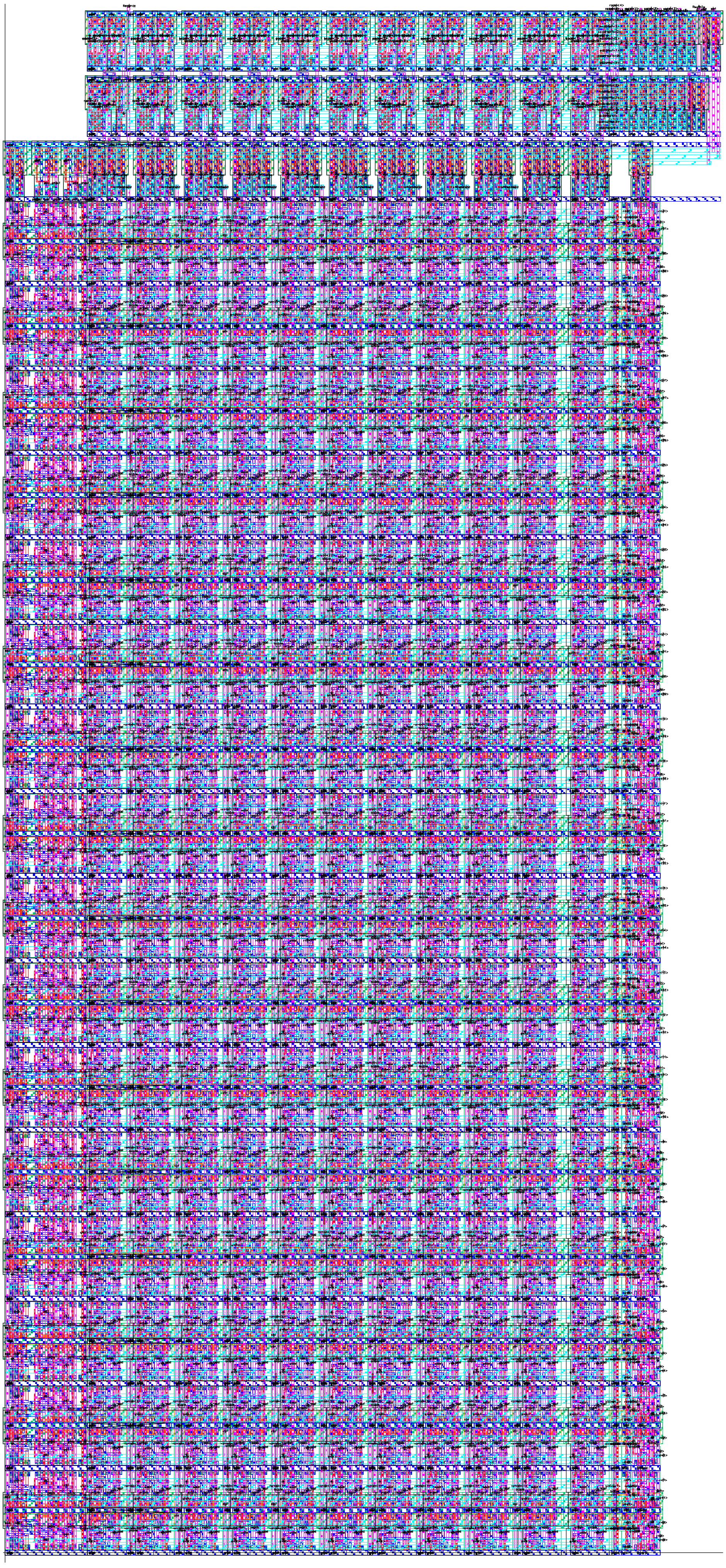
sha1.ioselect



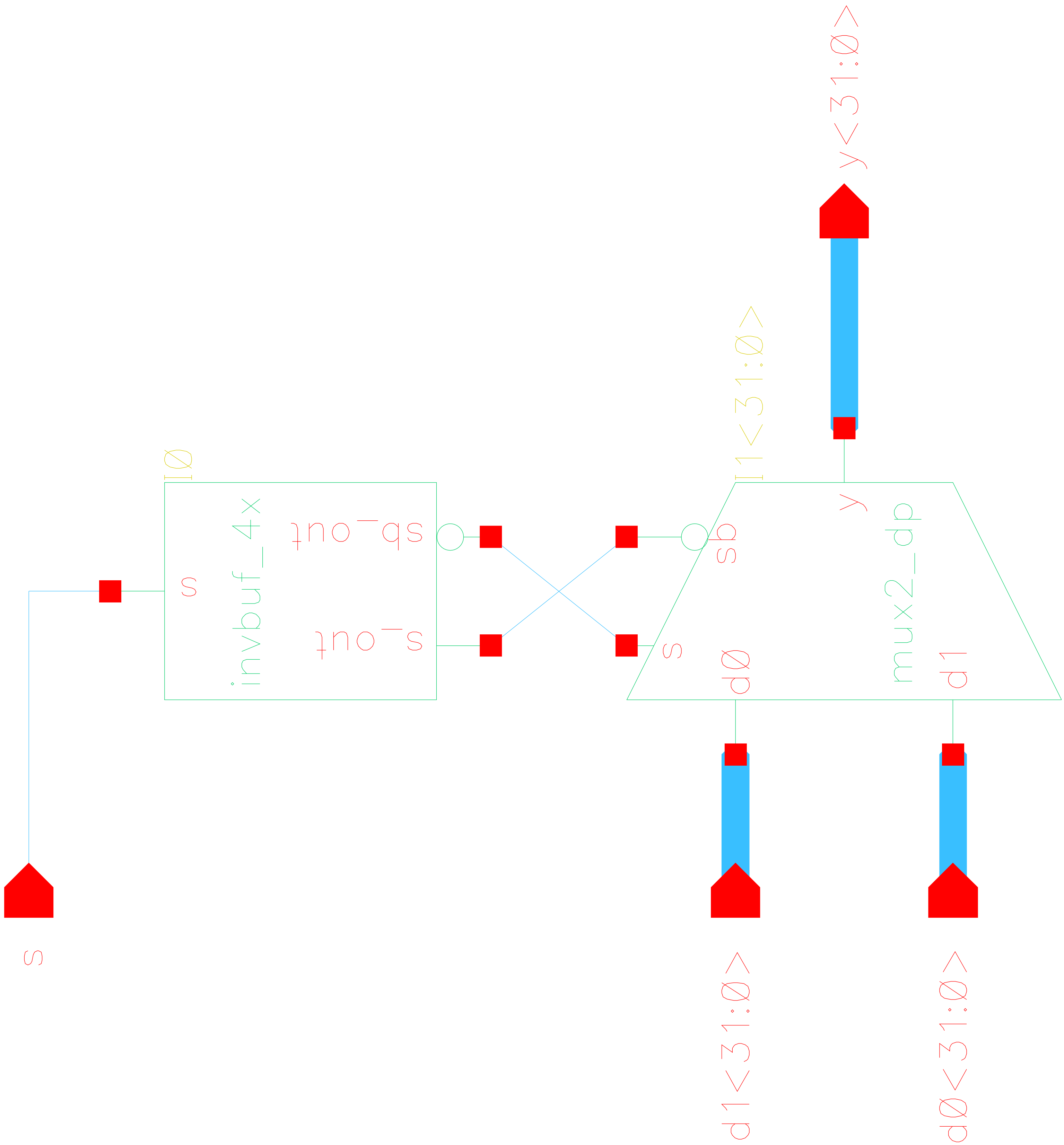


sha1.memconstbank

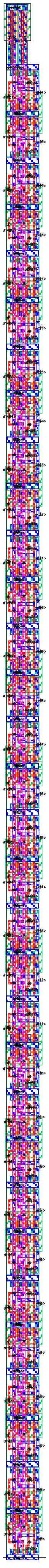




sha1.mux2_32



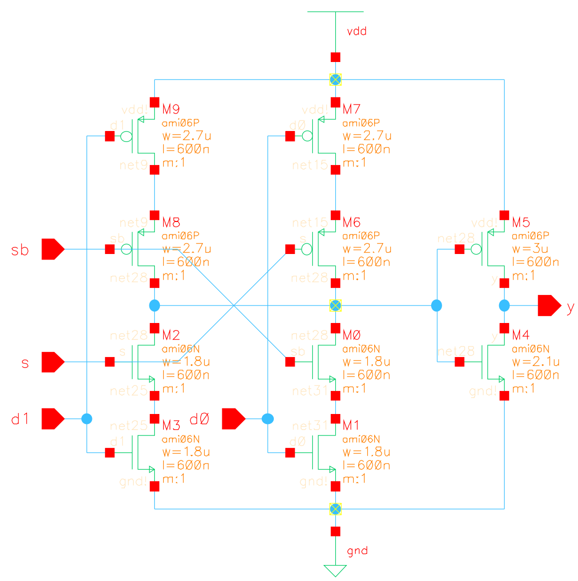
I0
invbuf_4x
s
sb_out
s_out
I1<31:0>
mux2_dp
sb
s
y
d0
d1
y<31:0>
s
d1<31:0>
d0<31:0>



