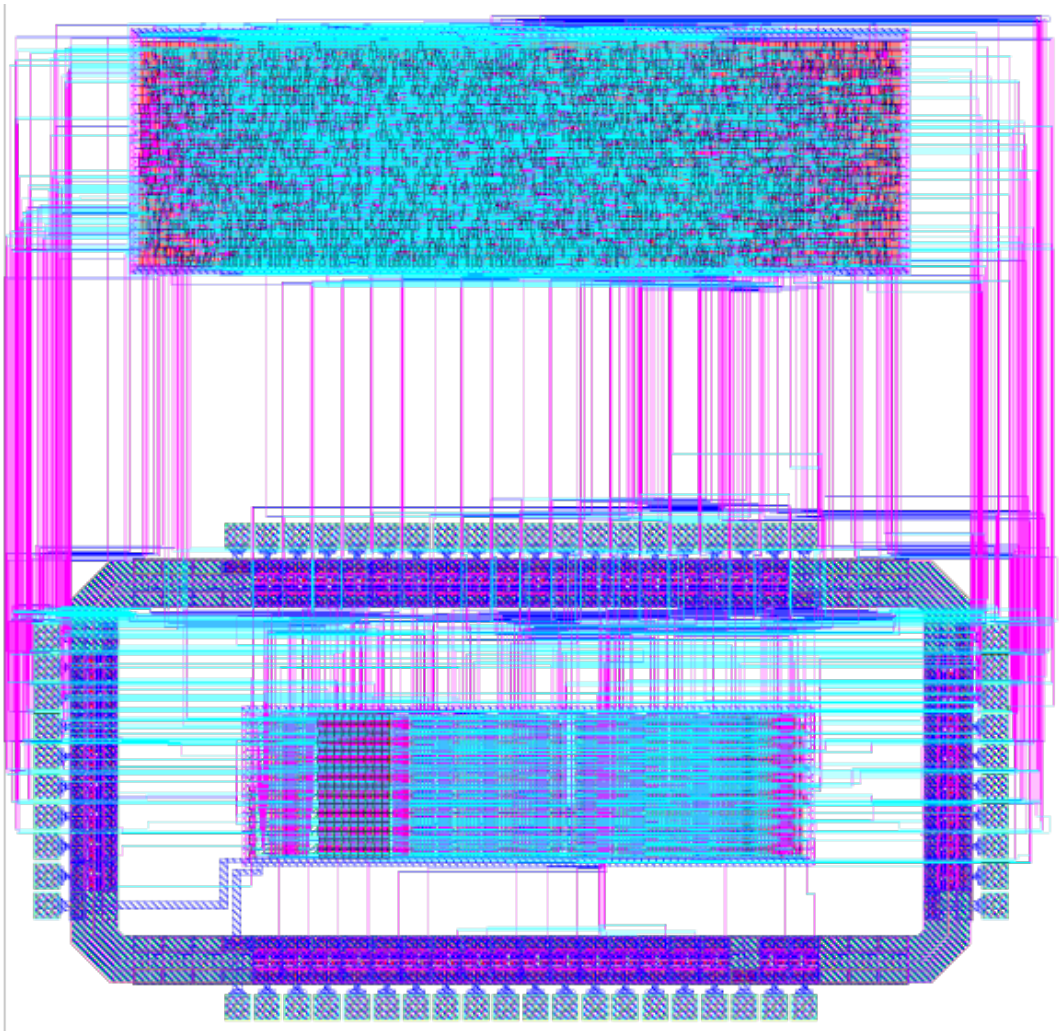
E158 Final Report: Design of a PDP-11 Processor

Andrew Carter, Gregory Fong, James Chris Parks, Eric Zhang

2011-04-17



1 Introduction

The goal of the project was to implement the full instruction set of the PDP-11/40 Processor on a chip. The PDP-11 was a series of 16-bit minicomputers designed and distributed by DEC from the 1970s to the 1990s. The chip would implement the multicycle CISC architecture CPU of the PDP-11 processor. The project will implement the full 16-bit datapath, controller, register file, ALU/shifter, and memory interface.

The chip was designed in the AMI 0.5 micron process ($\lambda = 0.3$ micron) for a 60 pin DIP package.

2 Specifications

2.1 Pinout

The final chip will have a total of 60 I/O pins.

Function (*Names*)	I/O	Bit Width
Memory Address (*MemAddr*)	Output	16
Memory Read Data (*MemReadData*)	Input	16
Memory Write Data (*MemWriteData*)	Output	8
Memory Enable (*MemEn*)	Output	1
Memory Write Enable (*MemWrite*)	Output	1
Byte Instruction (*ReadByte*)	Output	1
Kernel (*Kernal*)	Output	1
Clock (*clk, clkb*)	Input	2
Reset	Input	1
Scan Chain Enable (*scan*)	Input	1
Scan Data In (*sdi*)	Input	1
Scan Data Out / Interrupt Clear (*sdo*)	Output	1
Clock Sense	Output	1
Power (*vdd*)	InOut	4
Ground (*gnd*)	InOut	4
Interrupt	Input	1

2.2 Theory of Operation

The implemented PDP-11 on reset will use the value provided to the Memory Read Data pins as its first Program Counter. The processor will issue memory requests using the Memory Address (The address refered to in the operation) and Memory Enable (Denotes use of memory) pins, with optional use of Memory Write Enable (Denotes write operation to memory) and Memory Write Data (Data to be written to memory) pins in the event of a write to memory. Writes are done in low-byte-first format, and is reflected in the 16-bit value presented by the Memory Address pins. An exception to this is when the Byte Instruction pin is high, in which case only one byte is written to / read from memory. If the Kernel pin is high, the permissions of the memory operation are that of the super-user. Like the Kernel pin, all of the aforementioned memory signals are high-enable.