sha1.mux2_dp

ZONE	REV	DESCRIPTION	DATE

REVISIONS



Harvey Mudd College

UPDATED
Mar 29 23:52:45 2010

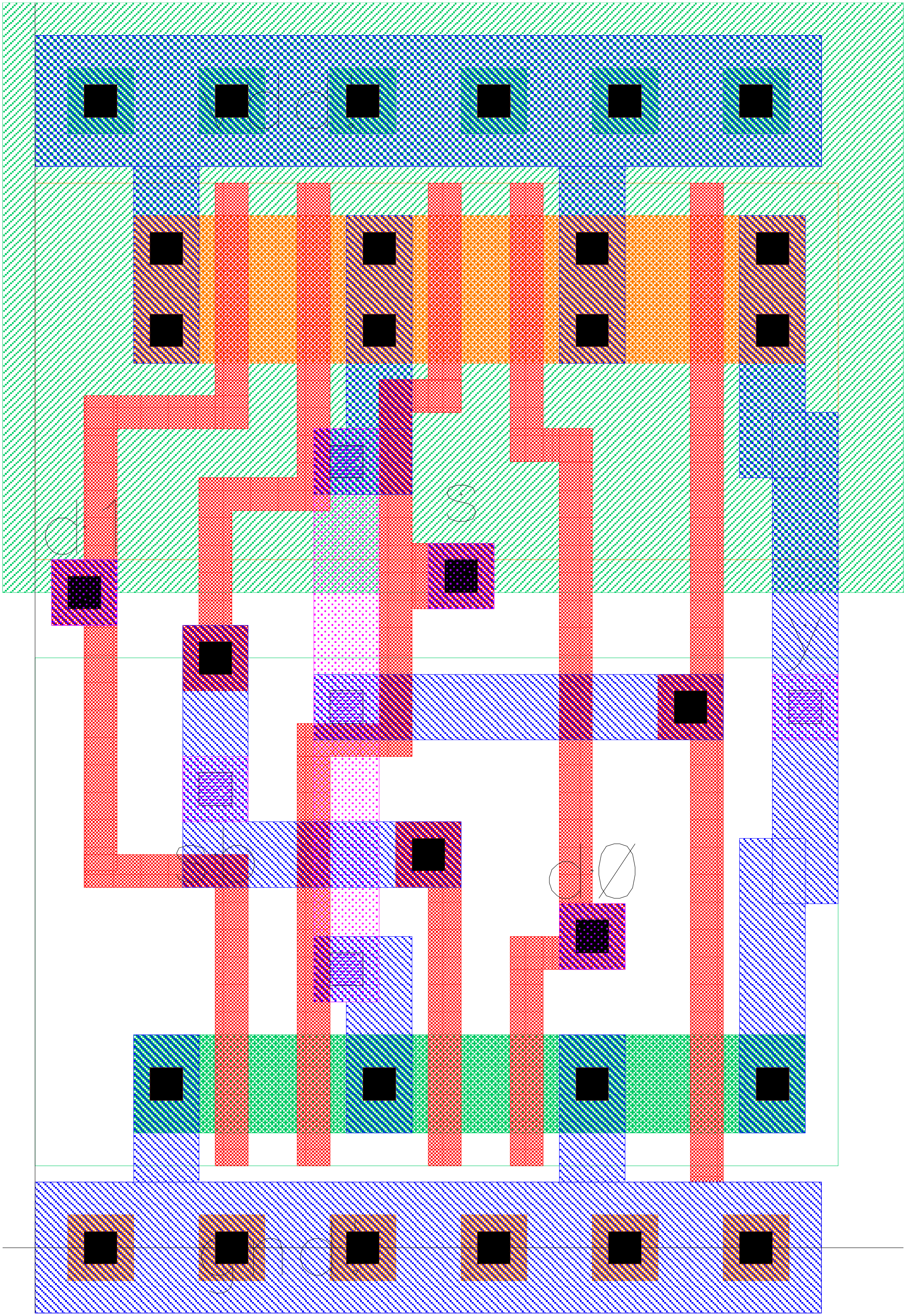
BY
wbuik

mux2_dp_1x

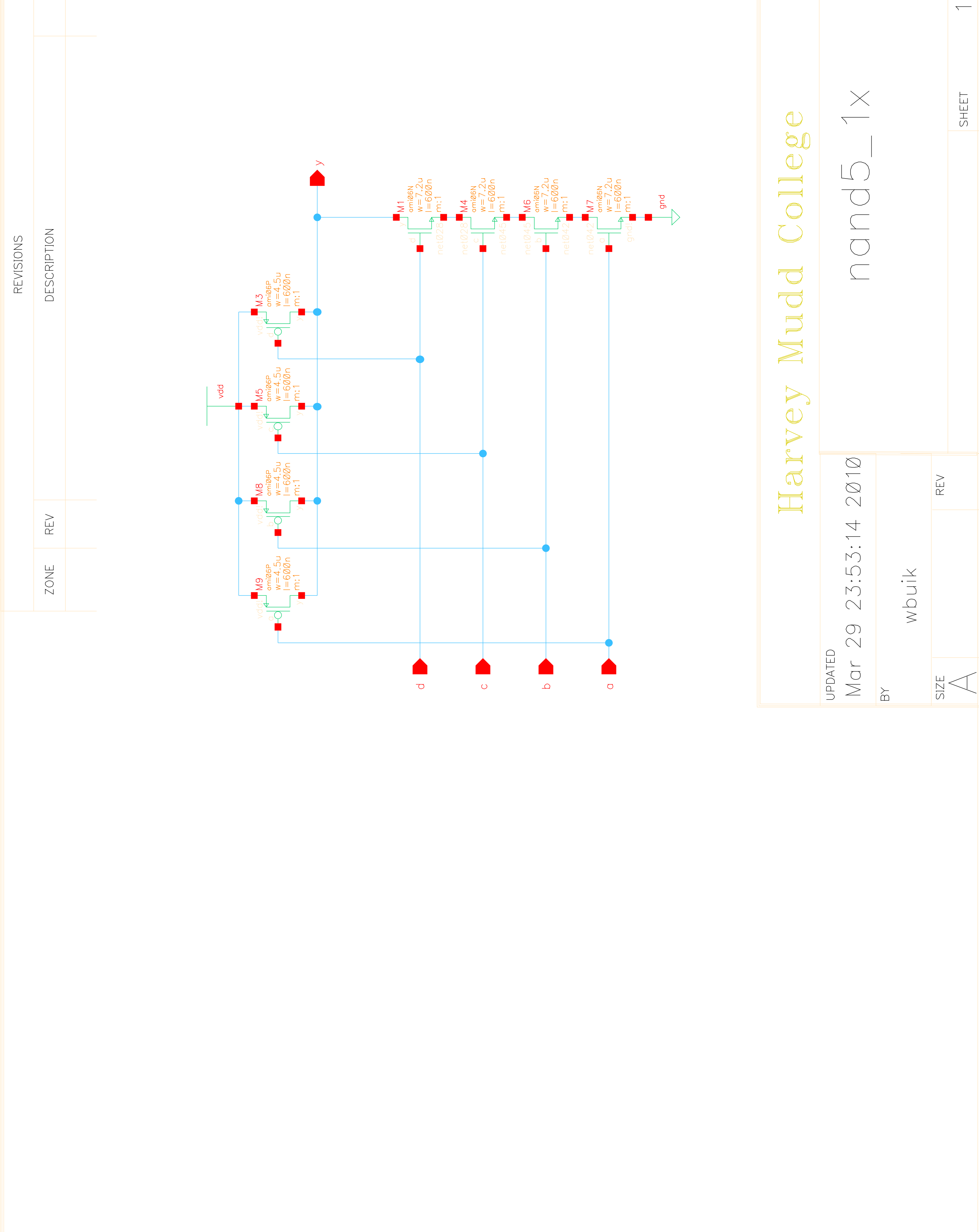
SIZE
A

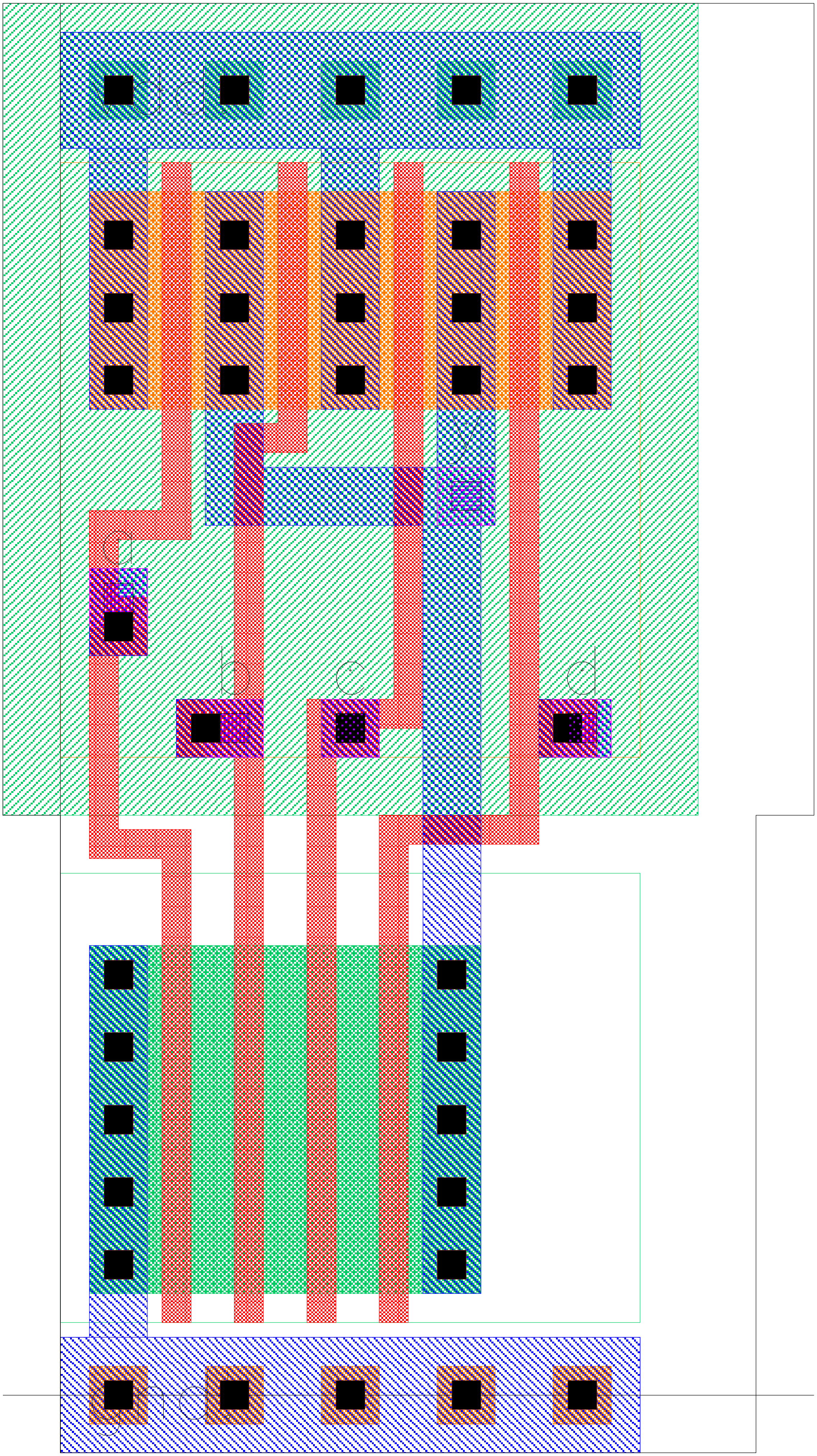
REV

SHEET 1 of 1

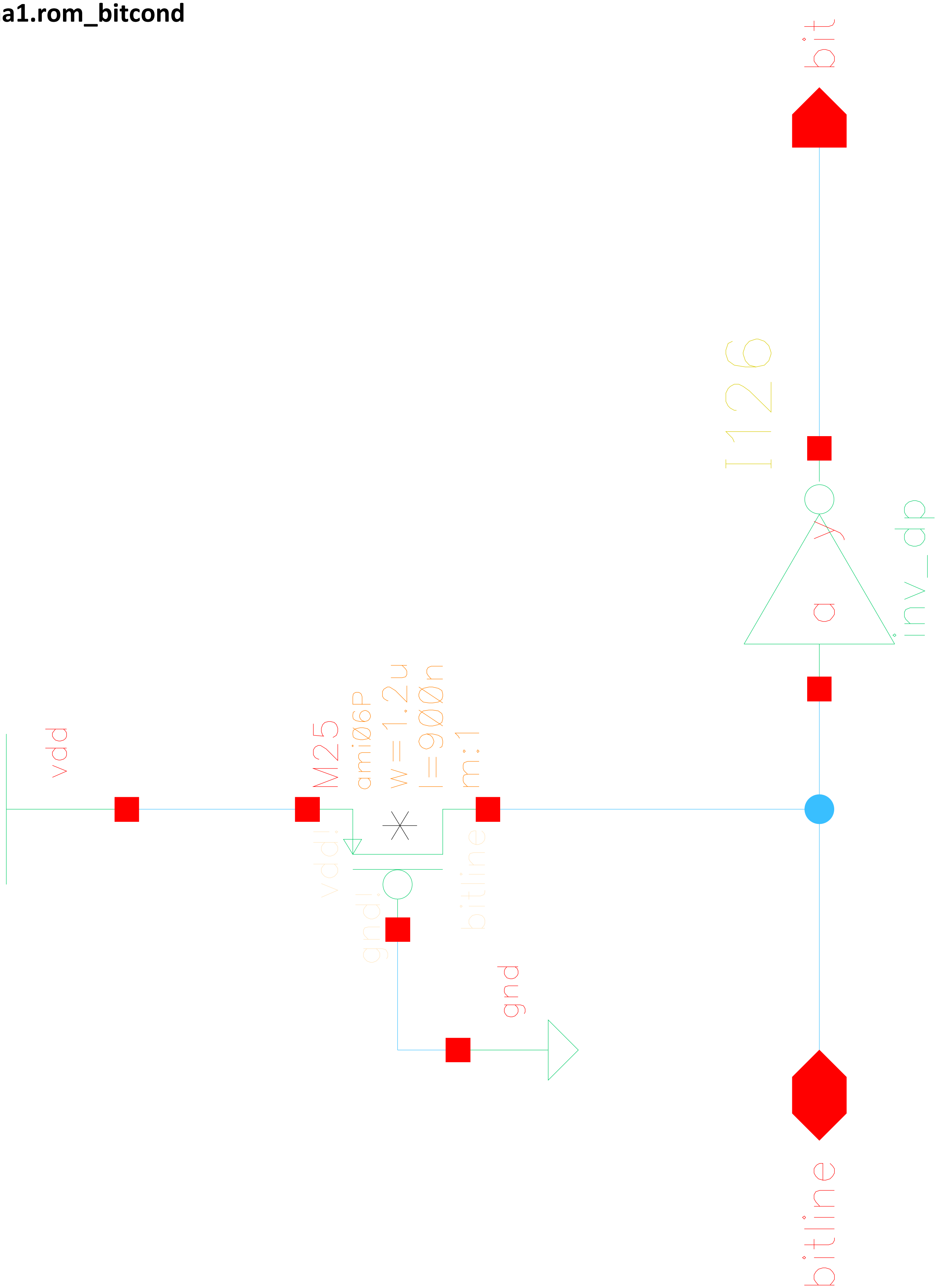

vdd
d
s
gnd

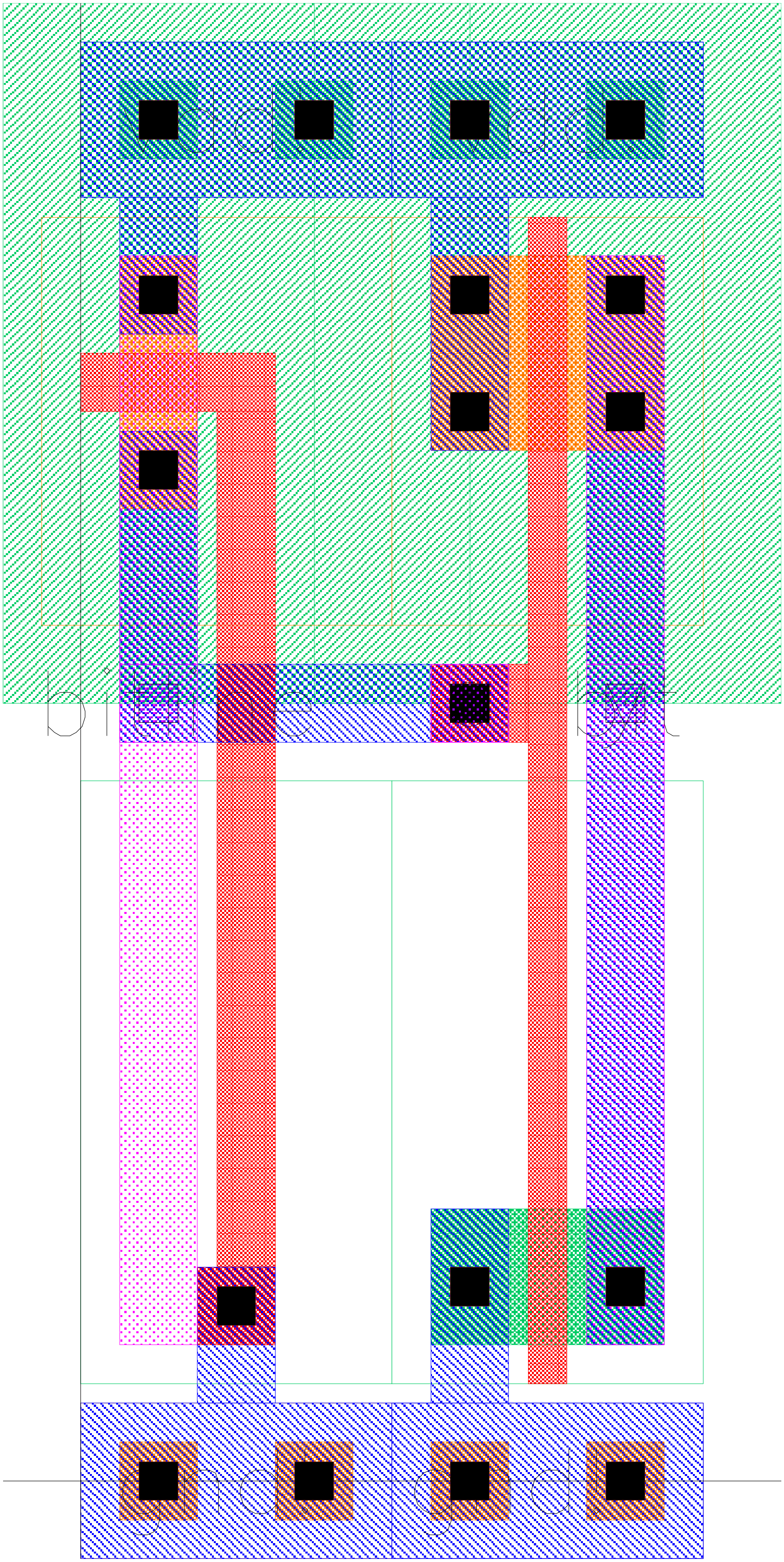
sha1.nand4_1x



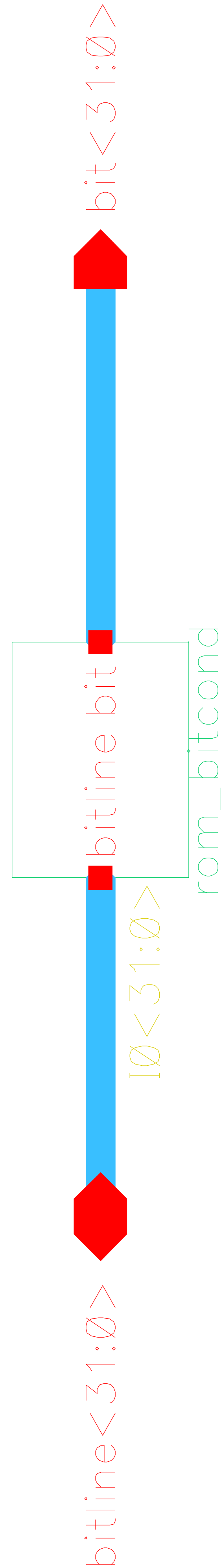

b
c
d

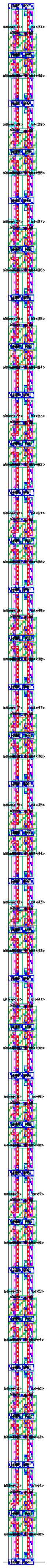
sha1.rom_bitcond



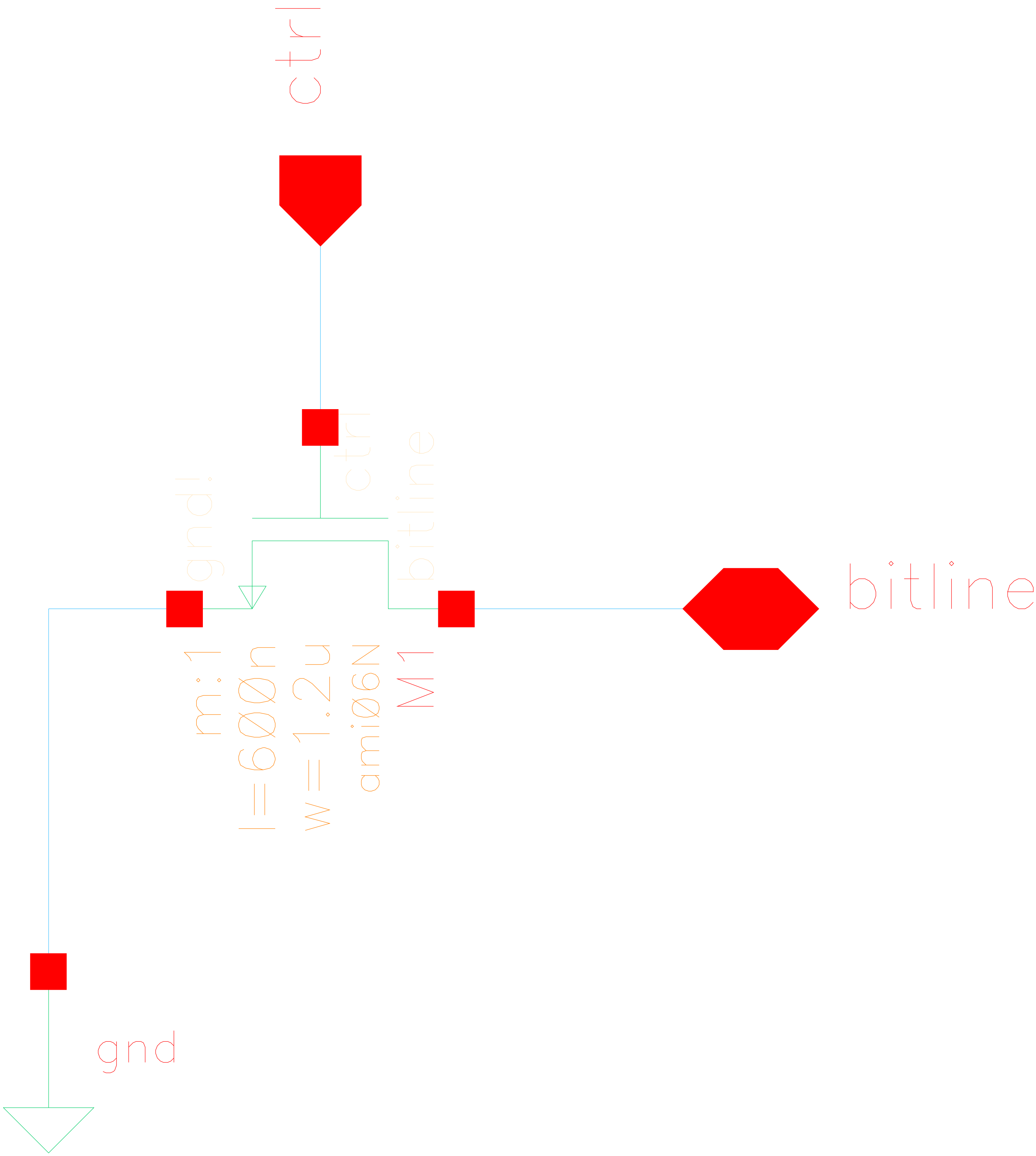


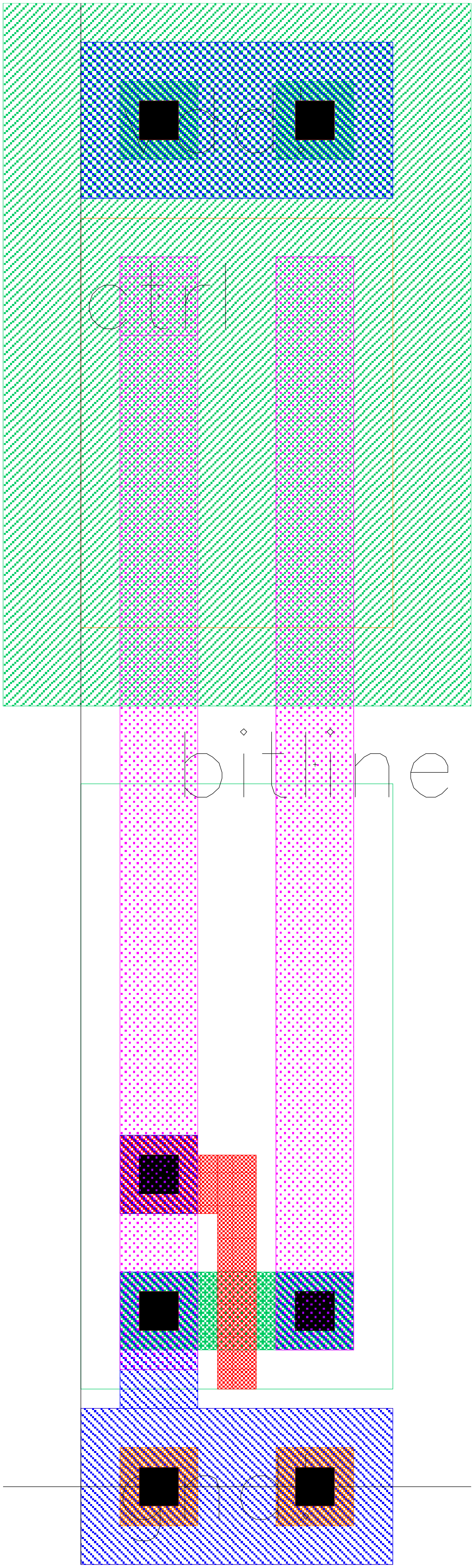
sha1.rom_bitcond32



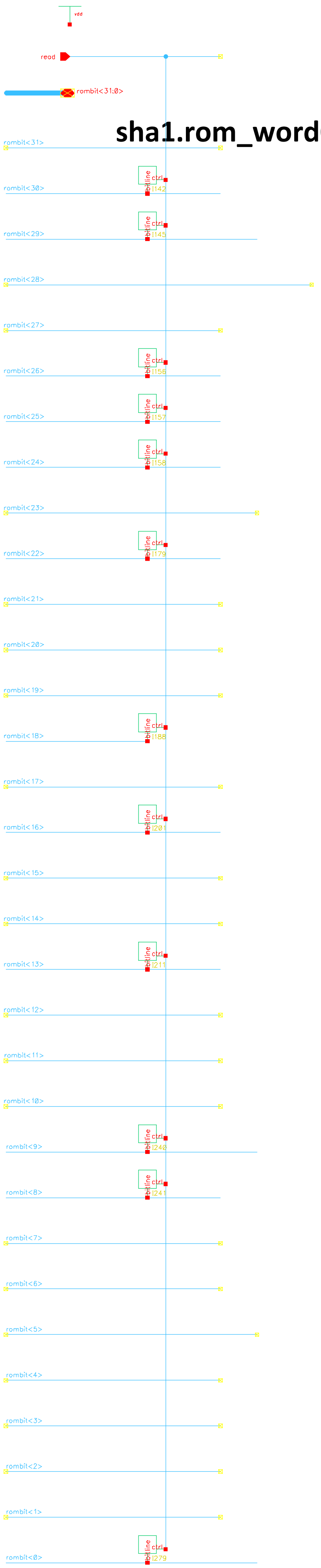


sha1.rom_one