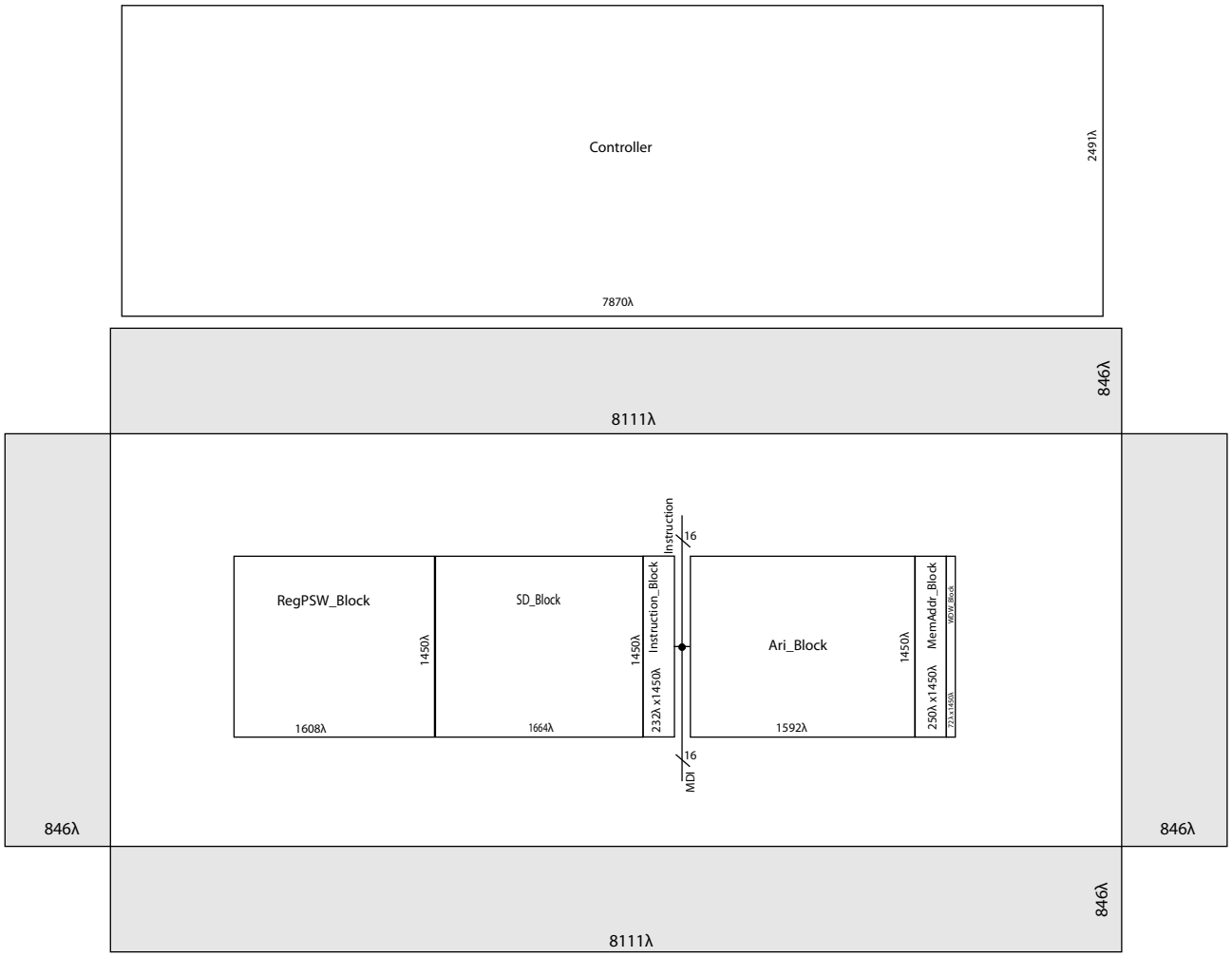
The two clock inputs are required to be non-overlapping in order to ensure proper operation, with a margin between the high-periods of the two clocks adjusted for internal clock skew in the design. The output Clock Sense is the first Clock input routed through the design to potentially detect faulty manufacturing.

The Scan Chain pins manipulate the scan chain in the design, with the Scan Data In pin being the input to the chain and the Scan Data Out pin being the output of the chain. The chain proceeds using both of the provided input clocks. The details of the Scan Chain ordering are provided later in this report.

The interrupt input denotes that an interrupt is arriving from off-chip to the PDP-11. The Scan Data Out pin, during these situations, acts as an Interrupt Clear pin, where, when driven high, denotes that the off-chip controller should consider the interrupt serviced.

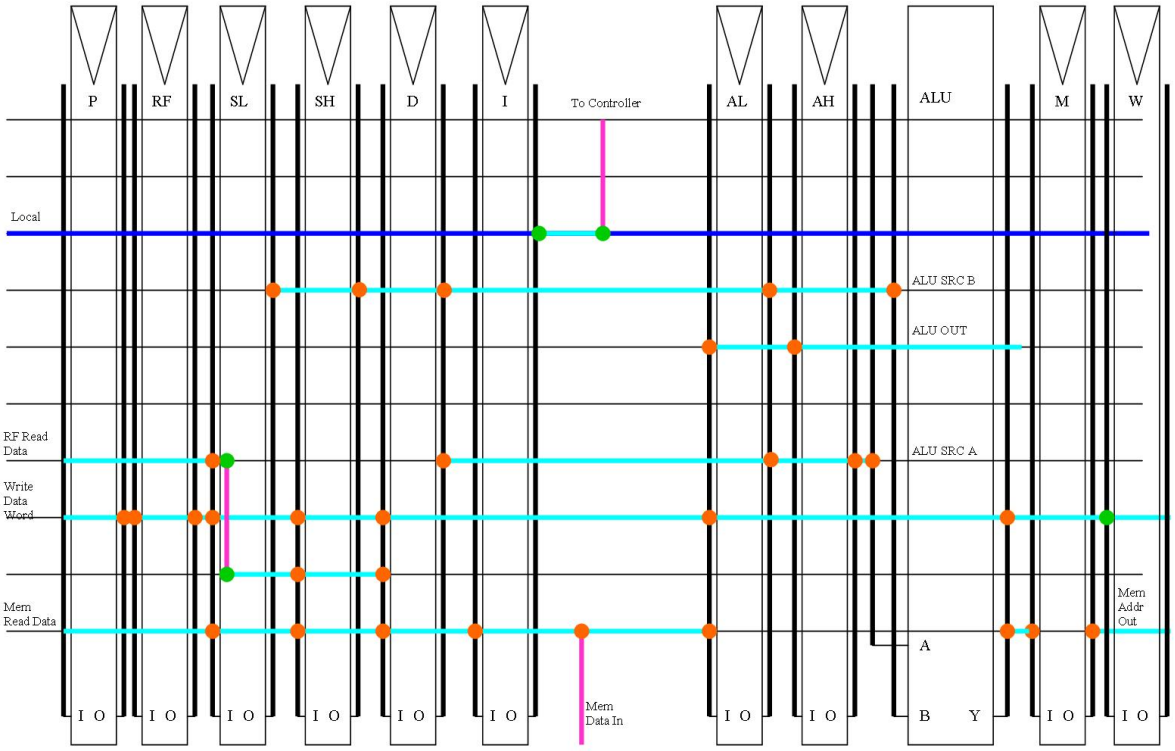
The processor implements the full ISA of the PDP-11/40 architecture. Operation of the processor begins with the on-chip Controller decoding an instruction from memory. The decoding is fully combinational, and drives the operation of the datapath, which, over the course of a finite number of cycles, completes the instruction.

3 Floorplan



The actual floorplan was somewhat different from the original floorplan as the group became more acquainted with the project's requirements. As early estimates of the size of the controller were much larger than expected, the size of the datapath was minimized through the use of inverted power and ground and an alteration to the design to remove the shifter. This added approximately 500λ to the vertical room for the controller. The overall width of the design did not change significantly due to the increasing sizes of the multiplexers offsetting the savings of removing the shifter. Unfortunately, the vertical savings from minimizing the datapath were insufficient to fit the controller in the design.

The sliceplan for the design is below.



4 Verification

- The Verilog passes the testbench.
- The schematics passes the testbench.
- Layout passes DRC. The datapath passes LVS, but the Controller does not pass LVS because the synthesizer routed metal over pins and insufficient time was budgeted for manual rerouting.
- No CIF was generated.

5 Postfabrication Test Plan

The testbench was initially developed to make sure the RTL performed as an actual PDP11.

The basic test programs are stored in *<repository>/test/basic/* in the files ending in **.lf** which stands for load file. This file contains instructions to be run and the state of registers and flags and memory writes after each instruction. This file is parsed and converted into an assembly file (**.af**), a register file which stores the registers and flags (**.tvf**), and a memory output file (**.mf**) by the script *<repository>/test/testLoader.py*. The assembly file is then converted into ascii binary by *<repository>/test/assembler.py*, which creates a file with no extensions. The programs are then run by sim-nc, and the log file stored in **.log**, registers and flags in **.reg**, memory in (**.mem**) and the scanchain vectors are in (**.sc**). The expected and actual reg files and mem files are run through *diff*, and the results are checked with *<repository>/test/checkDiff.py*. This can all be run using the command *make tests* in *<repository>/test/basic/*, and the makefile is generated by *<repository>/test/basic/make.py tests.txt*. A list of all tests is stored in *<repository>/test/basic/tests.txt*.

If the programs do not work on the chip, then scan vectors are provided as **.sc** files generated by *make tests*. These can be used to debug the test at a microcode level.

6 Design Time

Component	*Design Time (Man-hours)*
Proposal	5
RTL	55
Test Vectors	75
Schematics	80
Synthesis	60
Checkoff Rush	10
Layout	60
Testing	60
Final Report	16
Total	421

7 File Locations

All file locations are relative to the repository head.

- Verilog: `test/rtl_new/`
- Test Vectors: `test/basic/`

- Synthesis Results: `synth/` and `soc/`
- Cadence library: `pdp11_lib`, `pdp11_sch_new`
- PDF chip plot: `pdp11_prints/layouts/THE_CHIP.pdf`
- PDF of this report: `pdp11_prints/finalreport.pdf`

References

[1] Gill, Arthur. *Machine and Assembly Language Programming of the PDP-11*. Englewood Cliffs, NJ: Prentice-Hall, 1978. Print.

[2] *PDP11 Processor Handbook: PDP11/04/34a/44/60/70*. [Maynard, Mass.]: Digital Equipment Corporation, 1979. Print.

A Code

A.1 ACC_Hi_Block.v

```
module ACC_Hi_Block(
    input   [15:0]  ALUOut,
    input   [3:0]   ACCHiSel, ACCHiSel_b,
    input           clk, clk_b, clkb, clkb_b, scan, scan_b,
                    reset, sdi,
                    ALUASel, ALUASel_b,
    output  [15:0]  ALUA,
    output          ACCHi0
    );
    wire    [15:0]  ACCHi, ACCHiNext;
    Mux4_dp #(.WIDTH(16)) ACCHiMux(
        .d0(ALUOut),
        .d1(16'h8000),
        .d2({1'b0,ACCHi[15:1]}),
        .d3(ACCHi),
        .sel(ACCHiSel), .sel_b(ACCHiSel_b),
        .q(ACCHiNext)
    );
    scanflop_r #(.WIDTH(16)) ACCHiB(
        .clk(clk), .clk_b(clk_b),
        .clkb(clkb), .clkb_b(clkb_b),
        .scan(scan), .scan_b(scan_b),
        .sdi(sdi), .reset(reset),
        .d(ACCHiNext),
        .q(ACCHi)
    );
    Tri_dp #(.WIDTH(16)) ALUATri(
        .d(ACCHi),
        .en(ALUASel),
        .en_b(ALUASel_b),
        .q(ALUA)
    );
    assign ACCHi0 = ACCHi[0];
endmodule
```

A.2 ACC_Lo_Block.v

```
module ACC_Lo_Block(
    output  [15:0]  WDW,
    input   [15:0]  ALUOut, MRD,
    input   [3:0]   ACCLoSel, ACCLoSel_b,
    input   [2:0]   TrapVal,
    input   [1:0]   WDWSel, WDWSel_b,
    input           clk, clk_b, clkb, clkb_b, scan, scan_b,
                    reset, sdi,
                    ALUASel, ALUASel_b,
                    ALUBSel, ALUBSel_b,
                    MANSel, MANSel_b,
    output  [15:0]  ALUA, ALUB, MAN,
    output          ACCLo0
    );
    wire    [15:0]  ACCLo, ACCLoNext;
```

```
    Mux4_dp #(.WIDTH(16)) ACCLoMux(
        .d0(ALUOut[15:0]),
        .d1({11'b000_0000_0000,TrapVal[2:0], 2'b00}),
        .d2(MRD),
        .d3(ACCLo),
        .sel(ACCLoSel), .sel_b(ACCLoSel_b),
        .q(ACCLoNext)
    );
    scanflop_r #(.WIDTH(16)) ACCLoB(
        .clk(clk), .clk_b(clk_b),
        .clkb(clkb), .clkb_b(clkb_b),
        .scan(scan), .scan_b(scan_b),
        .sdi(sdi), .reset(reset),
        .d(ACCLoNext),
        .q(ACCLo)
    );
    Tri_dp #(.WIDTH(16)) ALUATri(
        .d(ACCLo),
        .en(ALUASel),
        .en_b(ALUASel_b),
        .q(ALUA)
    );
    Tri_dp #(.WIDTH(16)) ALUBTri(
        .d(ACCLo),
        .en(ALUBSel),
        .en_b(ALUBSel_b),
        .q(ALUB)
    );
    Tri_dp #(.WIDTH(16)) MANTri(
        .d(ACCLo),
        .en(MANSel),
        .en_b(MANSel_b),
        .q(MAN)
    );
    Mux2_dp #(.WIDTH(16)) WDWMux(
        .d0(ACCLo),
        .d1({{8{ACCLo[7]}},ACCLo[7:0]}),
        .sel(WDWSel),
        .sel_b(WDWSel_b),
        .q(WDW)
    );
    assign ACCLo0 = ACCLo[0];
endmodule
```