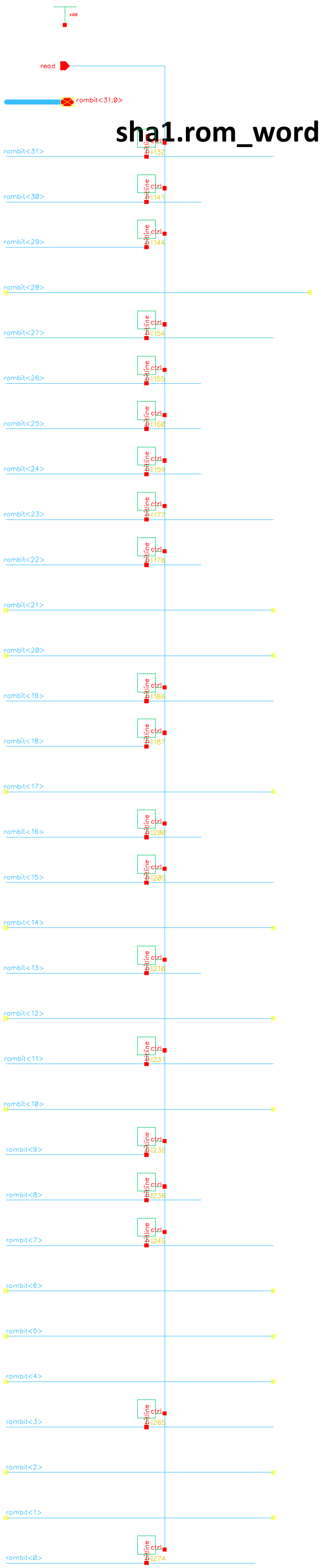



ctrl
bitline

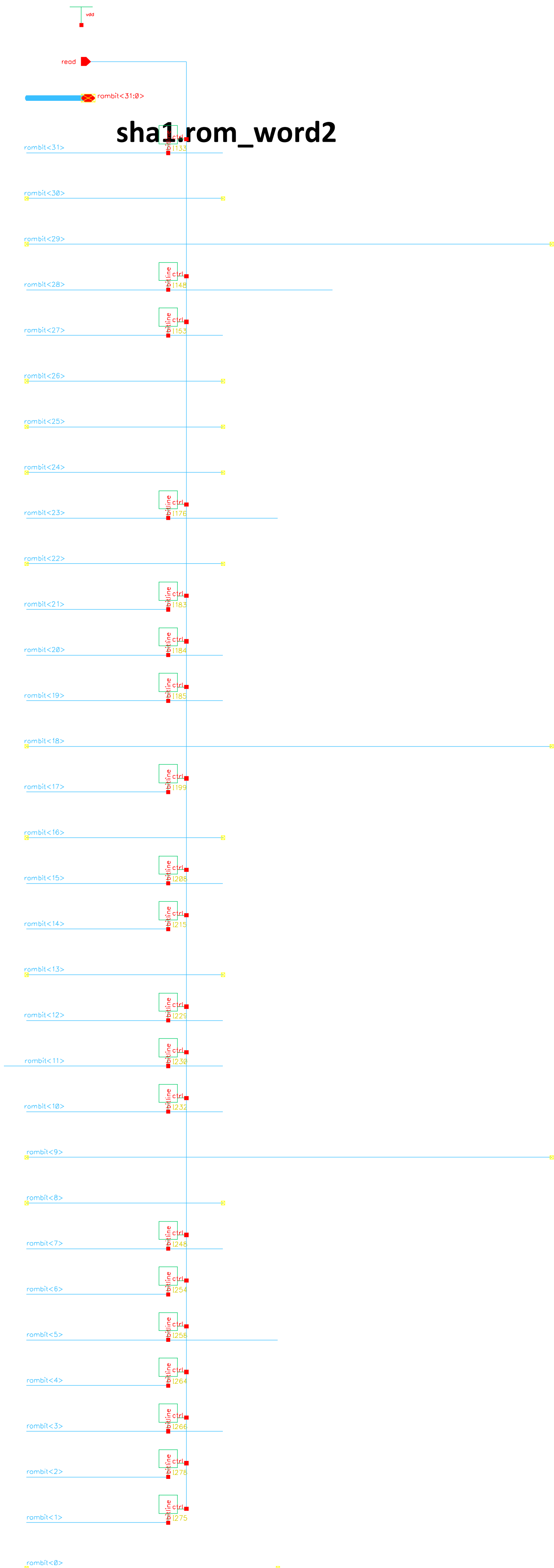
sha1.rom_word0



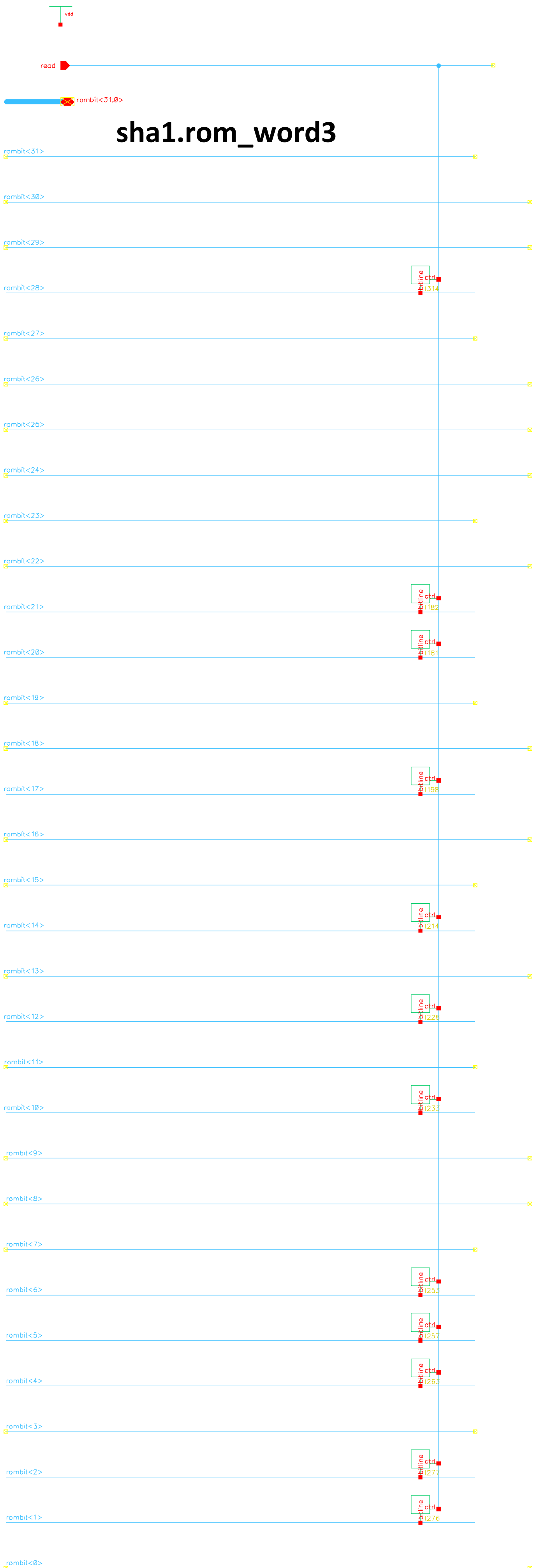
sha1.rom_word1



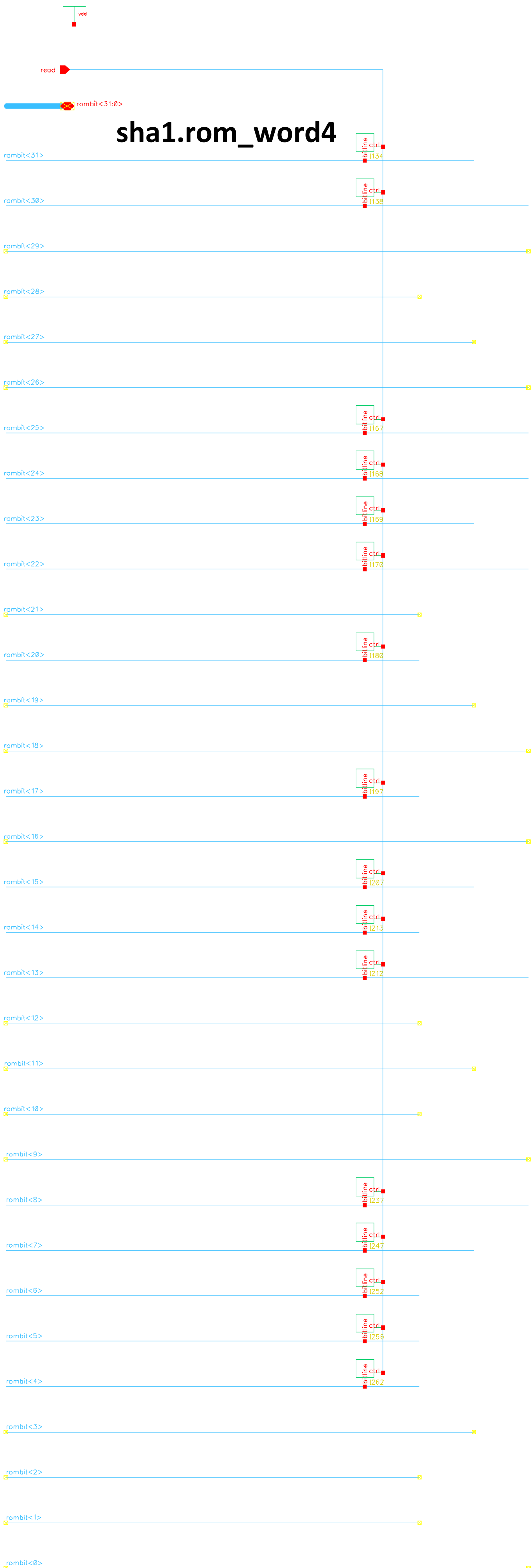
sha1_rom_word2



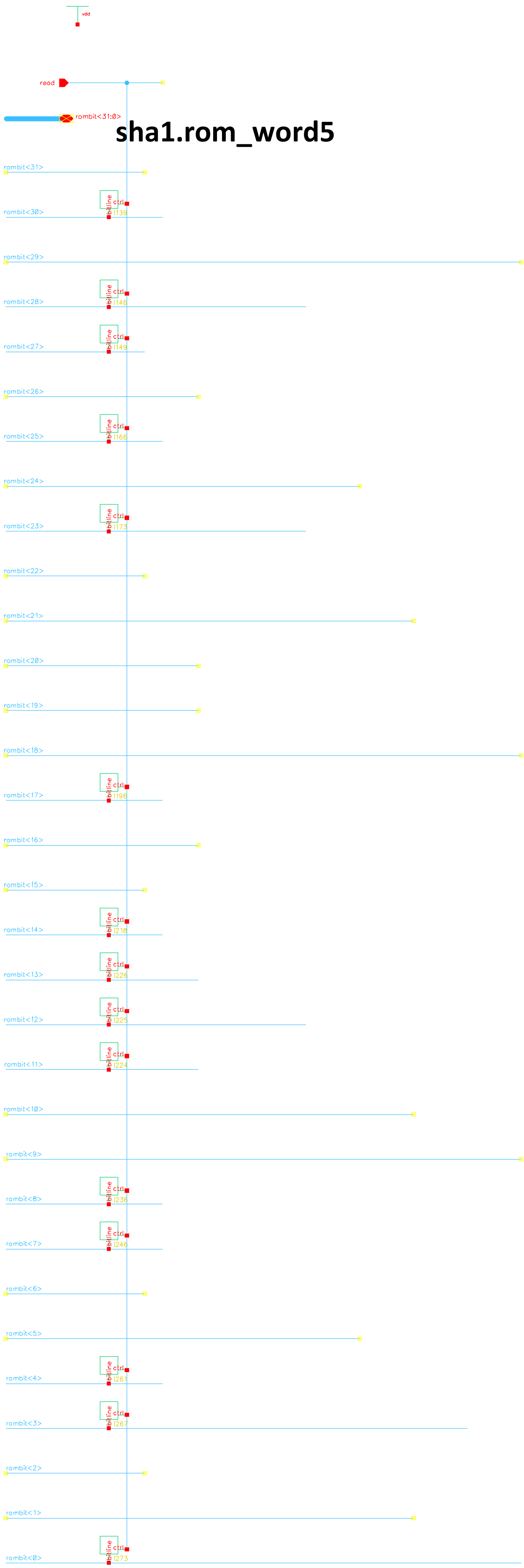
sha1.rom_word3



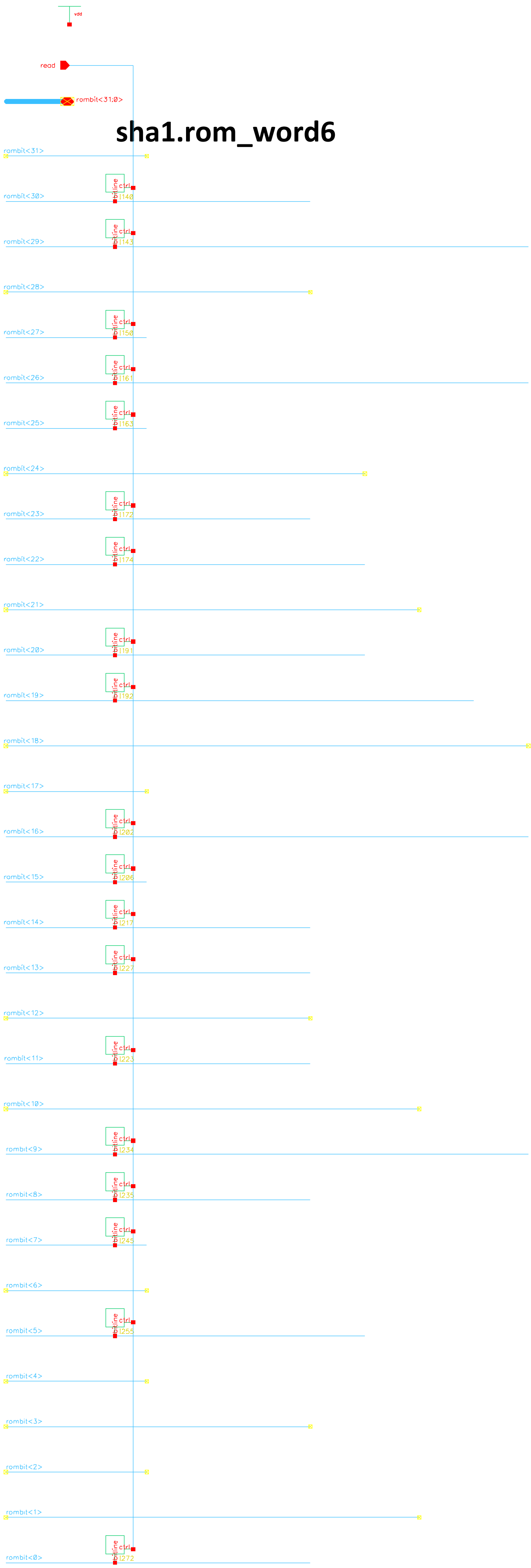
sha1.rom_word4



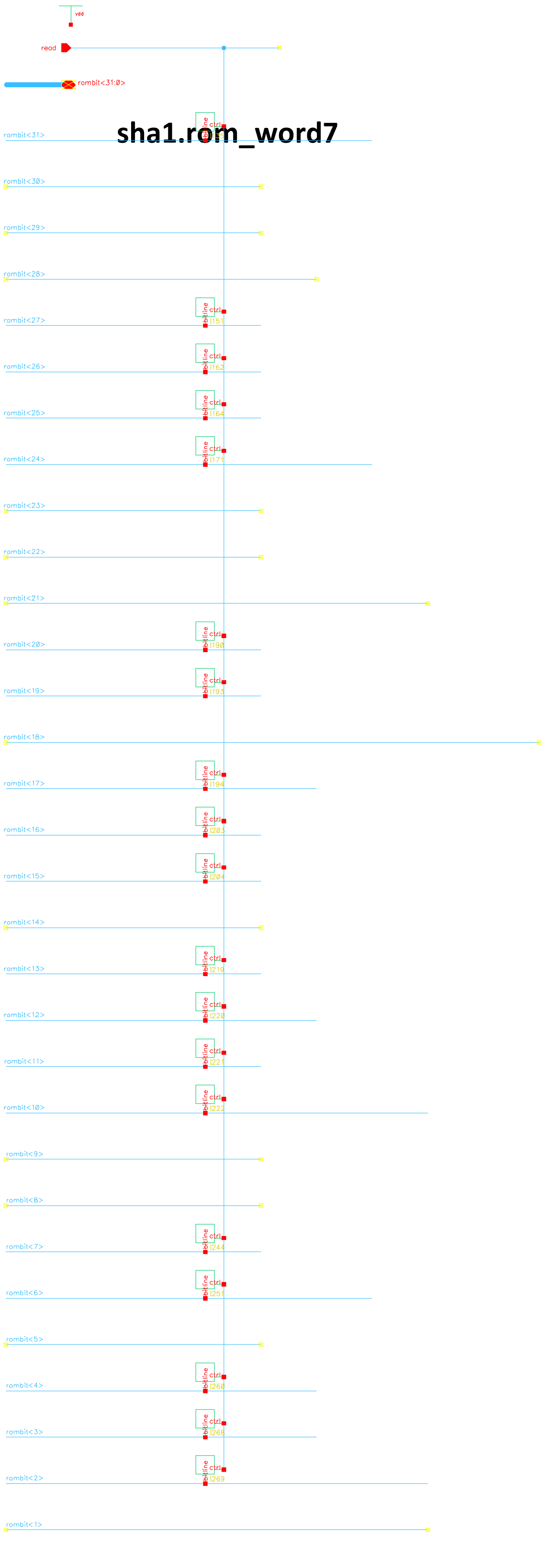
sha1.rom_word5

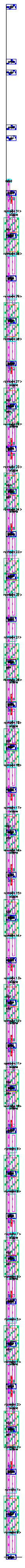


sha1.rom_word6

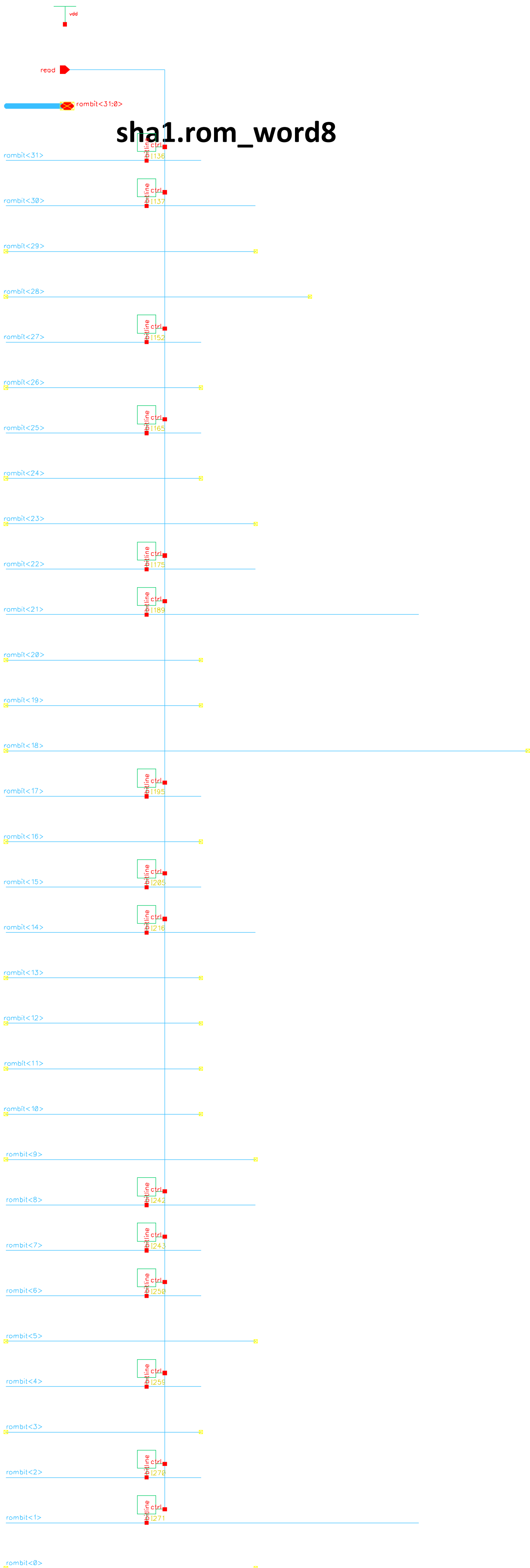


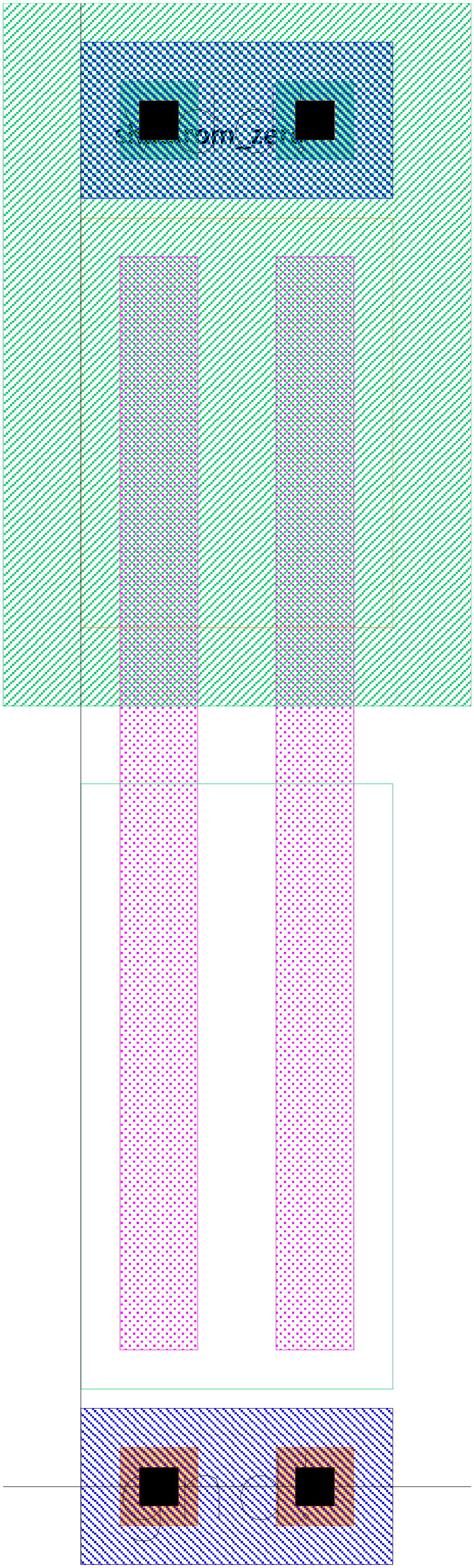
sha1.rom_word7





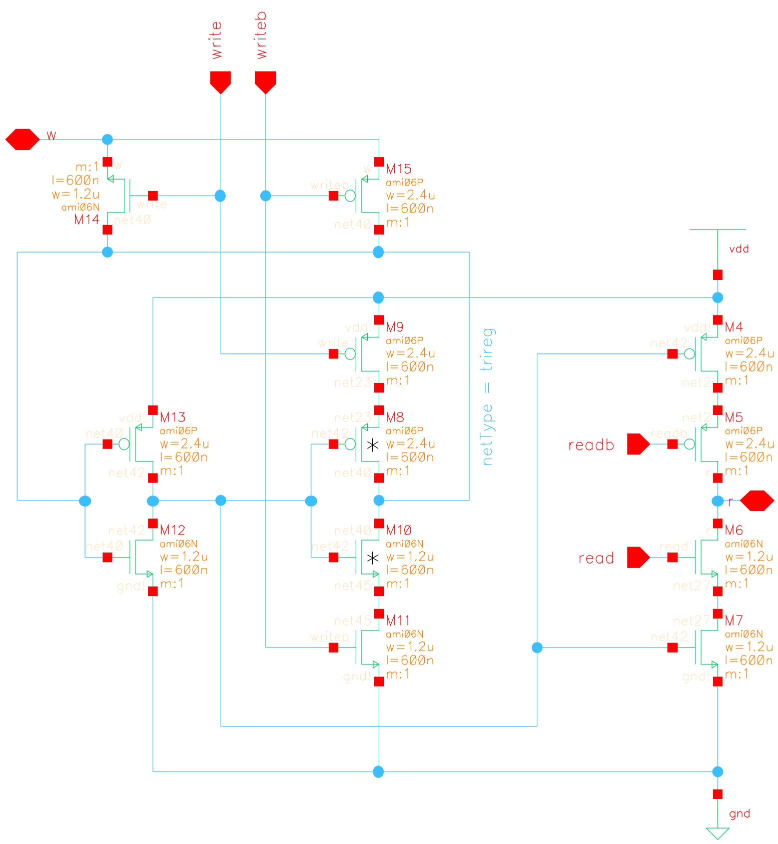
sha1.rom_word8





sha1.sram_cell