A.3 ALU.v

```
/* ALU Control looks like
 * ALU_Ctrl = {Add,sub,BIC,or,xor,comp}
 * Where BIC is ALU_A & ~ALU_B
 *
 * ALU_A gets DST
 * ALU_B gets SRC
 */
module ALU (input  [15:0] ALU_A, ALU_B,
            input  invertB, // Subtracting B or BIC ALU_Ctrl[3 or 5]
            input  invertA, // Subtracting A. ALU_Ctrl[4]
```

```
            input  add_b,   // Are we NOT adding. ALU_Ctrl[1]
            input  C_in,    // What is the input carry. ALU_Ctrl[4 or 5]
            input  enALU,   // Enable the output of the ALU. This is for ADD, SUB, or XOR. ALU_Ctrl[1, 4, 5, or 6]
            input  enALU_b, // Inverse of above
            input  enAND,   // ALU_Ctrl[7 or 3]
            input  enAND_b, // Inv
            input  enOR,    // ALU_Ctrl[2]
            input  enOR_b,  // Inv
            input  enCOMP,  // ALU_Ctrl[0]
            input  enCOMP_b,// Inv
            output [15:0] ALU_Out,
            output        C_Word,
            output          C_Byte);


    wire [15:0] B_in;
    wire [15:0] A_in;
    wire [15:0] AND_Out;
    wire [15:0] Add_Out;
    wire [15:0] OR_Out;
    wire [15:0] XOR_Out;

    // In truth, the true and complementary pairs are used on
    // dp-inverting tristates, and the logical operations are
    // done in negation (NAND instead of AND, etc) with the
    // exception of the XOR and COMP operations, where an extra
    // inverter ensures correct logic.
    wire Add_Op = enALU & ~enALU_b;
    wire And_Op = enAND & ~enAND_b;
    wire Or_Op  = enOR & ~enOR_b;
    wire Comp_Op = enCOMP & ~enCOMP_b;

    assign B_in = invertB ? ~ALU_B : ALU_B; // Implemented as XOR
    assign A_in = invertA ? ~ALU_A : ALU_A; // Implemented as XOR

    assign AND_Out = A_in & B_in;
    assign OR_Out  = A_in | B_in;
    assign XOR_Out = Add_Out;
    // add_b is high if we're doing an XOR op, and low if we're
    // doing an ADD op (including subtraction).
    Adder16 adder(.a(A_in), .b(B_in), .cin(C_in), .add_b(add_b), .y(Add_Out), .c_word(C_Word),.c_byte(C_Byte));
    Mux4_OneHot #(.WIDTH(16)) out_mux(.d0(Add_Out),.d1(AND_Out),.d2(OR_Out),.d3(~B_in),
                                      .sel0(Add_Op),.sel1(And_Op),.sel2(Or_Op),.sel3(Comp_Op),.q(ALU_Out));


endmodule

module Adder16 (input  [15:0] a, b,
                input  cin,
                input  add_b,
                output [15:0] y,
                output c_word,
                output c_byte);

    wire [16:0] carries;
    wire [15:0] carrybars;
```

```
    assign carries[0] = cin;
    genvar g;
    generate
        for(g=0;g<16;g=g+1) begin : A
            Adder a16(a[g], b[g], carries[g], y[g], carrybars[g]);
            nor2 n16(.a(carrybars[g]), .b(add_b), .y(carries[g+1]));
        end
    endgenerate

    assign c_word = carries[16];
    assign c_byte = carries[8];

endmodule

module Adder(input a,
             input b,
             input cin,
             output y,
             output cout_b);

    minority m(a,b,cin,cout_b);
    assign y = ((cout_b & (a|b|cin)) | (a&b&cin));

endmodule

module minority(input a,
                input b,
                input c,
                output co);

    assign co = ~((c & (a | b)) | (a & b));
endmodule

module nor2(input a,
            input b,
            output y);
    assign y = ~(a | b);
endmodule
```

A.4 Ari_Block.v

```
‘timescale 1ns / 100ps
module Ari_Block(
    inout   [15:0]  ALUA, ALUB, WDW,
    input   [15:0]  MRD,
    input   [7:0]   Instruction,
    input   [3:0]   ALUBSel, ALUBSel_b,
                    ACCLoSel, ACCLoSel_b,
                    ACCHiSel, ACCHiSel_b,
                    MANSel, MANSel_b,
    input   [2:0]   TrapVal,
                    ALUAConst,
                    ALUASel, ALUASel_b,
                    WDWSel, WDWSel_b,
    input           clk, clk_b, clkb, clkb_b, scan, scan_b,
                    reset, sdi,
```

```
                ALUinvertB, ALUinvertA, add_b,
                ALUC_in, enALU, enALU_b, ALUenAND, ALUenAND_b,
                ALUenOR, ALUenOR_b, ALUenCOMP, ALUenCOMP_b,
output  [15:0]  MAN,
output          CWALUOut, CBALUOut, ACCHi0
);
wire    [15:0]  ALUOut;
wire            ACCLo0;


ACC_Lo_Block ALB(
    .WDW(WDW), .ALUOut(ALUOut),
    .MRD(MRD), .ACCLoSel(ACCLoSel), .ACCLoSel_b(ACCLoSel_b),
    .TrapVal(TrapVal), .clk(clk), .clk_b(clk_b), .reset(reset),
    .clkb(clkb), .clkb_b(clkb_b),
    .scan(scan), .scan_b(scan_b), .sdi(sdi),
    .ALUASel(ALUASel[0]), .ALUASel_b(ALUASel_b[0]),
    .ALUBSel(ALUBSel[0]), .ALUBSel_b(ALUBSel_b[0]),
    .MANSel(MANSel[0]), .MANSel_b(MANSel_b[0]),
    .WDWSel(WDWSel[1:0]), .WDWSel_b(WDWSel_b[1:0]),
    .ALUA(ALUA), .ALUB(ALUB), .MAN(MAN), .ACCLo0(ACCLo0)
);
Mux2_dp #(.WIDTH(16)) Imm_ALUB_Tri(
    .d1({{7{Instruction[7]}},Instruction[7:0],1'b0}),
    .d0({9'h000, {6{ALUBSel[3]}} & Instruction[5:0], 1'b0}),
    .sel({ALUBSel[1], ALUBSel[2]}), .sel_b({ALUBSel_b[1], ALUBSel_b[2]}),
    .q(ALUB)
);
Tri_dp #(.WIDTH(16)) MDR_Tri(
    .d(MRD),
    .en(MANSel[1]), .en_b(MANSel_b[1]),
    .q(MAN)
);
ACC_Hi_Block AHB(
    .ALUOut(ALUOut),
    .ACCHiSel(ACCHiSel), .ACCHiSel_b(ACCHiSel_b),
    .clk(clk), .clk_b(clk_b), .reset(reset),
    .clkb(clkb), .clkb_b(clkb_b),
    .scan(scan), .scan_b(scan_b), .sdi(ACCLo0),
    .ALUASel(ALUASel[1]), .ALUASel_b(ALUASel_b[1]),
    .ALUA(ALUA), .ACCHi0(ACCHi0)
);
Tri_dp #(.WIDTH(16)) ALUA_Tri(
    .d({{14{ALUAConst[2]}}, ALUAConst[1], ALUAConst[0]}),
    .en(ALUASel[2]), .en_b(ALUASel_b[2]),
    .q(ALUA)
);
Tri_dp #(.WIDTH(16)) MANB_Tri(
    .d(ALUB),
    .en(MANSel[2]), .en_b(MANSel_b[2]),
    .q(MAN)
);
ALU alu(
    .ALU_A(ALUA), .ALU_B(ALUB),
    .invertB(ALUinvertB), .invertA(ALUinvertA), .add_b(add_b),
    .C_in(ALUC_in), .enALU(enALU), .enALU_b(enALU_b),
```

```
        .enAND(ALUenAND), .enAND_b(ALUenAND_b), .enOR(ALUenOR), .enOR_b(ALUenOR_b),
        .enCOMP(ALUenCOMP), .enCOMP_b(ALUenCOMP_b),
        .ALU_Out(ALUOut), .C_Word(CWALUOut), .C_Byte(CBALUOut)
    );
    Tri_dp #(.WIDTH(16)) ALU_Tri(
        .d(ALUOut),
        .en(MANSel[3]), .en_b(MANSel_b[3]),
        .q(MAN)
    );
    Tri_dp #(.WIDTH(16)) WDW_Tri(
        .d(ALUOut),
        .en(WDWSel[2]), .en_b(WDWSel_b[2]),
        .q(WDW)
    );
endmodule
```

A.5 Controller.h

```
‘define INSTR_CLEAR             4’b0000
‘define INSTR_SINGLE            4’b0001
‘define INSTR_DOUBLE            4’b0010
‘define INSTR_BRANCH            4’b0100
‘define INSTR_SPECIAL_BRANCH    4’b1000

‘define MODE_REG                3’b000
‘define MODE_DREG               3’b001
‘define MODE_INC                3’b010
‘define MODE_DINC               3’b011
‘define MODE_DEC                3’b100
‘define MODE_DDEC               3’b101
‘define MODE_IDX                3’b110
‘define MODE_DIDX               3’b111
```