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED
Apr 5 23:47:52 2010

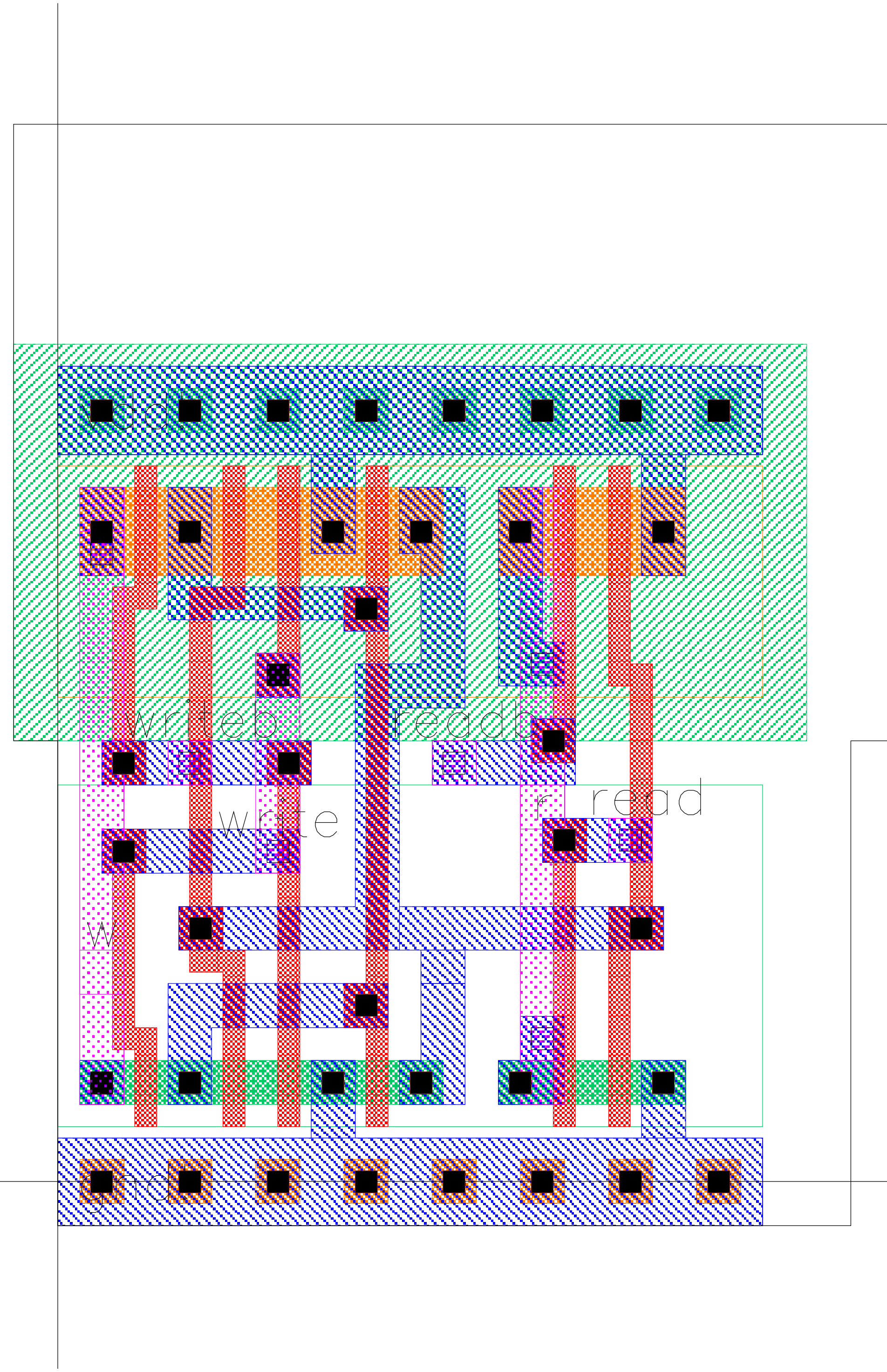
BY
wbuik

regram_dp

SIZE
A

REV

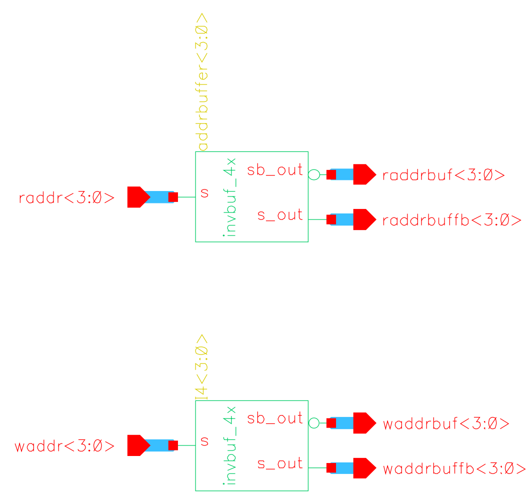
SHEET 1 of 1



writeb
readb
write
read
w
r

sha1.sram11_addrbuf

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED
Mar 30 13:52:24 2010

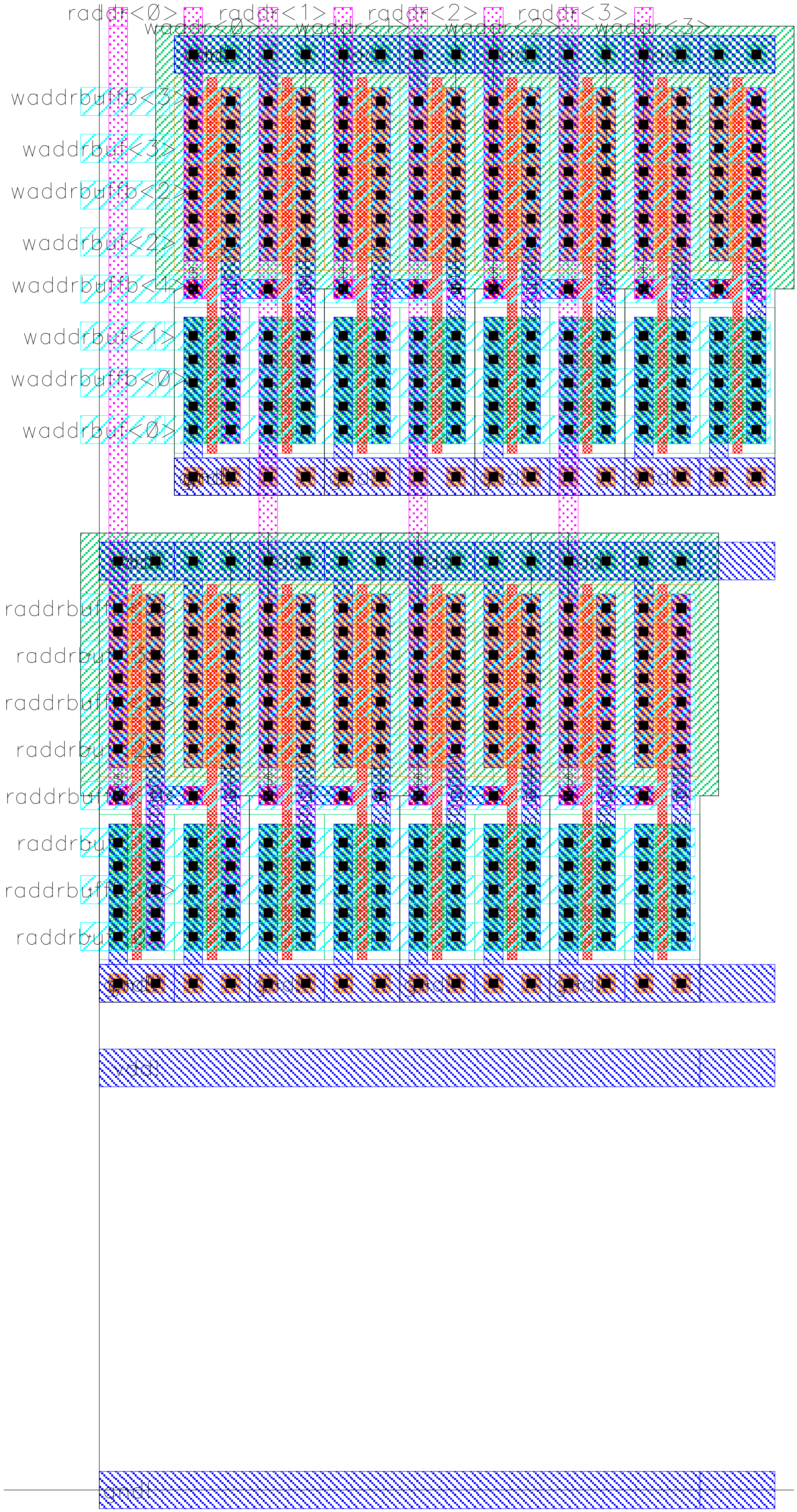
BY
wbuik

regramadrbuf

SIZE A

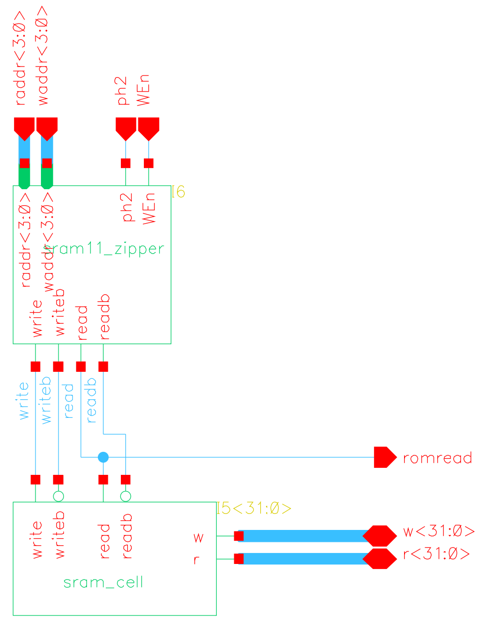
REV

SHEET 1 of 1


raddr<0>
raddr<1>
raddr<2>
raddr<3>
waddr<0>
waddr<1>
waddr<2>
waddr<3>
waddrbuffb<3>
waddrbuf<3>
waddrbuffb<2>
waddrbuf<2>
waddrbuffb<1>
waddrbuf<1>
waddrbuffb<0>
waddrbuf<0>
vdd!
gnd!