A.6 Controller.v

```
/* Controller.v
 *
 * Created By William Koven
 *
 * Yeah, good stuff
 *
 */
‘timescale 1ns / 100ps
‘include "States.h"
‘include "pdp11.h"
‘include "Controller.h"

module Controller(input                                 reset,
                  input      [‘STATE_LEN-1:0]           State,
                  input      [15:0]                     Instruction,
                  input      [15:0]                     PSW,
                  input                                 WD_Zero,
                  input                                 WD_Lo_Zero,
                  input                                 WD_Neg,
                  input                                 WD_7,
```

```
                input                                   DST_15,
                input                                   SRC_8,
                input                                   DST_7,
                input                                   DST_0,
                input                                   DST_Zero,
                input                                   SRC_0,
                input                                   SRC_15,
                input                                   SRC_16,
                input                                   SRC_31,
                input                                   Acc_Hi_0,
                input                                   Interrupt,
                output reg [`STATE_LEN-1:0]             Next_State,
                output wire [5:0]                       Next_State_Types,
                input [5:0]                             State_Types,
                output reg [`ALU_CTRL_LEN-1:0]          ALU_Ctrl,
                output reg [`ALU_A_SEL_LEN-1:0]         ALU_A_Sel,
                output reg [`ALU_B_SEL_LEN-1:0]         ALU_B_Sel,
                output reg [`SRC_LO_SEL_LEN-1:0]        SRC_Lo_Sel,
                output reg [`SRC_HI_SEL_LEN-1:0]        SRC_Hi_Sel,
                output reg [`DST_SEL_LEN-1:0]           DST_Sel,
                output reg [`WD_SEL_LEN-1:0]            WD_Sel,
                output reg [`MEM_ADDR_SEL_LEN-1:0]      Mem_Addr_Sel,
                output reg [`ACC_LO_SEL_LEN-1:0]        ACC_Lo_Sel,
                output reg [`ACC_HI_SEL_LEN-1:0]        ACC_Hi_Sel,
                output reg [`RF_ADDR_SEL_LEN-1:0]       RF_Addr_Sel,
                output reg [`SHAMT_SEL_LEN-1:0]         Shamt_Sel,
                output reg [`PSW_SEL_LEN-1:0]           PSW_Sel,
                output reg [`C_SEL_LEN-1:0]             C_Sel,
                output reg [`V_SEL_LEN-1:0]             V_Sel,
                output reg [`Z_SEL_LEN-1:0]             Z_Sel,
                output reg [`N_SEL_LEN-1:0]             N_Sel,
                output reg [`TRAP_SEL_LEN-1:0]          Trap_Sel,
                output reg                              Use_Cin,
                output reg                              Ashc,
                output reg                              Rotate,
                output reg                              SFT_Inv_Shamt,
                output reg [`SFT_SRC_HI_LEN-1:0]        Shift_SRC_Hi,
                output reg [`SFT_SRC_LO_LEN-1:0]        Shift_SRC_Lo,
                output reg                              RF_Enable,
                output reg                              RF_Write_En,
                output reg                              Mem_Enable,
                output reg                              Mem_Write_En,
                output reg                              Mem_Space_Cur,
                output reg                              Priority_Sel,
                output reg                              Interrupt_Clear,
                output reg                              Lo_Hi_Bit,
                output reg                              Lo_Lo_Bit,
                output reg                              Lo_Mid_Bit,
                output                                  Instr_Update_En,
                output                                  Byte,
                output reg                              Read_Byte,
                output reg                              WD_Byte_Sel,
                output reg                              Mem_Addr_Out_Base,
                output                                  MOVB
);
```

```
wire [3:0]                                                    Instr_type;
reg [3:0]                                                    Next_Instr_type;
wire                                                        mult_div_sign;
reg                                                         Next_mult_div_sign;
wire                                                        remainder_sign;
reg                                                            Next_remainder_sign;
wire                                                        Mem_DST;
wire                                                        C_Cur;
wire                                                        V_Cur;
wire                                                        Z_Cur;
wire                                                        N_Cur;
wire                                                        Trace_Trap;
wire [4:0]                                                    Single_Op = Instruction[`SINGLE_OP];
wire [2:0]                                                    Double_Op1 = Instruction[`DOUBLE_OP1];
wire                                                        DST_Neg;
wire                                                        Dec_Byte;
wire                                                        SRC_Reg_0 = Instruction[6], SRC_Reg_5 = Instruction[5];
assign C_Cur = PSW[0];
assign V_Cur = PSW[1];
assign Z_Cur = PSW[2];
assign N_Cur = PSW[3];
assign Trace_Trap = PSW[4];
assign Dec_Byte = Instruction[15] & ((Instr_type == `INSTR_SINGLE &
                                     (Single_Op == `CLR | Single_Op == `COMP | Single_Op == `INC |
                                      Single_Op == `DEC | Single_Op == `NEG | Single_Op == `ADDC |
                                      Single_Op == `SUBC | Single_Op == `TEST | Single_Op == `ROTR |
                                      Single_Op == `ROTL | Single_Op == `ASR | Single_Op == `ASL)) |
                                    (Instr_type == `INSTR_DOUBLE &
               (Double_Op1 == `MOV | Double_Op1 == `CMP | Double_Op1 == `BIT |
                Double_Op1 == `BIC | Double_Op1 == `BIS)));
assign Byte = Dec_Byte & (State != `DST_DEC0 & State != `DST_DDEC0 &
                          State != `SRC_DEC0 & State != `SRC_DDEC0);

assign MOVB = Instruction[15] & (Instr_type == `INSTR_DOUBLE) & (Double_Op1 == `MOV);

assign DST_Neg = Byte ? DST_7 : DST_15;

assign Mem_DST = |Instruction[`DST_MODE];
assign Instr_type = State_Types[5:2];
assign mult_div_sign = State_Types[1];
assign remainder_sign = State_Types[0];
assign Next_State_Types = {Next_Instr_type, Next_mult_div_sign,
                          Next_remainder_sign};


 always @(*) begin
     if (reset) begin
         Next_Instr_type <= `INSTR_CLEAR;
         Next_mult_div_sign <= 1'b0;
         Next_remainder_sign <= 1'b0;
     end
     else begin
         if (State == `FETCH0) begin
             Next_Instr_type <= `INSTR_CLEAR;
         end
         else if (State == `SINGLE0) begin
```

```
            Next_Instr_type <= `INSTR_SINGLE;
        end
        else if (State == `DOUBLE0) begin
            Next_Instr_type <= `INSTR_DOUBLE;
        end
        else if (State == `BRANCH0) begin
            Next_Instr_type <= `INSTR_BRANCH;
        end
        else if (State == `SPECIAL_BRANCH0) begin
            Next_Instr_type <= `INSTR_SPECIAL_BRANCH;
        end
        else if (State == `DIV2) begin
            Next_mult_div_sign <= DST_15 ^ SRC_31;
            Next_remainder_sign <= SRC_31;
        end
        else if (State == `MUL0) begin
            Next_mult_div_sign <= DST_15 ^ SRC_15;
        end
        else begin
            Next_Instr_type <= Instr_type;
            Next_mult_div_sign <= mult_div_sign;
            Next_remainder_sign <= remainder_sign;
        end
    end
end

/* Next State Logic */
always @(*) begin
     case (State)
    `FETCH0: begin
       Next_State = Trace_Trap | Interrupt ? `TRAP0 : `FETCH1;
    end
    `FETCH1: Next_State = `FETCH2;
    `FETCH2: Next_State = `FETCH3;
    `FETCH3: case (Instruction[`PREFIX])
                `PREFIX_SINGLE: Next_State = `SINGLE0;
                `PREFIX_BRANCH: Next_State = `BRANCH0;
                default: Next_State = `DOUBLE0;
             endcase
    `SINGLE0: begin

       if (Instruction[`JUMP_BITS] == `JUMP_INST & Instruction[15]) begin
                    // Next_State = Instruction[`TRAP_BIT] ? `TRAP0 : `EMT0;
                    Next_State = `TRAP0;
             end
             else begin
               case (Instruction[`SINGLE_OP])
                    `MARK:  Next_State = `SINGLE_EX0;
                    `CSM:   Next_State = `SINGLE_EX0;
                    default: case (Instruction[`DST_MODE])
                                `MODE_REG:  Next_State = `DST_REG0;
                                `MODE_DREG: Next_State = `DST_DREG0;
                                `MODE_INC:  Next_State = `DST_INC0;
                                `MODE_DINC: Next_State = `DST_DINC0;
                                `MODE_DEC:  Next_State = `DST_DEC0;
                                `MODE_DDEC: Next_State = `DST_DDEC0;
```

```
                        `MODE_IDX:  Next_State = `DST_IDX0;
                        `MODE_DIDX: Next_State = `DST_DIDX0;
                    endcase
            endcase
        end // else: !if(Instruction[`JUMP_BITS] == `JUMP_INST & Instruction[15])
    end
`DOUBLE0: begin
    if (Instruction[`DOUBLE_OP1] == 3'b111)
            case (Instruction[`DOUBLE_OP2])
                `MUL:       Next_State = `SRC_REG0;
                `DIV:       Next_State = `SRC_REG0;
                `ASH:       Next_State = `SRC_REG0;
                `ASHC:      Next_State = `SRC_REG0;
                `XOR:       Next_State = `SRC_REG0;
                `FLOP:      Next_State = `BROKEN;
                `SYS:       Next_State = `BROKEN;
                `SOB:       Next_State = `SOB0;
            endcase
        else
            case (Instruction[`SRC_MODE])
                `MODE_REG:  Next_State = `SRC_REG0;
                `MODE_DREG: Next_State = `SRC_DREG0;
                `MODE_INC:  Next_State = `SRC_INC0;
                `MODE_DINC: Next_State = `SRC_DINC0;
                `MODE_DEC:  Next_State = `SRC_DEC0;
                `MODE_DDEC: Next_State = `SRC_DDEC0;
                `MODE_IDX:  Next_State = `SRC_IDX0;
                `MODE_DIDX: Next_State = `SRC_DIDX0;
            endcase // case (Instruction[`SRC_MODE])
    end
`DOUBLE1:   if (Instruction[`DOUBLE_OP1] == 3'b111) begin
            case (Instruction[`DOUBLE_OP2])
                `ASH:       Next_State = `DOUBLE_EX0;
                `ASHC:      Next_State = `DOUBLE_EX0;
                default: case (Instruction[`DST_MODE])
                            `MODE_REG:  Next_State = `DST_REG0;
                            `MODE_DREG: Next_State = `DST_DREG0;
                            `MODE_INC:  Next_State = `DST_INC0;
                            `MODE_DINC: Next_State = `DST_DINC0;
                            `MODE_DEC:  Next_State = `DST_DEC0;
                            `MODE_DDEC: Next_State = `DST_DDEC0;
                            `MODE_IDX:  Next_State = `DST_IDX0;
                            `MODE_DIDX: Next_State = `DST_DIDX0;
                        endcase
            endcase
        end
        else begin
            case (Instruction[`DST_MODE])
                `MODE_REG:  Next_State = `DST_REG0;
                `MODE_DREG: Next_State = `DST_DREG0;
                `MODE_INC:  Next_State = `DST_INC0;
                `MODE_DINC: Next_State = `DST_DINC0;
                `MODE_DEC:  Next_State = `DST_DEC0;
                `MODE_DDEC: Next_State = `DST_DDEC0;
                `MODE_IDX:  Next_State = `DST_IDX0;
                `MODE_DIDX: Next_State = `DST_DIDX0;
```

```
                endcase
            end
`DST_REG0:  Next_State = `EXECUTE0;
`DST_DREG0: Next_State = `DST_DREG1;
`DST_DREG1: Next_State = `EXECUTE0;
`DST_INC0:  Next_State = `DST_INC1;
`DST_INC1:  Next_State = `EXECUTE0;
`DST_DINC0: Next_State = `DST_DINC1;
`DST_DINC1: Next_State = `DST_DINC2;
`DST_DINC2: Next_State = `EXECUTE0;
`DST_DEC0:  Next_State = `DST_DEC1;
`DST_DEC1:  Next_State = `EXECUTE0;
`DST_DDEC0: Next_State = `DST_DDEC1;
`DST_DDEC1: Next_State = `DST_DDEC2;
`DST_DDEC2: Next_State = `EXECUTE0;
`DST_IDX0:  Next_State = `DST_IDX1;
`DST_IDX1:  Next_State = `DST_IDX2;
`DST_IDX2:  Next_State = `DST_IDX3;
`DST_IDX3:  Next_State = `EXECUTE0;
`DST_DIDX0: Next_State = `DST_DIDX1;
`DST_DIDX1: Next_State = `DST_DIDX2;
`DST_DIDX2: Next_State = `DST_DIDX3;
`DST_DIDX3: Next_State = `DST_DIDX4;
`DST_DIDX4: Next_State = `EXECUTE0;
// SRC_REG States
`SRC_REG0:  Next_State = `DOUBLE1;
`SRC_DREG0: Next_State = `SRC_DREG1;
`SRC_DREG1: Next_State = `DOUBLE1;
`SRC_INC0:  Next_State = `SRC_INC1;
`SRC_INC1:  Next_State = `DOUBLE1;
`SRC_DINC0: Next_State = `SRC_DINC1;
`SRC_DINC1: Next_State = `SRC_DINC2;
`SRC_DINC2: Next_State = `DOUBLE1;
`SRC_DEC0:  Next_State = `SRC_DEC1;
`SRC_DEC1:  Next_State = `DOUBLE1;
`SRC_DDEC0: Next_State = `SRC_DDEC1;
`SRC_DDEC1: Next_State = `SRC_DDEC2;
`SRC_DDEC2: Next_State = `DOUBLE1;
`SRC_IDX0:  Next_State = `SRC_IDX1;
`SRC_IDX1:  Next_State = `SRC_IDX2;
`SRC_IDX2:  Next_State = `SRC_IDX3;
`SRC_IDX3:  Next_State = `DOUBLE1;
`SRC_DIDX0: Next_State = `SRC_DIDX1;
`SRC_DIDX1: Next_State = `SRC_DIDX2;
`SRC_DIDX2: Next_State = `SRC_DIDX3;
`SRC_DIDX3: Next_State = `SRC_DIDX4;