sha1.sram11_word

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

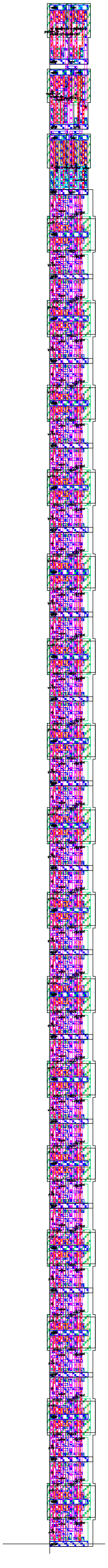
UPDATED
Apr 13 15:06:37 2010

BY
wbuik

regramvector_dp

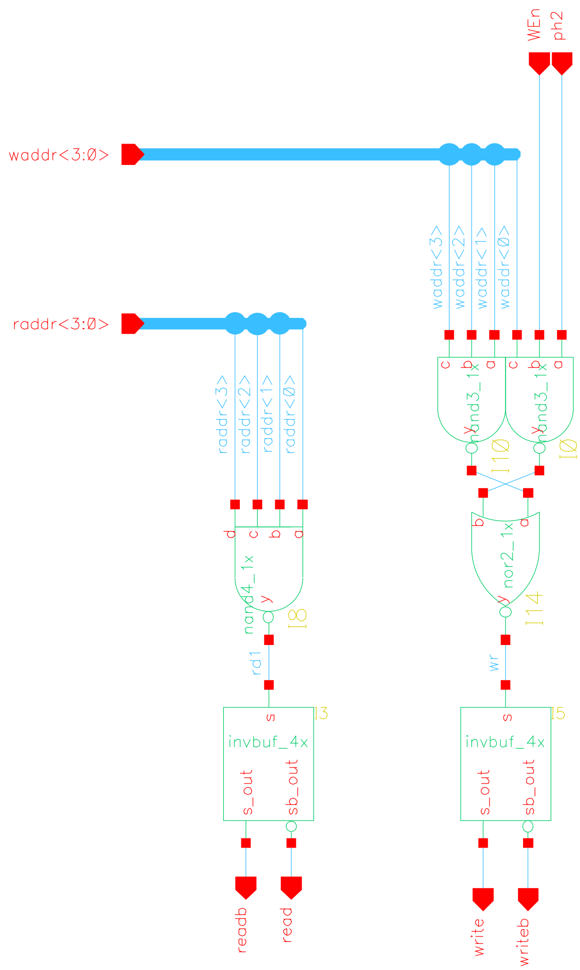
SIZE A REV

SHEET 1 of 1



sha1.sram11_zipper

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED
Apr 6 19:04:17 2010

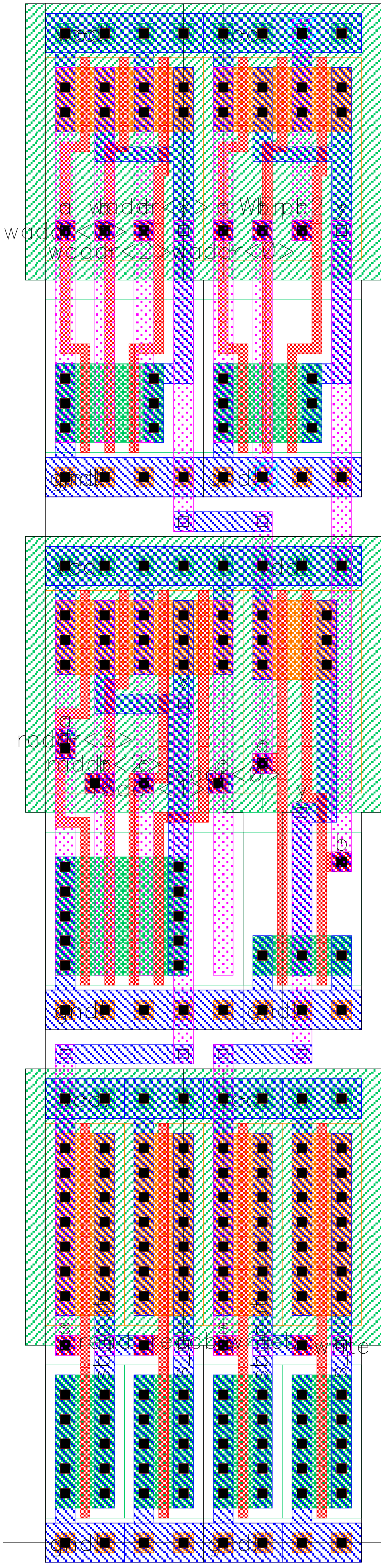
BY
wbuik

regram_zipper

SIZE A

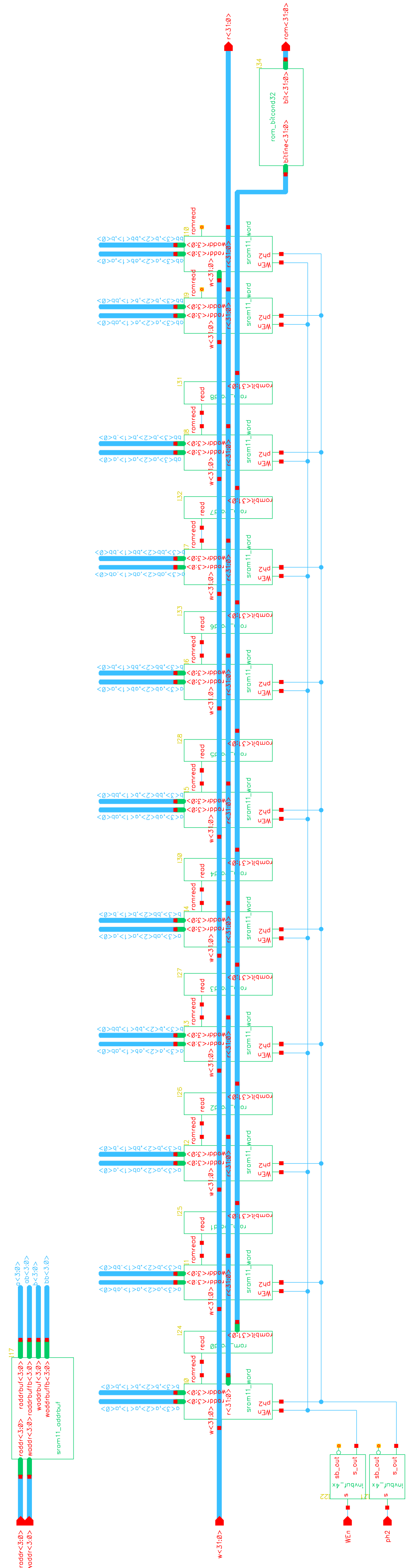
REV

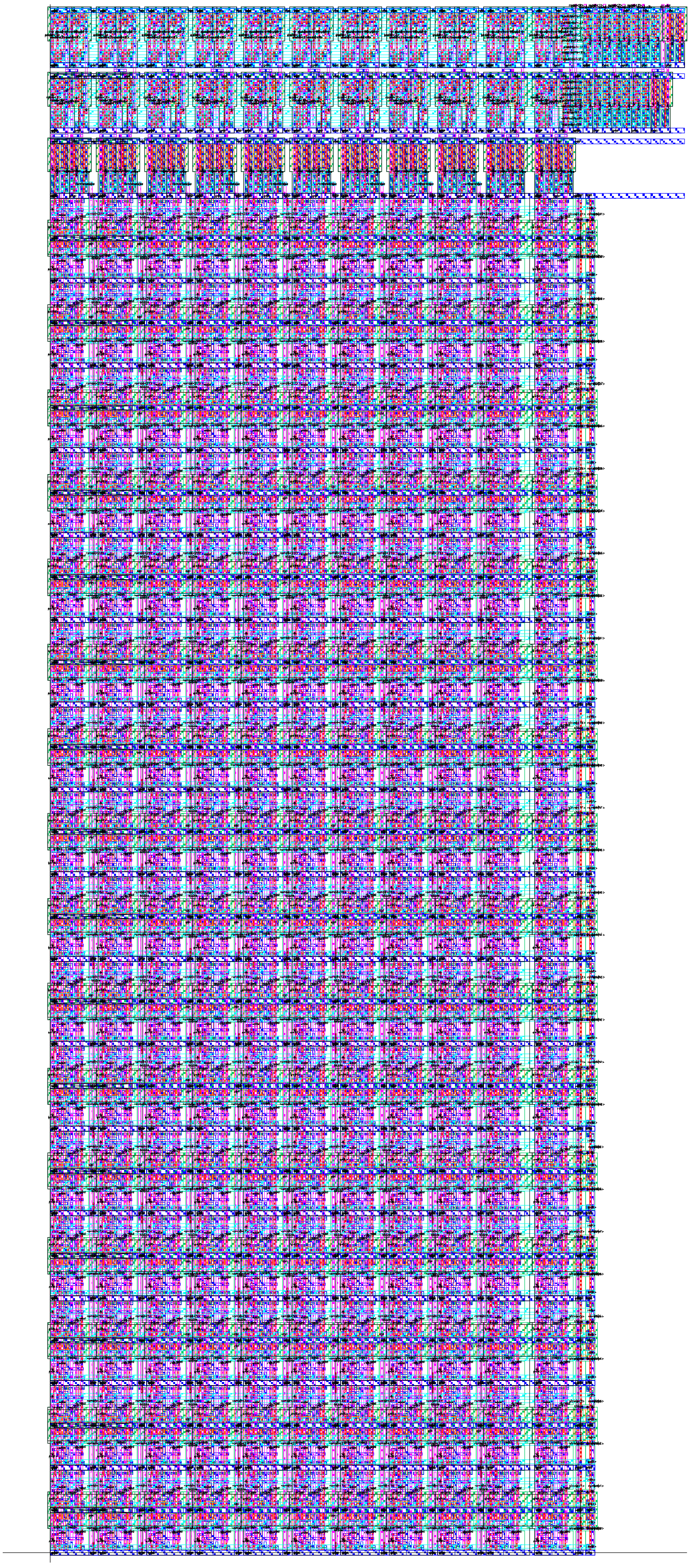
SHEET 1 of 1



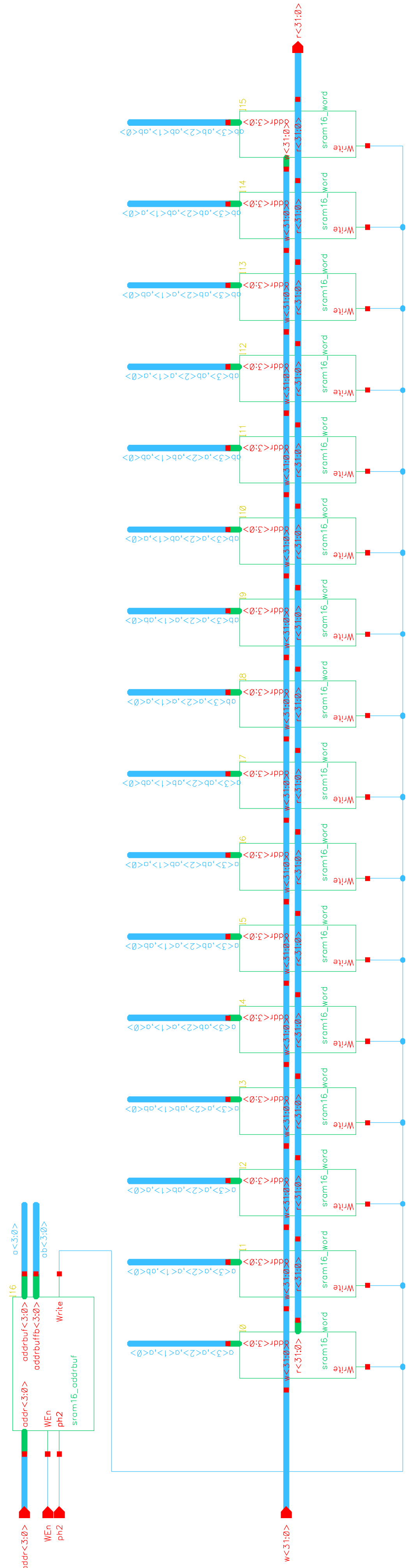
waddr<2>
waddr<0>
raddr<3>
raddr<2>
gnd

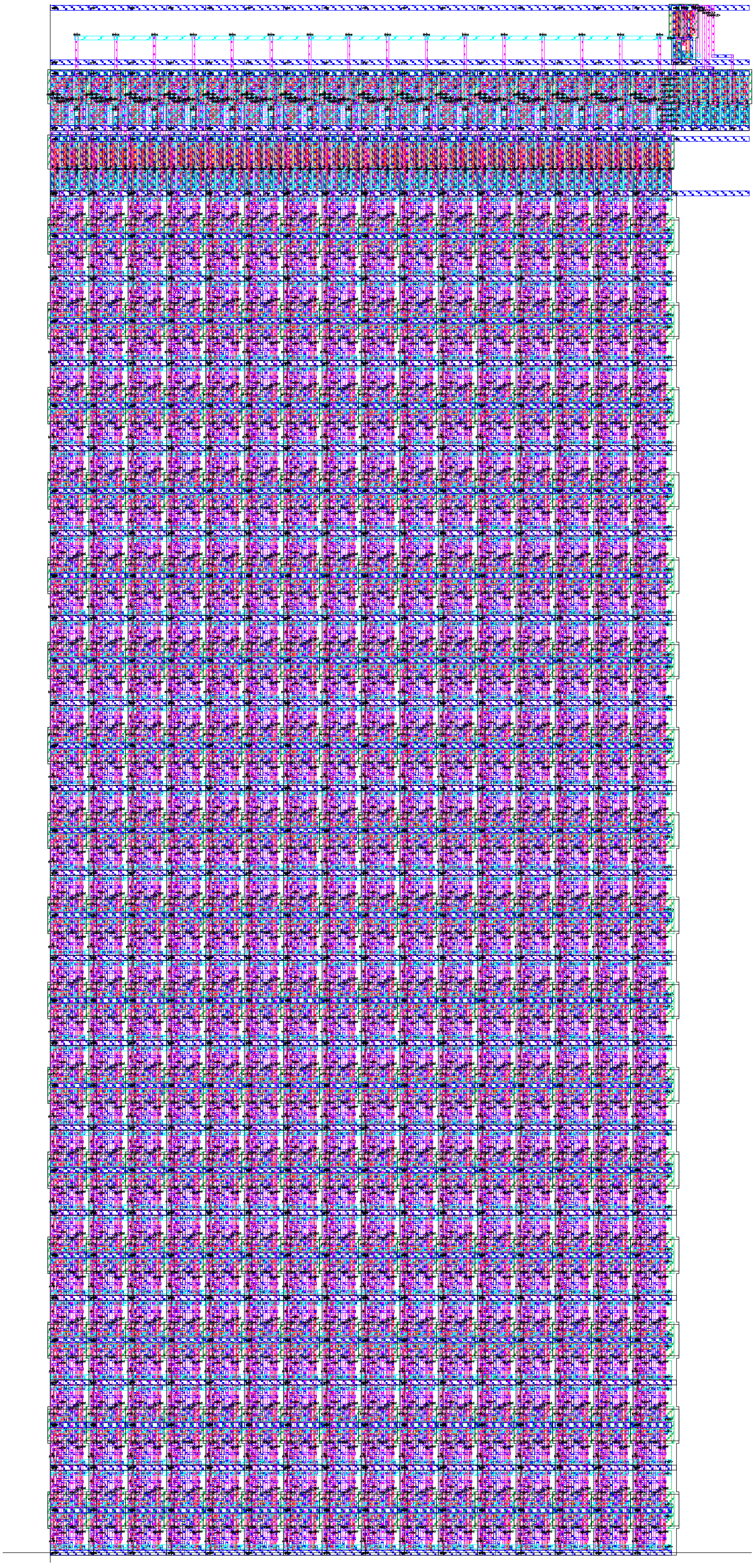
sha1.sram11rom





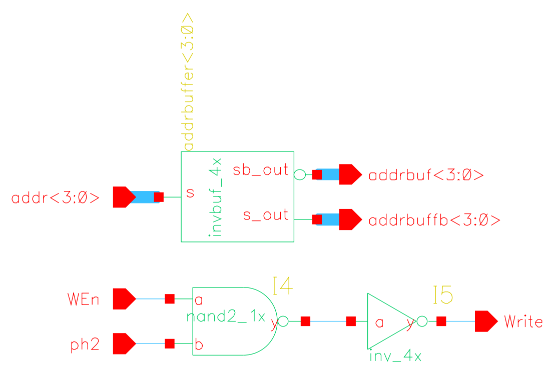
sha1.sram16


r<31:0>
w<31:0>
addr<3:0>
WEn
ph2
a<3:0>
ab<3:0>
addrbuf<3:0>
addrbuffb<3:0>
Write
sram16_addrbuf
sram16_word
addr<3:0>
w<31:0>
r<31:0>



REVISIONS			
ZONE	REV	DESCRIPTION	DATE

sha1.sram16_addrbuf



Harvey Mudd College

UPDATED
Apr 6 16:17:21 2010

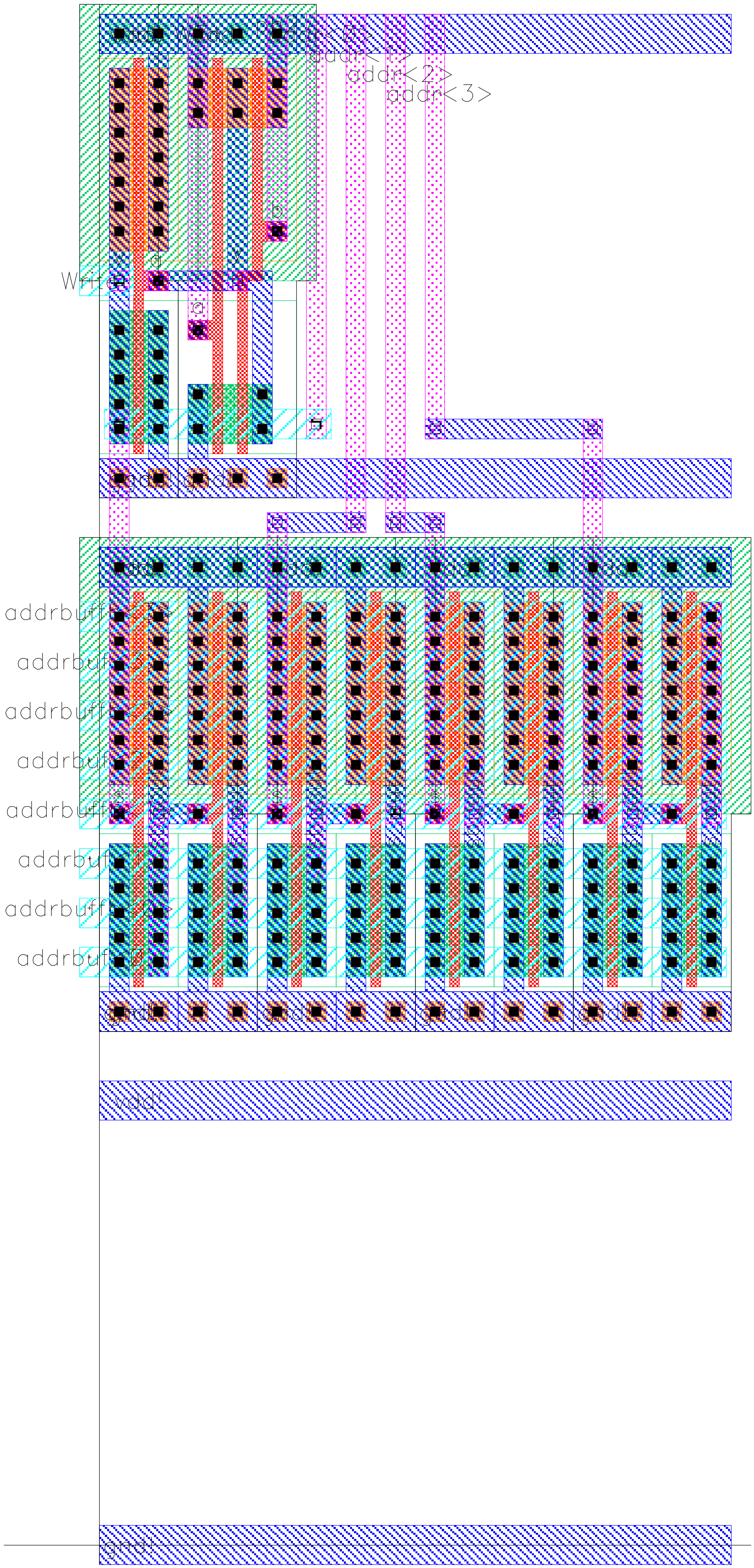
BY
wbuik

regramadrbuf

SIZE A

REV

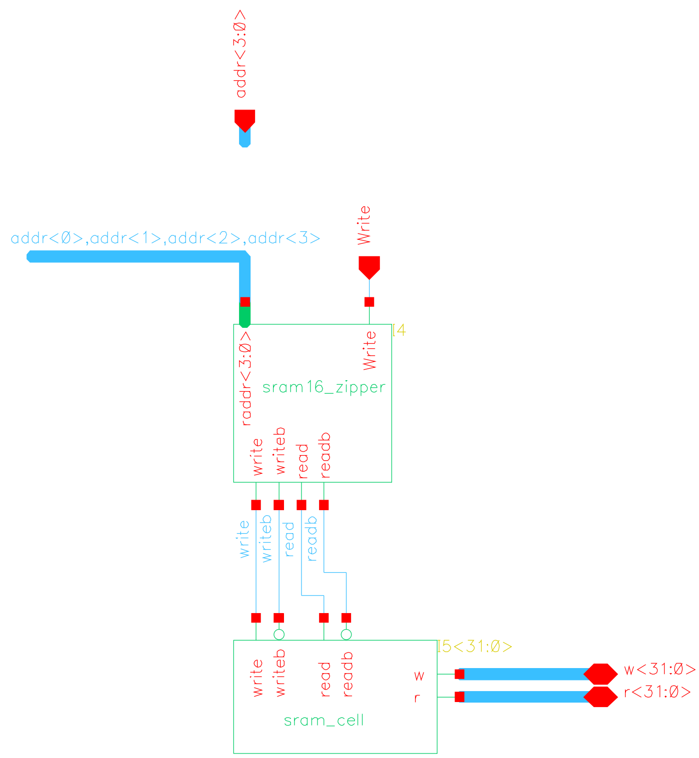
SHEET 1 of 1


addr<1>
addr<2>
addr<3>
Write
vdd!
gnd!

sha1.sram16_word

ZONE	REV	DESCRIPTION	DATE

REVISIONS



Harvey Mudd College

UPDATED
Apr 12 16:02:53 2010

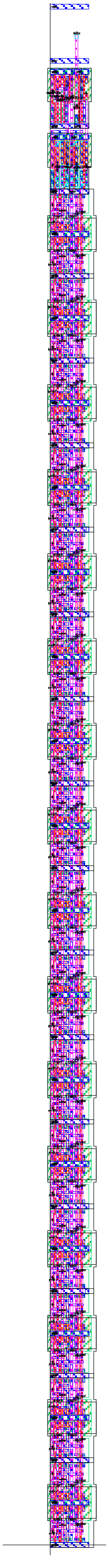
BY
wbuik

SIZE
A

REV

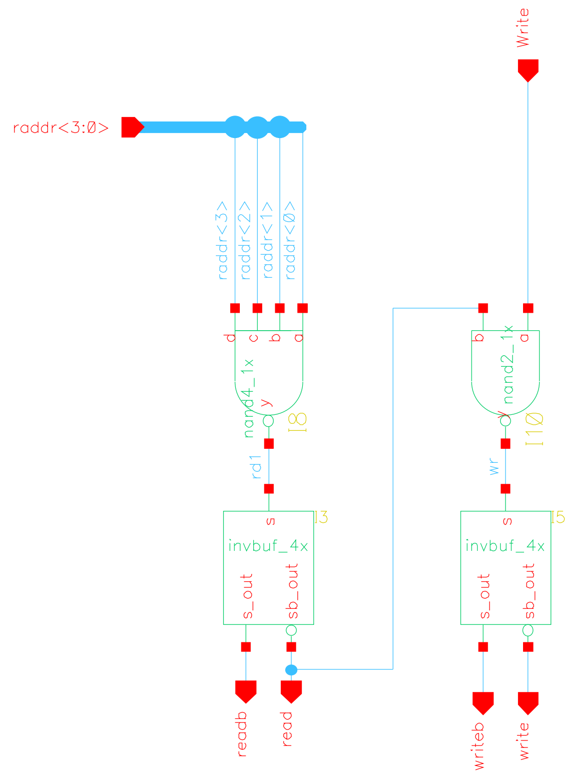
regramvector_dp

SHEET 1 of 1



sha1.sram16_zipper

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED
Apr 12 01:58:09 2010

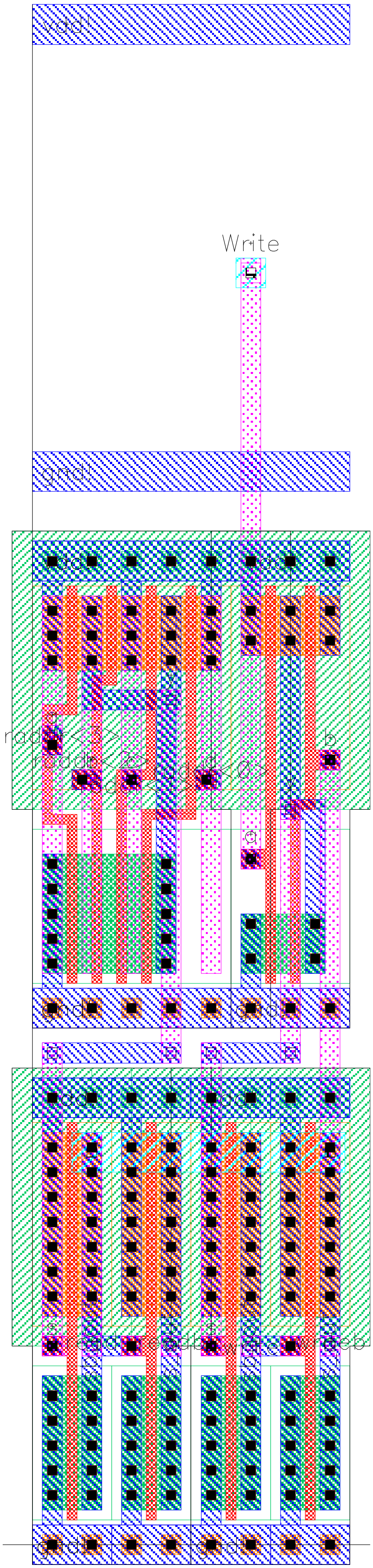
BY
wbuik

SIZE
A

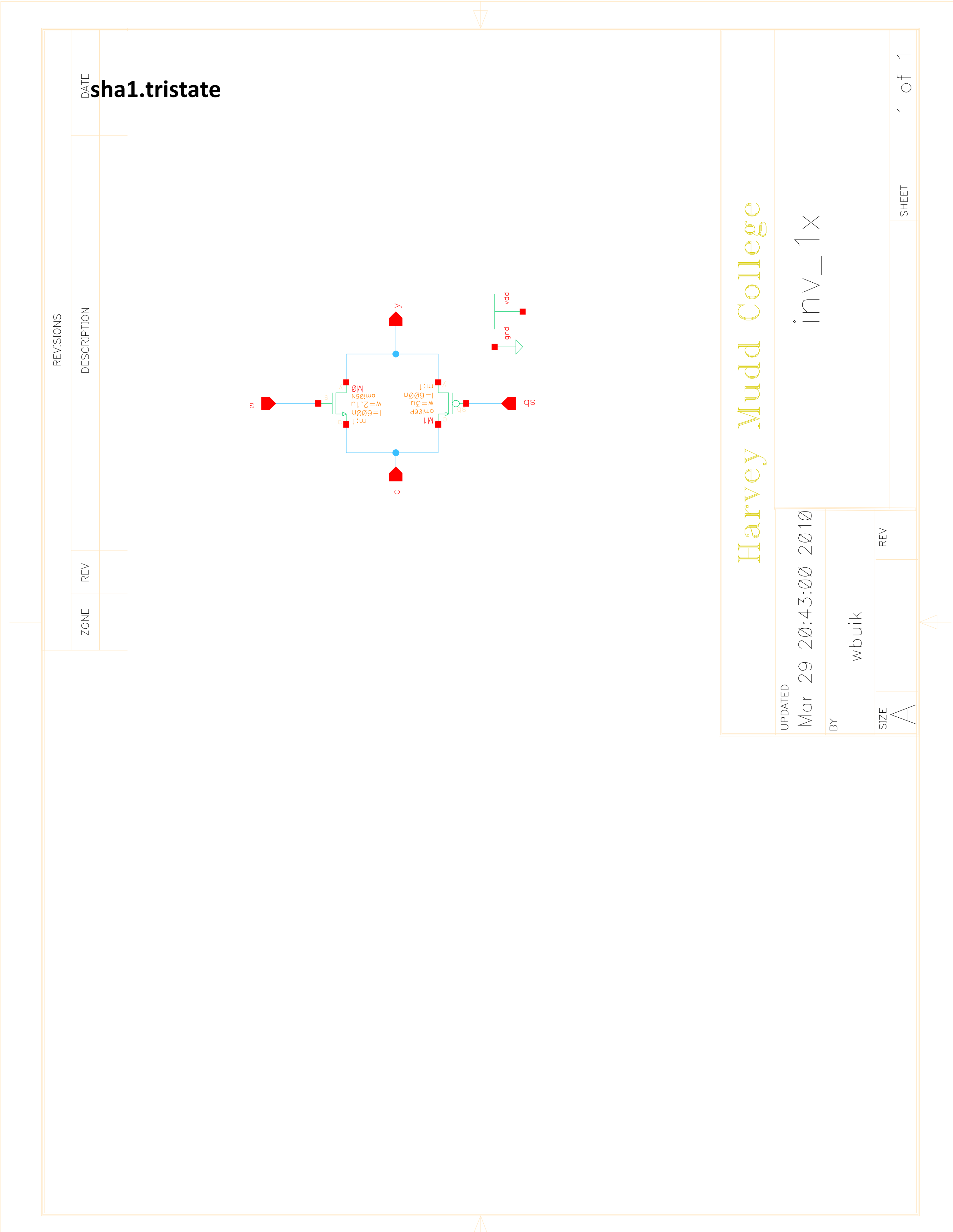
REV

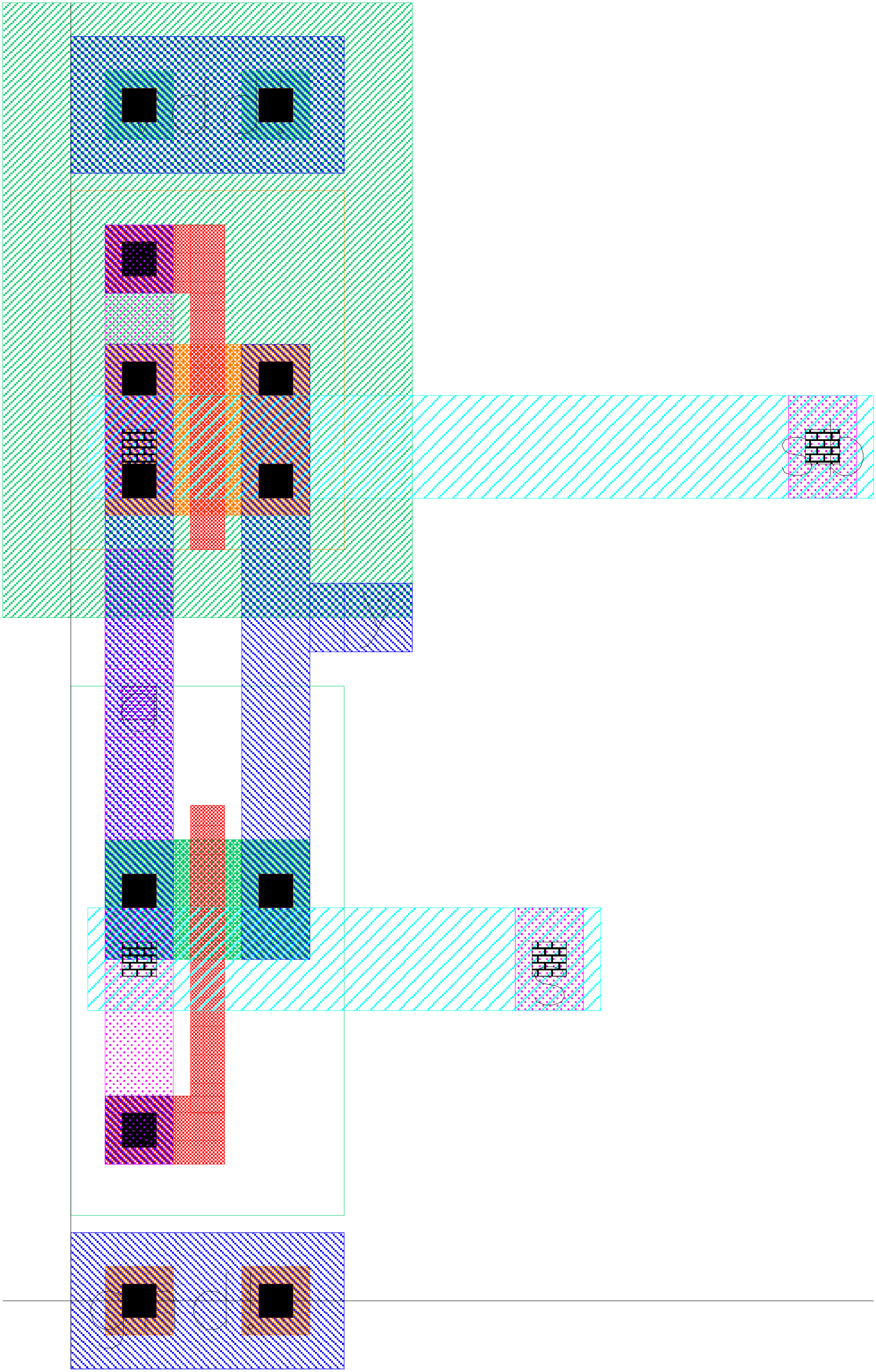
regram_zipper

SHEET 1 of 1

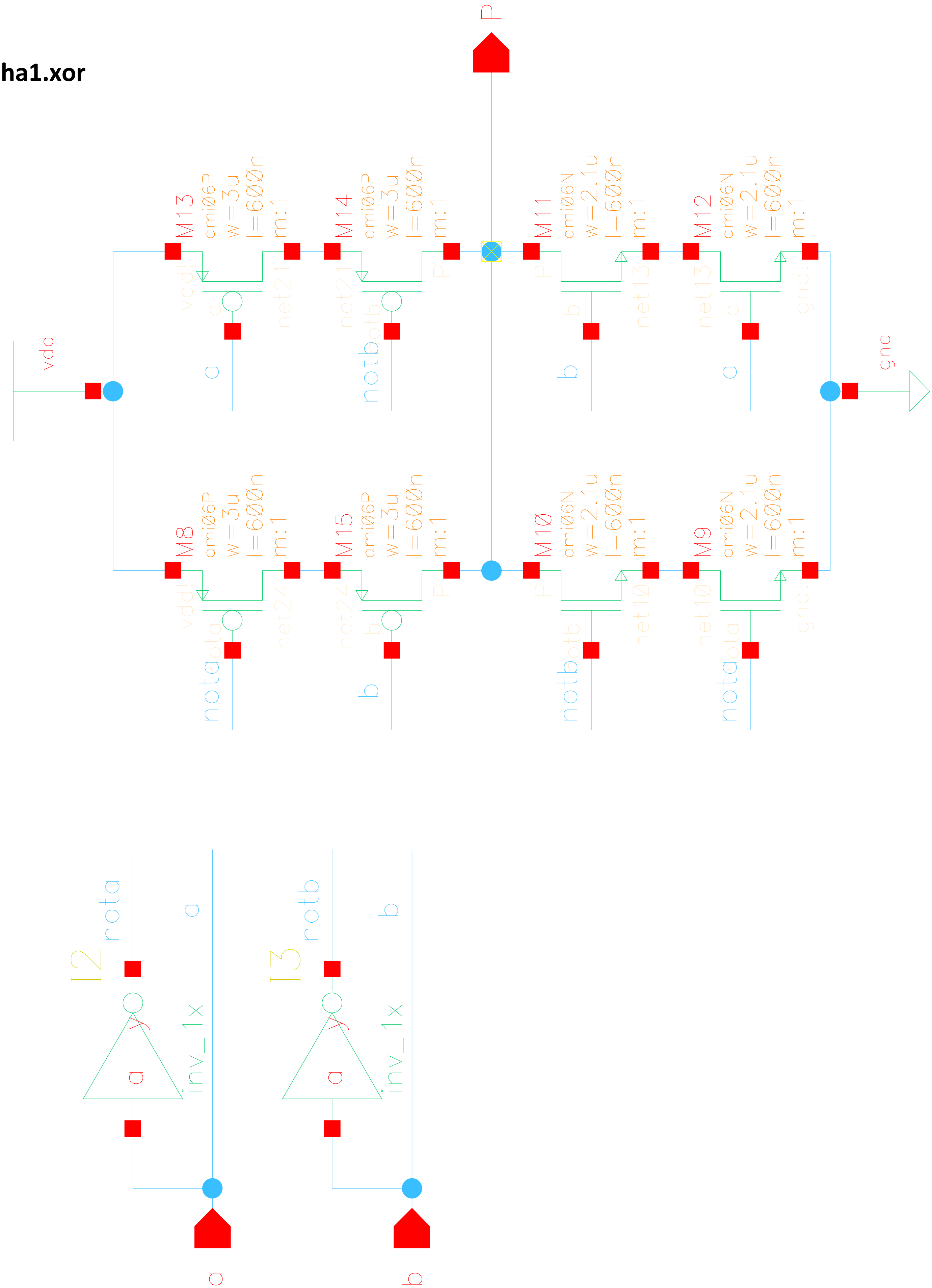

vdd
Write
gnd
vdd
b
gnd
vdd
writeb
gnd

sha1.tristate

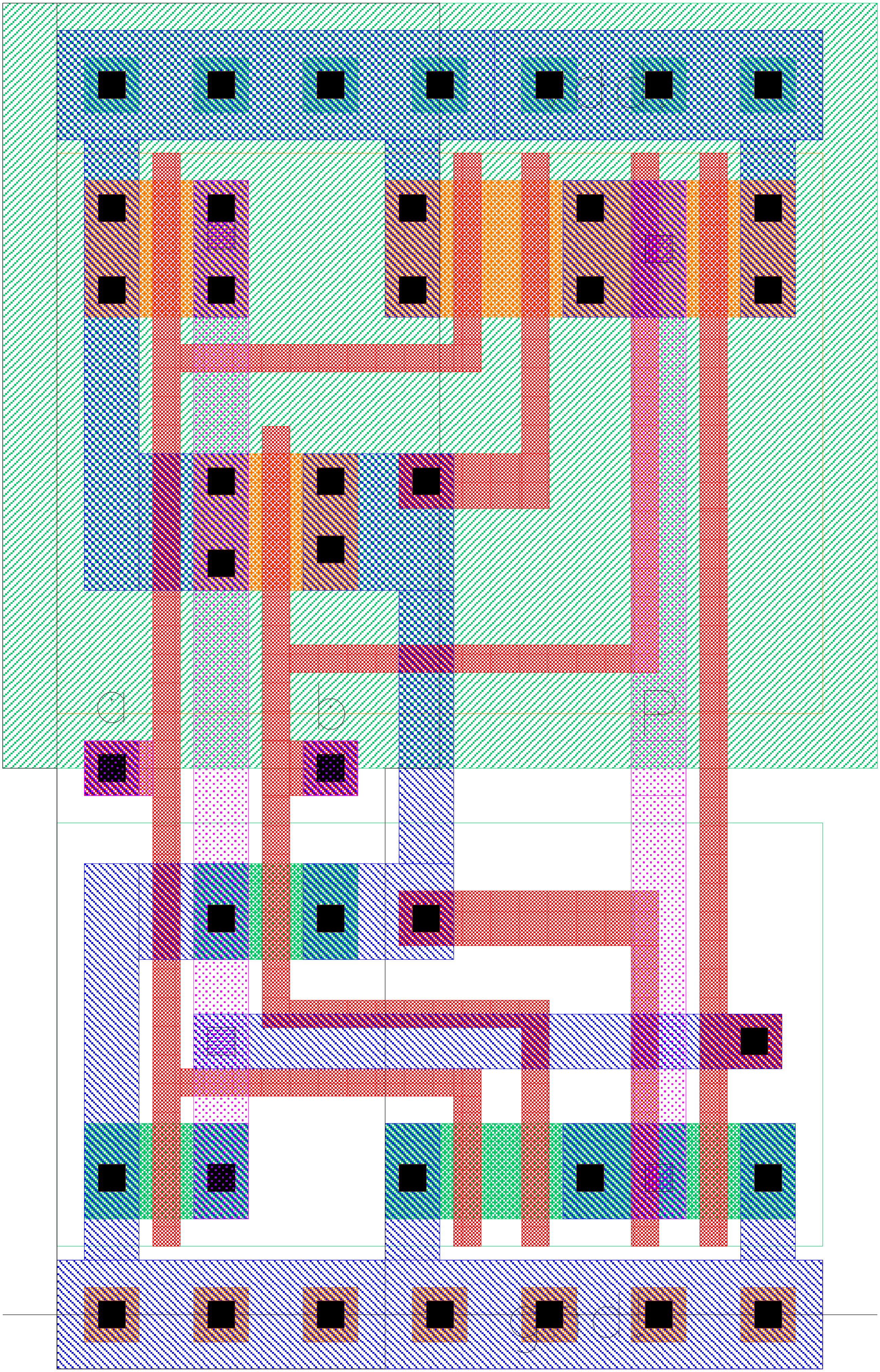




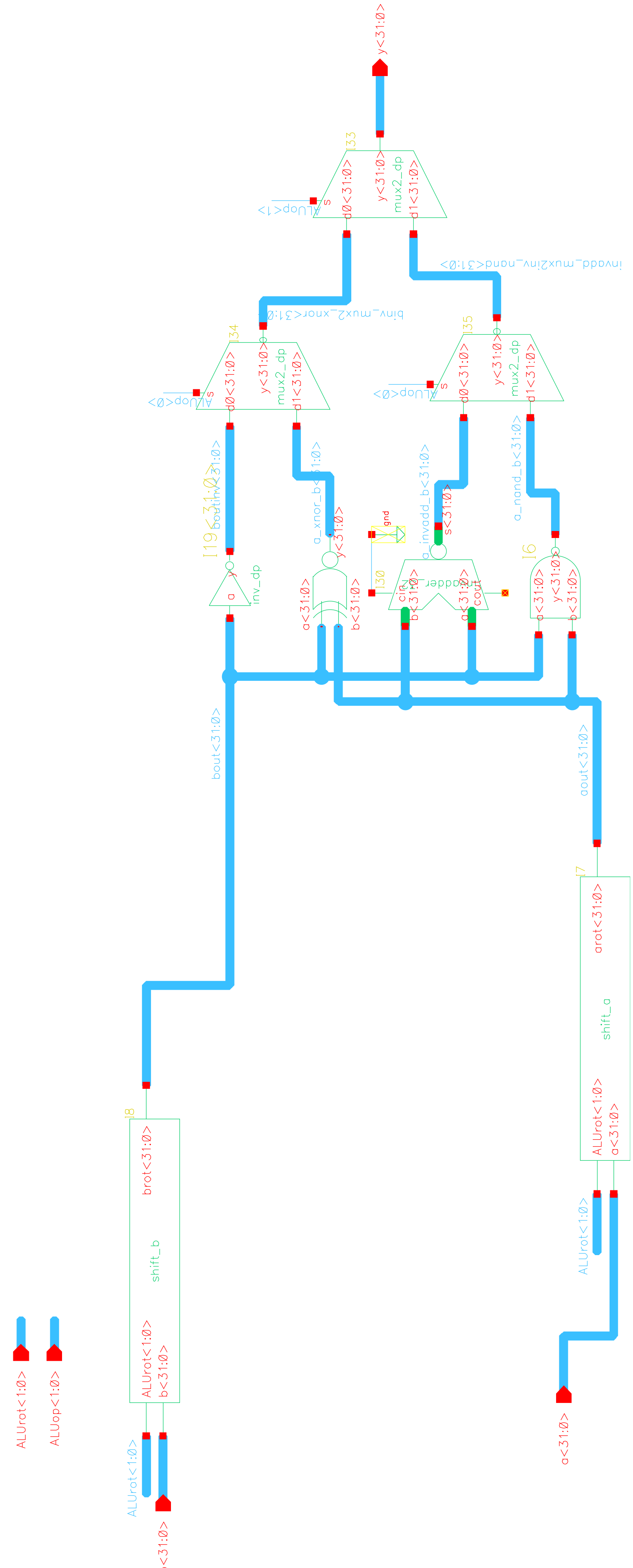
sha1.xor



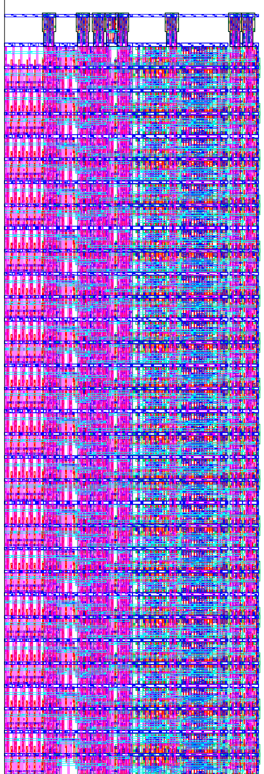
P
vdd
gnd
M13
ami06P
w=3u
l=600n
m:1
M14
ami06P
w=3u
l=600n
m:1
M11
ami06N
w=2.1u
l=600n
m:1
M12
ami06N
w=2.1u
l=600n
m:1
M8
ami06P
w=3u
l=600n
m:1
M15
ami06P
w=3u
l=600n
m:1
M10
ami06N
w=2.1u
l=600n
m:1
M9
ami06N
w=2.1u
l=600n
m:1
a
notb
b
a
nota
b
notb
nota
I2
nota
a
inv_1x
I3
notb
b
inv_1x
a
b


vdd
a
b
p
gnd

sha1_alu.alu

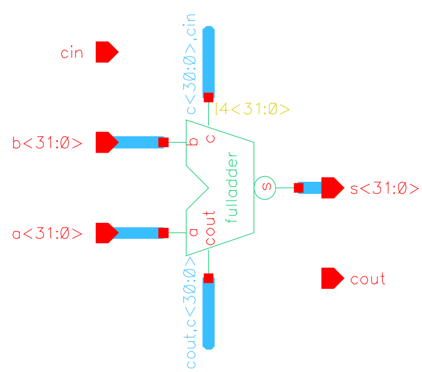


y<31:0>
mux2_dp
d0<31:0>
d1<31:0>
ALUop<1>
ALUop<0>
invadd_mux2inv_nand<31:0>
binv_mux2_xnor<31:0>
boutinv<31:0>
a_xnor_b<31:0>
a_invadd_b<31:0>
a_nand_b<31:0>
inv_dp
adder
cin
cout
s<31:0>
a<31:0>
b<31:0>
gnd
bout<31:0>
aout<31:0>
shift_a
arot<31:0>
ALUrot<1:0>
shift_b
brot<31:0>
ALUop<1:0>



sha1_alu.invadder_32

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED
Apr 14 02:47:32 2010

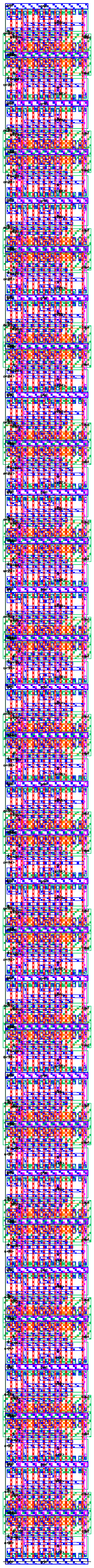
BY
wbuik

adder_8

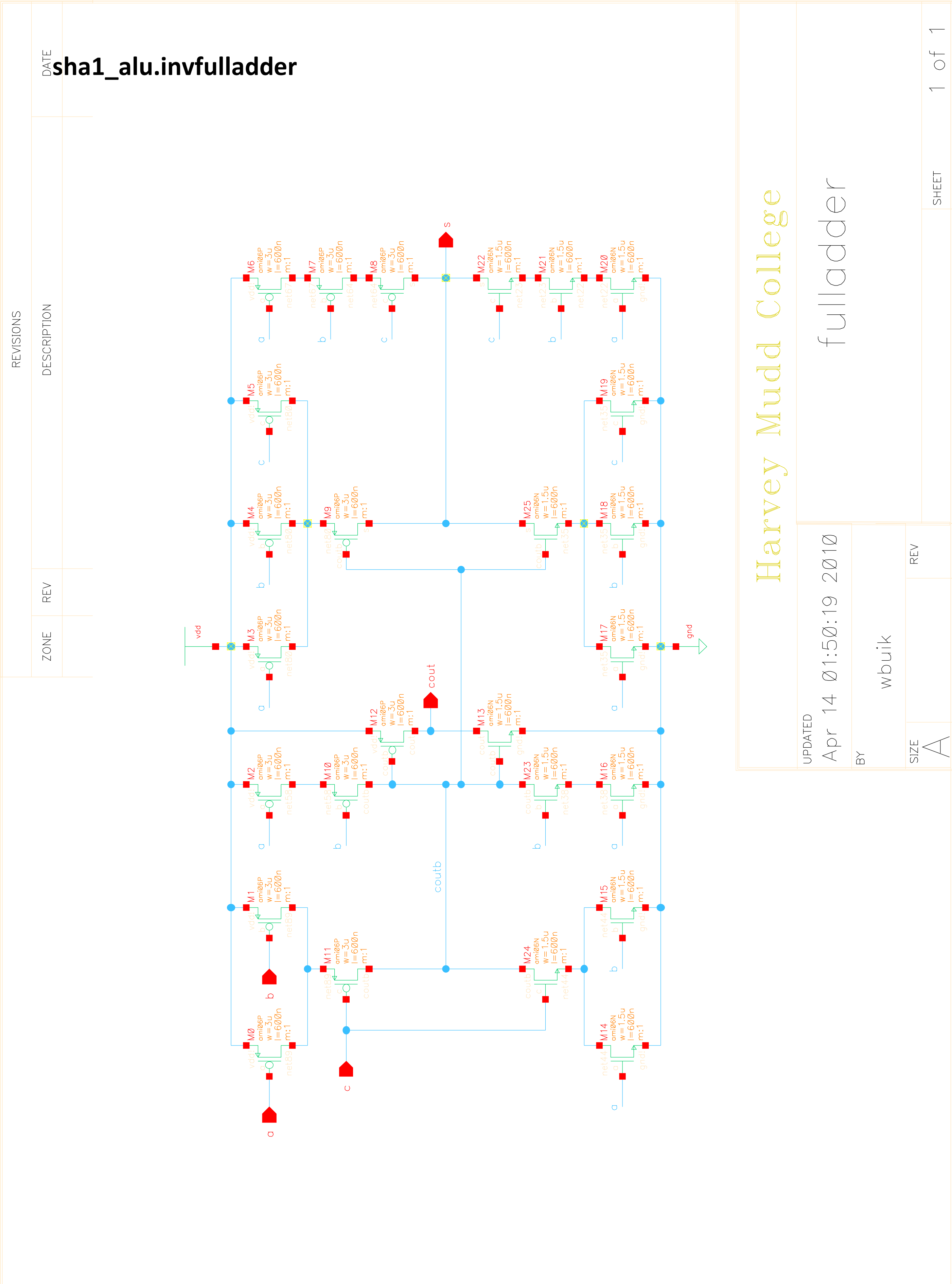
SIZE
A

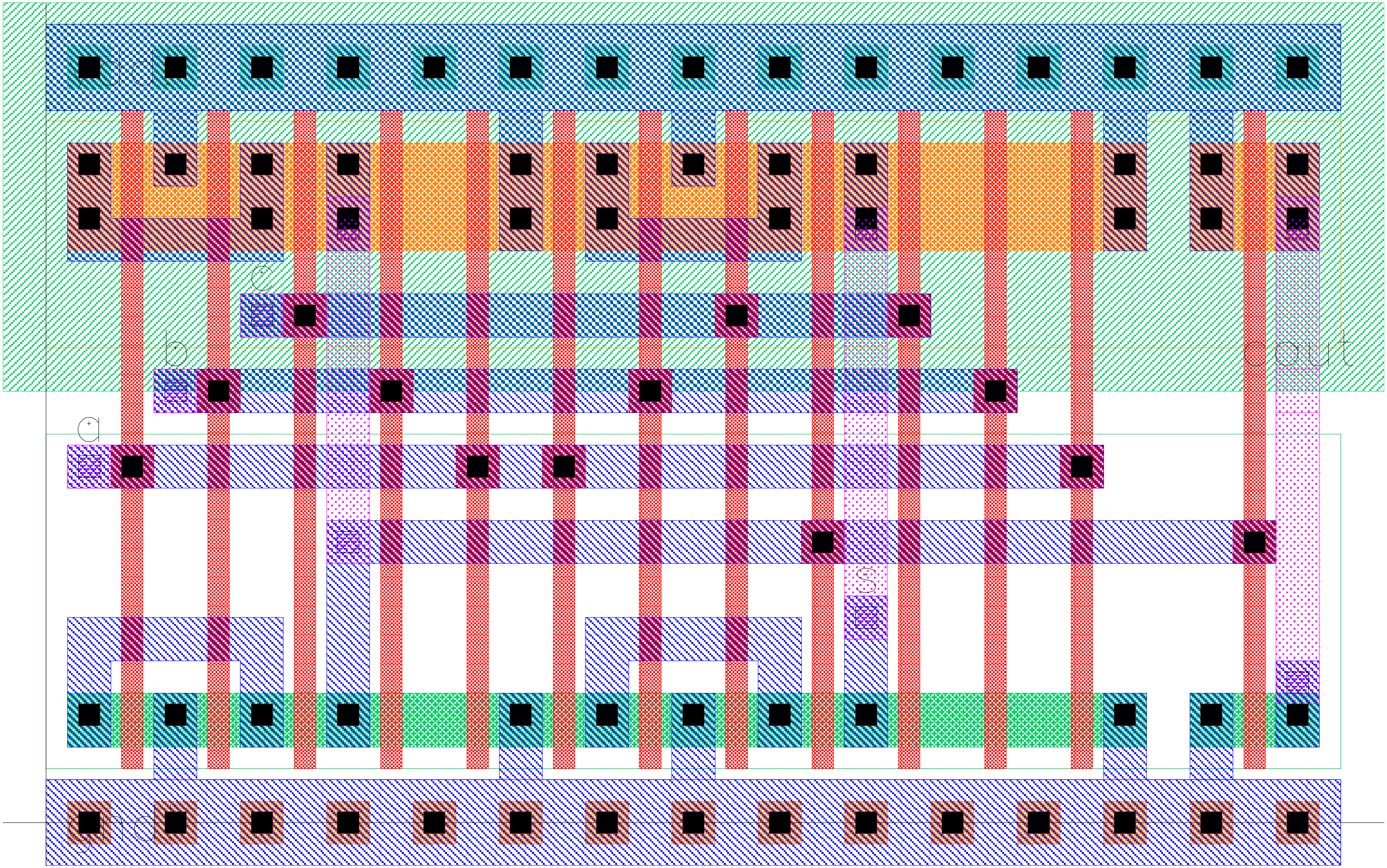
REV

SHEET 1 of 1

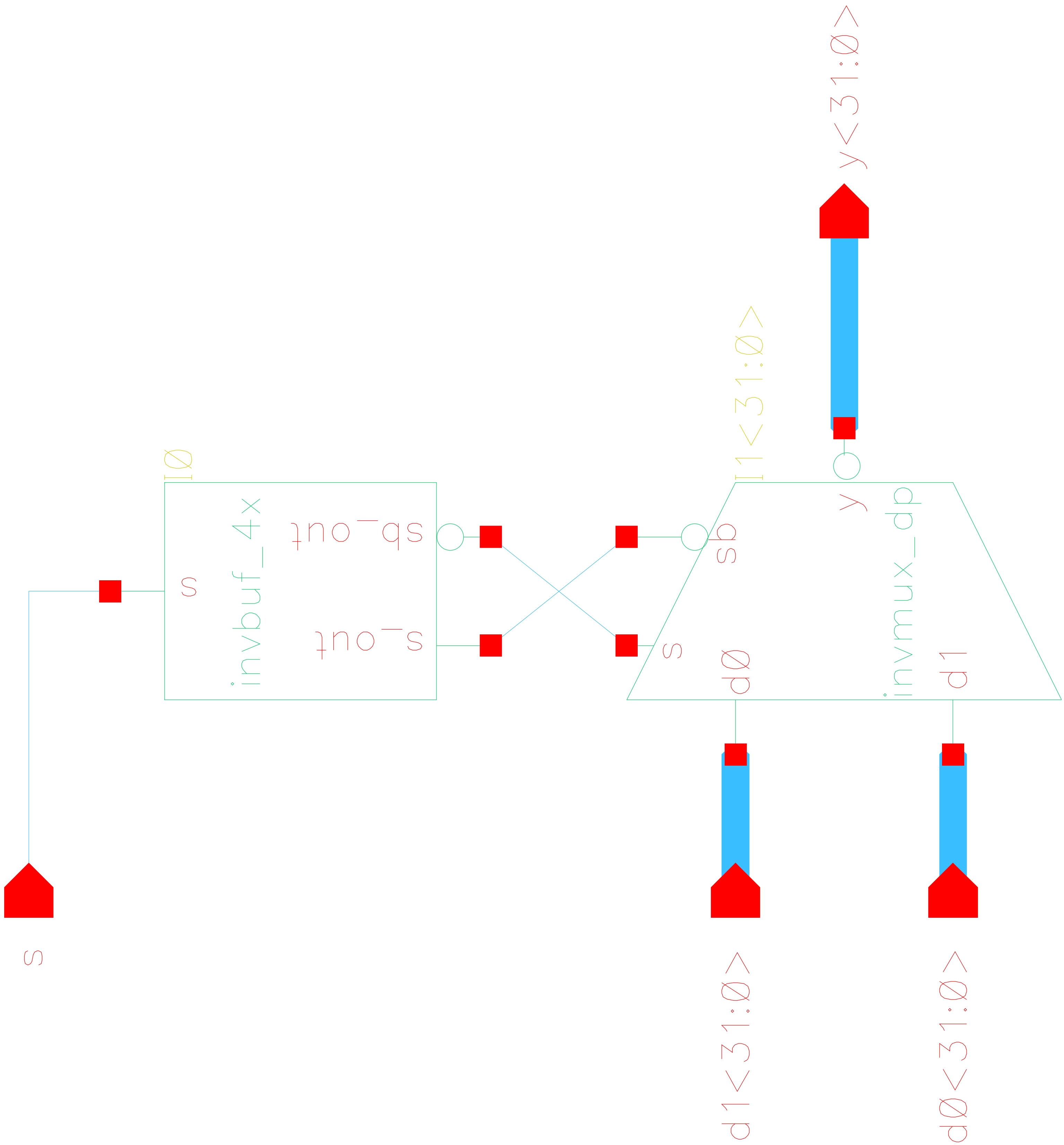


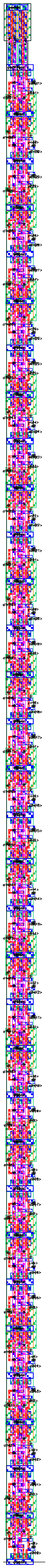
sha1_alu.invfulladder




c
b
a
s
cout

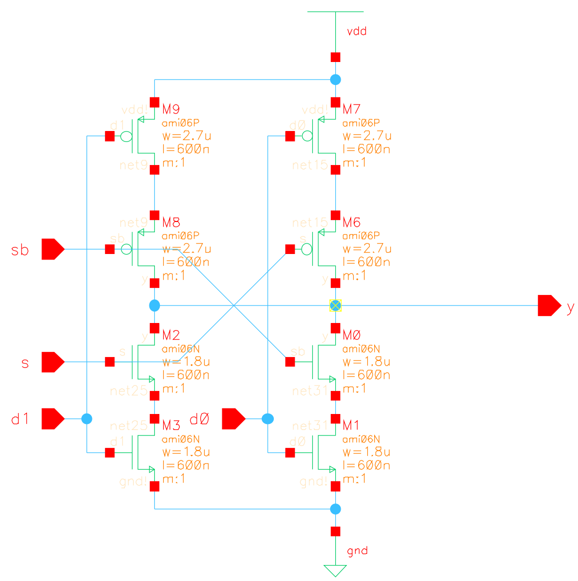
sha1_alu.invmux2_32





sha1_alu.invmux2_dp

REVISIONS			
ZONE	REV	DESCRIPTION	DATE



Harvey Mudd College

UPDATED
Mar 29 23:52:23 2010

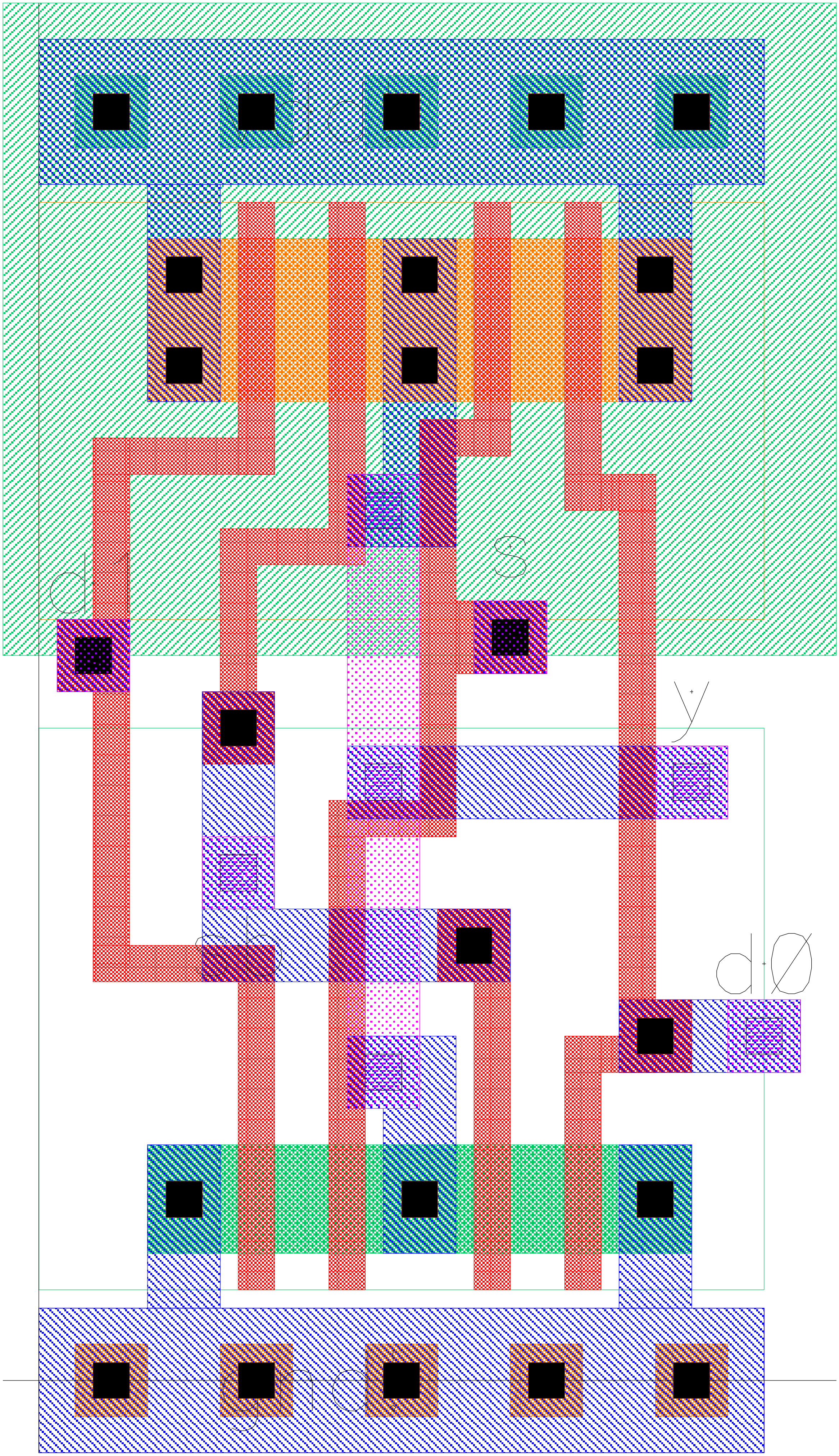
BY
wbuik

SIZE
A

REV

mux2_dp_1x

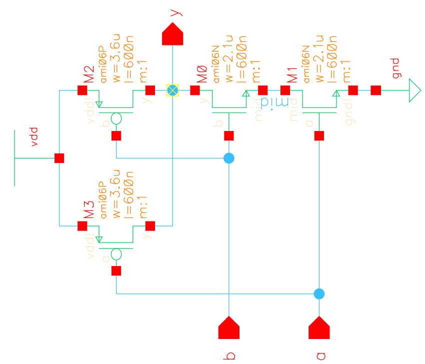
SHEET 1 of 1


s
y
d0

sha1_alu.nand2_1x

ZONE	REV	DESCRIPTION	DATE

REVISIONS



Harvey Mudd College

nand2_1x

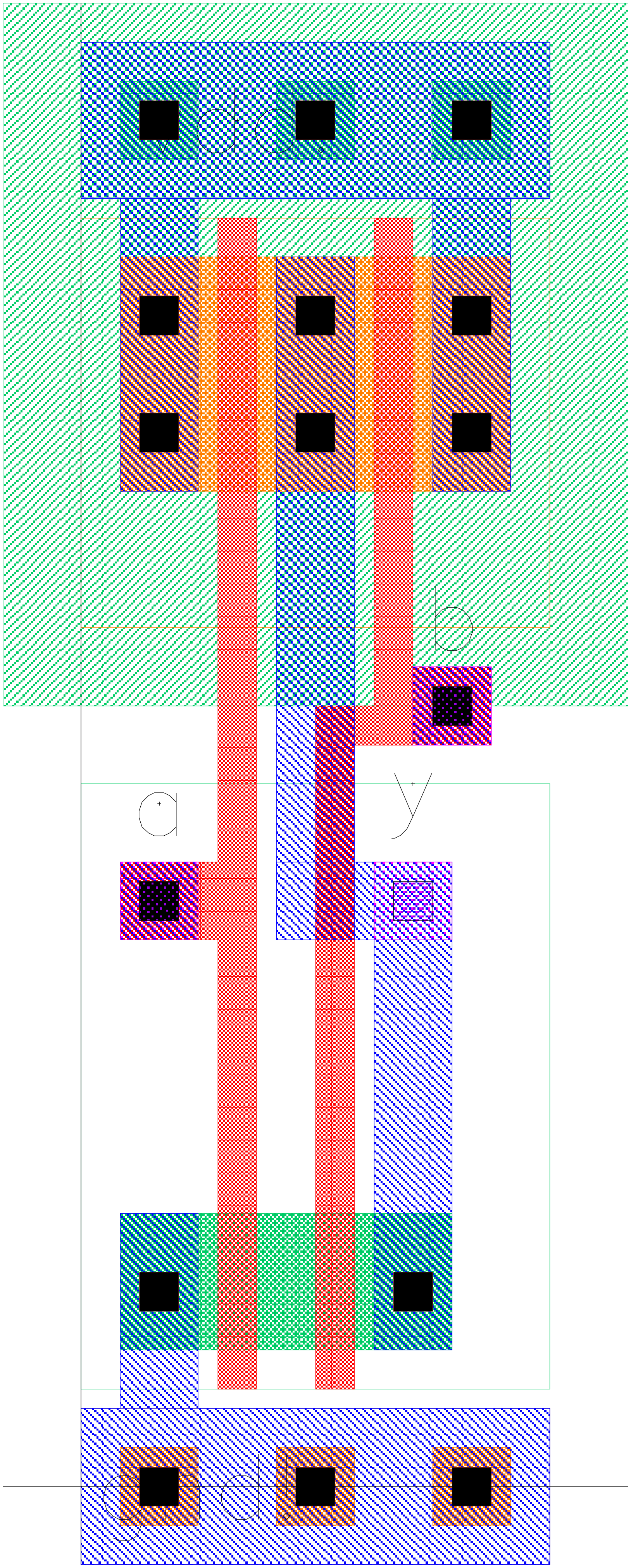
UPDATED Apr 14 02:07:56 2010

BY wbuik

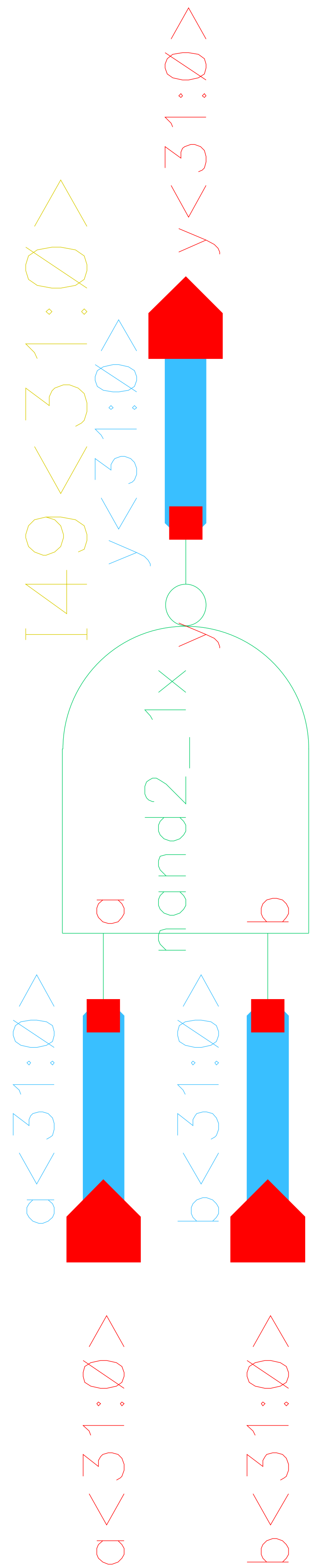
SIZE A

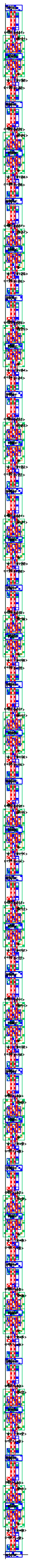
REV

SHEET 1 of 1

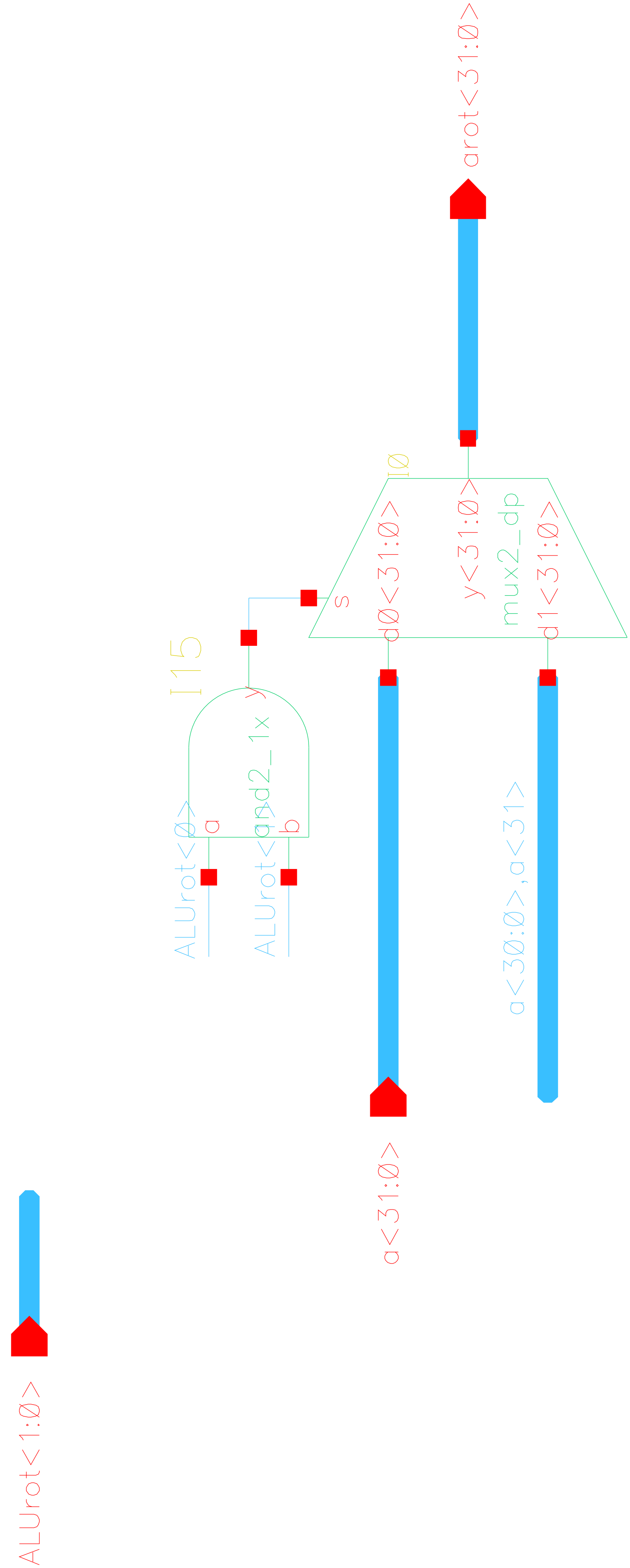

vdd
b
a
y
gnd

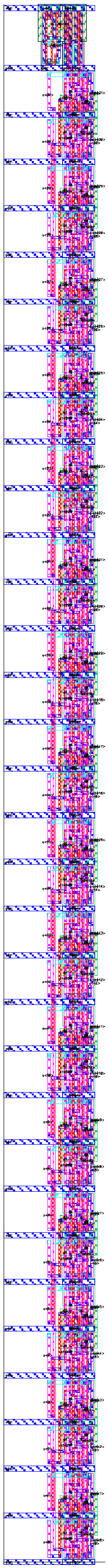
sha1_alu.nand2_32x



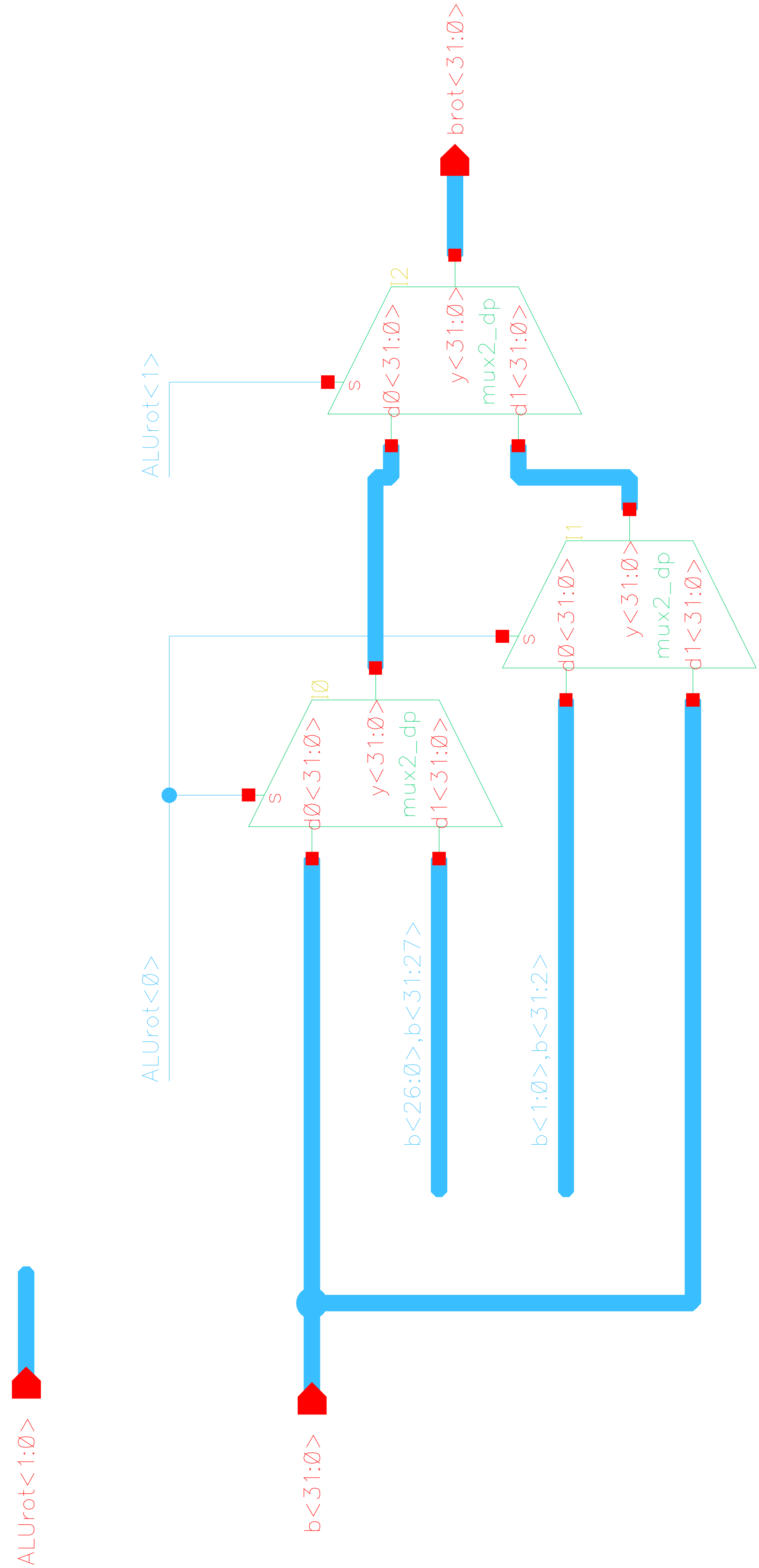


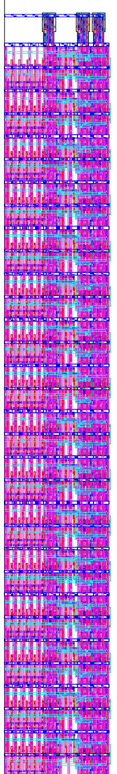
sha1_alu.shift_a



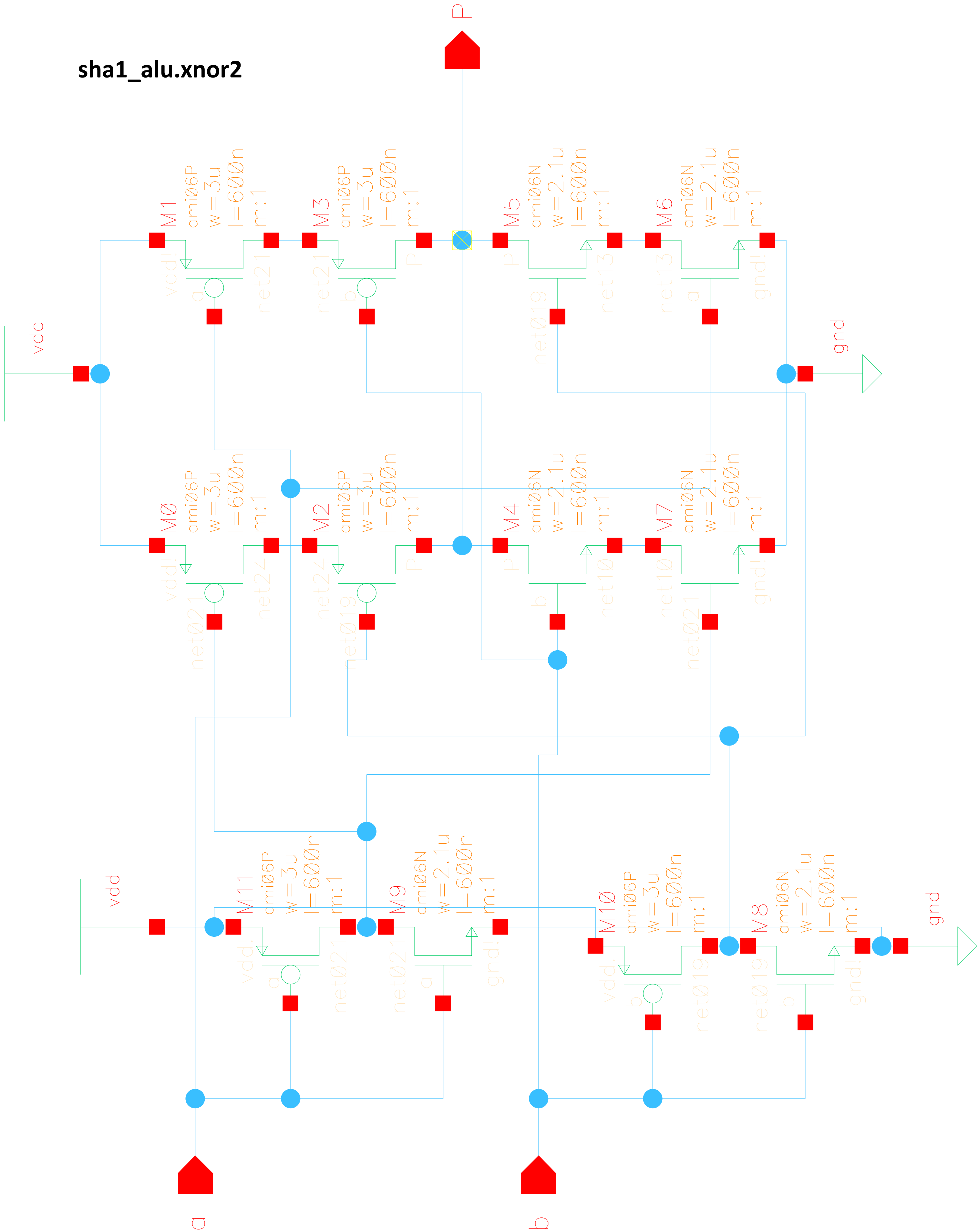


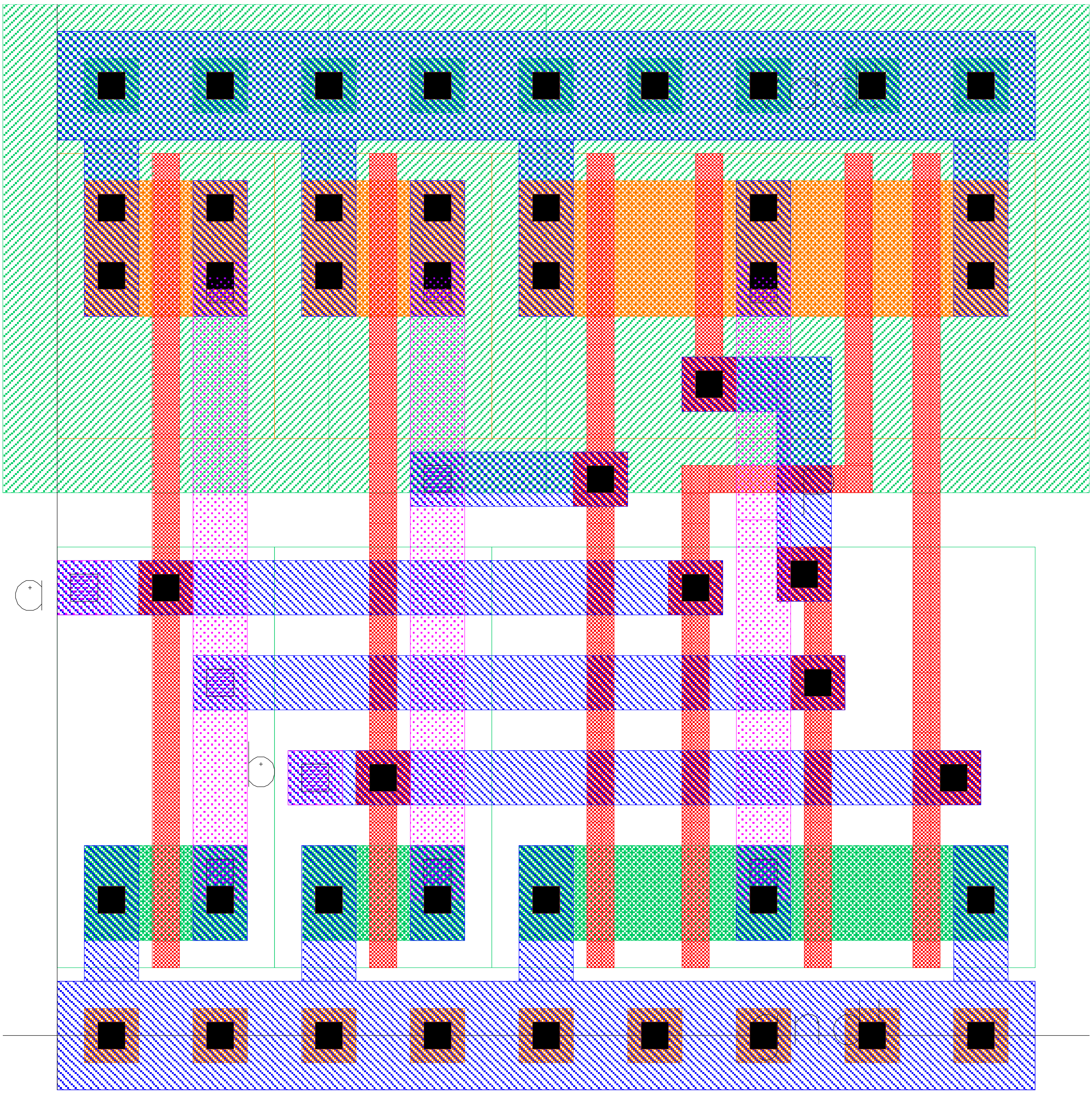
sha1_alu.shift_b



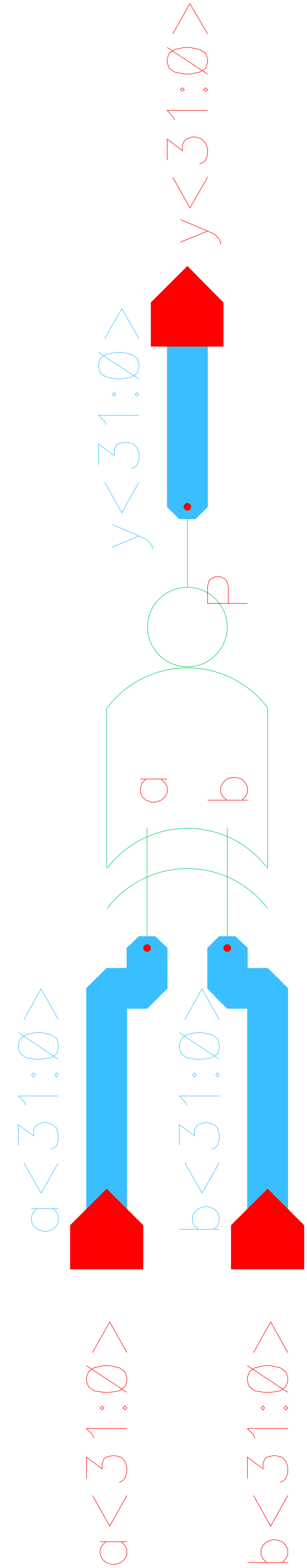


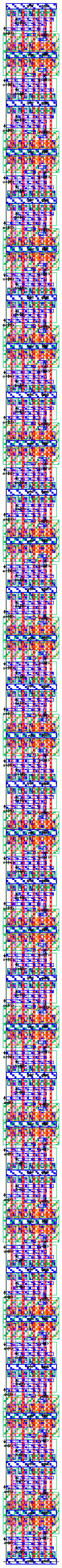
sha1_alu.xnor2





sha1_alu.xnor2_32x





Appendix C (SHA-1 Pseduocode)

```
Initialize variables:
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

Pre-processing:
append the bit '1' to the message
append 0 ≤ k < 512 bits '0', so that the resulting message length (in bits)
   is congruent to 448 ≡ −64 (mod 512)
append length of message (before pre-processing), in bits, as 64-bit big-endian integer

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

    Extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1

    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then
            f = (b and c) xor ((not b) and d)
            k = 0x5A827999
        else if 20 ≤ i ≤ 39
            f = b xor c xor d
            k = 0x6ED9EBA1
        else if 40 ≤ i ≤ 59
            f = (b and c) xor (b and d) xor (c and d)
            k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
            f = b xor c xor d
            k = 0xCA62C1D6

        temp = (a leftrotate 5) + f + e + k + w[i]
        e = d
        d = c
        c = b leftrotate 30
        b = a
        a = temp

    Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian):
digest = hash = h0 append h1 append h2 append h3 append h4
```

Source: http://en.wikipedia.org/wiki/Sha1, accessed 4/19/2010.

Appendix D (Test Bench Generator)

C# code used to calculate the state of all registers for each round of the SHA1 algorithm as the chip implements it. Used to debug the Verilog implementation of the chip and to generate test benches.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SHATest {
    class Program {

        static UInt32 RotateLeft(UInt32 x, int num) {
            for (; num > 0; num--) {
                UInt32 temp = (x >> 31) & 0x00000001;
                x = (x << 1) | temp;
            }
            return x;
        }

        enum OP {
            PASS,
            ADD,
            AND,
            XOR
        }

        enum LROT : int {
            NONE = 0,
            ONE_A = 1,
            FIVE_B = 5,
            THIRTY_B = 30
        }

        static UInt32 ALU(UInt32 a, UInt32 b, OP op, LROT lrot) {

            if(lrot == LROT.FIVE_B || lrot == LROT.THIRTY_B)
                b = RotateLeft(b, (int)lrot);
            if (lrot == LROT.ONE_A)
                a = RotateLeft(a, (int)lrot);

            switch (op) {
                case OP.ADD:
                    return a + b;
                case OP.XOR:
                    return a ^ b;
                case OP.PASS:
                    return b;
                case OP.AND:
                    return a & b;
            }
            return 0;
        }

        static void Main(string[] args) {

            // Memory
            UInt32[] W = new UInt32[16];
            UInt32 A = 0, B = 0, C = 0, D = 0, E = 0, t2 = 0;
            UInt32 H0 = 0, H1 = 0, H2 = 0, H3 = 0, H4 = 0;
            UInt32 t1 = 0;

            UInt32 NULL = 0;

            // Constants
            UInt32 Zero = 0;
            UInt32 AInit = 0x67452301;
            UInt32 BInit = 0xEFCDAB89;
```

```
UInt32 CInit = 0x98BADCFE;
UInt32 DInit = 0x10325476;
UInt32 EInit = 0xC3D2E1F0;
UInt32 K1 = 0x5A827999;
UInt32 K2 = 0x6ED9EBA1;
UInt32 K3 = 0x8F1BBCDC;
UInt32 K4 = 0xCA62C1D6;

// Initilize
//for (int i = 0; i < 16; i++) W[i] = 0xFFFFFFFF;
//W[0] = 0x80000000;
//for (int i = 1; i < 14; i++) W[i] = 0;
//W[14] = 0;
//W[15] = 0;

W[0] = 0x8A921FC4;
W[1] = 0x452C45D2;
W[2] = 0xABC243FE;
W[3] = 0xEC429CBD;
W[4] = 0x452C45D2;
W[5] = 0xEC429CBD;
W[6] = 0x452C45D2;
W[7] = 0xEC429CBD;
W[8] = 0x8A921FC4;
W[9] = 0x452C45D2;
W[10] = 0xABC243FE;
W[11] = 0x452C45D2;
W[12] = 0xEC429CBD;
W[13] = 0x452C45D2;
W[14] = 0x8A921FC4;
W[15] = 0x452C45D2;

//for (int i = 0; i < 16; i++) W[i] = 0x8A921FC4;

H0 = AInit;
H1 = BInit;
H2 = CInit;
H3 = DInit;
H4 = EInit;

A = H0;
B = H1;
C = H2;
D = H3;
E = H4;

for (int i = 0; i < 80; i++) {
    // Crypto
    //



    //Calc f
    if (i >= 0 && i < 20) {
        //F = (B & C) ^ (~B & D);
        t1 = ALU(NULL, D, OP.PASS, LROT.NONE);
        t1 = ALU(t1, C, OP.XOR, LROT.NONE);
        t1 = ALU(t1, B, OP.AND, LROT.NONE);
        t1 = ALU(t1, D, OP.XOR, LROT.NONE);

        t1 = ALU(t1, K1, OP.ADD, LROT.NONE);
    } else if (i >= 20 && i < 40) {
        // F = B ^ C ^ D
        t1 = ALU(NULL, B, OP.PASS, LROT.NONE);
        t1 = ALU(t1, C, OP.XOR, LROT.NONE);
        t1 = ALU(t1, D, OP.XOR, LROT.NONE);

        t1 = ALU(t1, K2, OP.ADD, LROT.NONE);
    } else if (i >= 40 && i < 60) {
        //F = (B & C) ^ (B & D) ^ (C & D);
        t1 = ALU(NULL, C, OP.PASS, LROT.NONE);
```

```
            t2 = ALU(t1, D, OP.AND, LROT.NONE);
            t1 = ALU(NULL, C, OP.PASS, LROT.NONE);
            t1 = ALU(t1, D, OP.XOR, LROT.NONE);
            t1 = ALU(t1, B, OP.AND, LROT.NONE);
            t1 = ALU(t1, t2, OP.XOR, LROT.NONE);

            t1 = ALU(t1, K3, OP.ADD, LROT.NONE);
        } else if (i >= 60 && i < 80) {
            // F = B ^ C ^ D
            t1 = ALU(NULL, B, OP.PASS, LROT.NONE);
            t1 = ALU(t1, C, OP.XOR, LROT.NONE);
            t1 = ALU(t1, D, OP.XOR, LROT.NONE);

            t1 = ALU(t1, K4, OP.ADD, LROT.NONE); //(C & D) + (B & (C # D))
        }

        t1 = ALU(t1, E, OP.ADD, LROT.NONE);
        t1 = ALU(t1, A, OP.ADD, LROT.FIVE_B);
        t1 = ALU(t1, W[0], OP.ADD, LROT.NONE);

        // New state


        E = ALU(NULL, D, OP.PASS, LROT.NONE);
        D = ALU(NULL, C, OP.PASS, LROT.NONE);
        C = ALU(NULL, B, OP.PASS, LROT.THIRTY_B);
        B = ALU(NULL, A, OP.PASS, LROT.NONE);
        A = ALU(t1, Zero, OP.ADD, LROT.NONE); // Pass A

        string aa = String.Format("{0:X8}", A);
        string bb = String.Format("{0:X8}", B);
        string cc = String.Format("{0:X8}", C);
        string dd = String.Format("{0:X8}", D);
        string ee = String.Format("{0:X8}", E);

        //Console.WriteLine(aa + " " + bb + " " + cc + " " + dd + " " + ee);

        // Message Extension
        //
        t1 = ALU(NULL, W[0], OP.PASS, LROT.NONE);
        t1 = ALU(t1, W[2], OP.XOR, LROT.NONE);
        t1 = ALU(t1, W[8], OP.XOR, LROT.NONE);
        t1 = ALU(t1, W[13], OP.XOR, LROT.NONE);
        t1 = ALU(t1, Zero, OP.ADD, LROT.ONE_A);

        // Downshift W
        for (int j = 0; j < 15; j++) {
            W[j] = W[j + 1];
        }
        W[15] = t1;
    }

    t1 = ALU(NULL, A, OP.PASS, LROT.NONE);
    H0 = ALU(t1, H0, OP.ADD, LROT.NONE);

    t1 = ALU(NULL, B, OP.PASS, LROT.NONE);
    H1 = ALU(t1, H1, OP.ADD, LROT.NONE);

    t1 = ALU(NULL, C, OP.PASS, LROT.NONE);
    H2 = ALU(t1, H2, OP.ADD, LROT.NONE);

    t1 = ALU(NULL, D, OP.PASS, LROT.NONE);
    H3 = ALU(t1, H3, OP.ADD, LROT.NONE);

    t1 = ALU(NULL, E, OP.PASS, LROT.NONE);
    H4 = ALU(t1, H4, OP.ADD, LROT.NONE);

    string Ahex = String.Format("{0:X8}", H0);
    string Bhex = String.Format("{0:X8}", H1);
    string Chex = String.Format("{0:X8}", H2);
    string Dhex = String.Format("{0:X8}", H3);
```

```
            string Ehex = String.Format("{0:X8}", H4);

            Console.WriteLine("");
            Console.WriteLine(Ahex + " " + Bhex + " " + Chex + " " + Dhex + " " + Ehex);
            //Console.WriteLine("da39a3ee 5e6b4b0d 3255bfef 95601890 afd80709".ToUpper());

            Console.ReadKey();
        }
    }
}
```