`SRC_DIDX4: Next_State = `DOUBLE1;
`EXECUTE0:  case (Instr_type)
                `INSTR_SINGLE: Next_State = `SINGLE_EX0;
                `INSTR_BRANCH: Next_State = `BRANCH_EX0;
                `INSTR_DOUBLE: Next_State = `DOUBLE_EX0;
                `INSTR_SPECIAL_BRANCH:  Next_State = Instruction[7] ? `SWAB0 : `JMP0;
                default: Next_State = `BROKEN;
            endcase
`SINGLE_EX0: if (Instruction[`JUMP_BITS] == `JUMP_INST) begin
                if (Instruction[15]) begin
```

```
                    // Next_State = Instruction[`TRAP_BIT] ? `TRAP0 : `EMT0;
                    Next_State = `TRAP0;
                end
                else
                    Next_State = `JSR0;
            end
            else begin
                case (Instruction[`SINGLE_OP])
                    //`SWAB:  Next_State = `SWAB0;
                    `CLR:   Next_State = `WRITE_BACK0;
                    `COMP:  Next_State = `WRITE_BACK0;
                    `INC:   Next_State = `WRITE_BACK0;
                    `DEC:   Next_State = `WRITE_BACK0;
                    `NEG:   Next_State = `WRITE_BACK0;
                    `ADDC:  Next_State = `WRITE_BACK0;
                    `SUBC:  Next_State = `WRITE_BACK0;
                    `TEST:  Next_State = `WRITE_BACK0;
                    `ROTR:  Next_State = `ROTR0;
                    `ROTL:  Next_State = `ROTL0;
                    `ASR:   Next_State = `ASR0;
                    `ASL:   Next_State = `ASL0;
                    `MARK:  Next_State = `MARK0;
                    //`MFPI:  Next_State = Byte ? `MFPD0 : `MFPI0;
                    //`MTPI:  Next_State = Byte ? `MTPD0 : `MTPI0;
                    `MFPI:  Next_State = `MFPI0;
                    `MTPI:  Next_State = `MTPI0;
                    `SXT:   Next_State = `WRITE_BACK0;
                    `CSM:   Next_State = `BROKEN;
                    default: Next_State = `BROKEN;
                endcase
            end
 `DOUBLE_EX0: case (Instruction[`DOUBLE_OP1])
                `MOV:   Next_State = `WRITE_BACK0;
                `CMP:   Next_State = `FETCH0;
                `BIT:   Next_State = `FETCH0;
                `BIC:   Next_State = `WRITE_BACK0;
                `BIS:   Next_State = `WRITE_BACK0;
                `ADD:   Next_State = `WRITE_BACK0;
                `REG:   case (Instruction[`DOUBLE_OP2])
                            `MUL: Next_State = `MUL0;
                            `DIV: Next_State = `DIV0;
                            `ASH: Next_State =  `ASH0;
                            `ASHC: Next_State = `ASHC0;
                            `XOR: Next_State = `WRITE_BACK0;
                            `FLOP: Next_State = `BROKEN;
                            `SYS: Next_State = `BROKEN;
                            `SOB: Next_State = `SOB0;
                        endcase
                default: Next_State = `BROKEN;
              endcase
`BRANCH0: begin
   if (~Instruction[15]) begin
      case (Instruction[`BRANCH_OP])
        `SYSB:  Next_State = `SPECIAL_BRANCH0;
        `BR:    Next_State = `BRANCH_EX0;
        `BNE:   Next_State = `BRANCH_EX0;
```

```
          `BEQ:   Next_State = `BRANCH_EX0;
          `BGE:   Next_State = `BRANCH_EX0;
          `BLT:   Next_State = `BRANCH_EX0;
          `BGT:   Next_State = `BRANCH_EX0;
          `BLE:   Next_State = `BRANCH_EX0;
        endcase
      end
      else begin
        case (Instruction[`BRANCH_OP])
          `BPL:   Next_State = `BRANCH_EX0;
          `BMI:   Next_State = `BRANCH_EX0;
          `BHI:   Next_State = `BRANCH_EX0;
          `BLOS:  Next_State = `BRANCH_EX0;
          `BVC:   Next_State = `BRANCH_EX0;
          `BVS:   Next_State = `BRANCH_EX0;
          `BCC:   Next_State = `BRANCH_EX0;
          `BCS:   Next_State = `BRANCH_EX0;
        endcase
      end
    end
  `BRANCH_EX0:    Next_State = `FETCH0;

    `SPECIAL_BRANCH0: begin
      if (Instruction[6]) // JMP
        case(Instruction[`DST_MODE])
          `MODE_REG:  Next_State = `DST_REG0;
          `MODE_DREG: Next_State = `DST_DREG0;
          `MODE_INC:  Next_State = `DST_INC0;
          `MODE_DINC: Next_State = `DST_DINC0;
          `MODE_DEC:  Next_State = `DST_DEC0;
          `MODE_DDEC: Next_State = `DST_DDEC0;
          `MODE_IDX:  Next_State = `DST_IDX0;
          `MODE_DIDX: Next_State = `DST_DIDX0;
        endcase
      else if (Instruction[7])
        Next_State = Instruction[5] ? `SCC0 : (Instruction[4] ? `SPL0 : `RTS0);
      else
        case (Instruction[2:0])
          `HALT:  Next_State = `HALT0;
          `WAIT:  Next_State = `WAIT0;
          `RTI:   Next_State = `RTT0;
          `BPT:   Next_State = `TRAP0;
          `IOT:   Next_State = `TRAP0;
          `NSI:   Next_State = `BROKEN;
          `RTT:   Next_State = `RTT0;
          `MFPT:  Next_State = `MFPT0;
        endcase
    end
    /* The following will probably take more than one cycle */
    `SINGLE_SFT0:   Next_State = `WRITE_BACK0;
    `ROTR0:         Next_State = `SINGLE_SFT0;
    `ROTL0:         Next_State = `SINGLE_SFT0;
    `ASR0:          Next_State = `SINGLE_SFT0;
    `ASL0:          Next_State = `SINGLE_SFT0;

    `MARK0:     Next_State = `MARK1;
```

```
  `MARK1:     Next_State = `MARK2;
  `MARK2:     Next_State = `MARK3;
  `MARK3:     Next_State = `MARK4;
  `MARK4:     Next_State = `MARK5;
  `MARK5:     Next_State = `MARK6;
  `MARK6:     Next_State = `FETCH0;

  `MFPI0:     Next_State = `MFPI1;
  `MFPI1:     Next_State = `MFPI2;
  `MFPI2:     Next_State = `MFPI3;
  `MFPI3:     Next_State = `MFPI4;
  `MFPI4:     Next_State = `FETCH1;

  //`MFPI0:     Next_State = `FETCH0;
  `MTPI0:     Next_State = `MTPI1;
  `MTPI1:     Next_State = `MTPI2;
  `MTPI2:     Next_State = `MTPI3;
  `MTPI3:     Next_State = `MTPI4;
  `MTPI4:     Next_State = `FETCH0;

  //`MTPI0:     Next_State = `FETCH0;
  /* Sign extend will only take 1 cycle */
  /* Call to Supervisor mode may take more than 1 cycle */
  //`CSM0:      Next_State = `FETCH0;

  `SWAB0:      Next_State = `SWAB1;
  `SWAB1:      Next_State = DST_Zero ? `SWAB4 : `SWAB2;
  `SWAB2:      Next_State = `SWAB3;
  `SWAB3:      Next_State = `SWAB1;
  `SWAB4:      Next_State = `WRITE_BACK0;

  /* Double Operand Instructions */
  /* MUL, DIV, ASHC will take multiple cycles */
  `MUL0: begin
     Next_State = `MUL1;
  end
  `MUL1:      Next_State = `MUL2;
  `MUL2:      Next_State = `MUL3;
  `MUL3:      Next_State = DST_Zero ? `MUL6 : `MUL4;
  `MUL4:      Next_State = `MUL5;
  `MUL5:      Next_State = `MUL3;
  `MUL6:      Next_State = `MUL7;
  `MUL7:      Next_State = `MUL8;
  `MUL8:      Next_State = mult_div_sign ? `MUL9 : `MUL11;
  `MUL9:      Next_State = `MUL10;
  `MUL10:     Next_State = `MUL11;
  `MUL11:     Next_State = `MUL12;
  `MUL12:     Next_State = `FETCH0;

  /* Divide !!! */
  `DIV0:      Next_State = Instruction[6] ? `BROKEN : `DIV1;
  `DIV1:      Next_State = `DIV2;
  `DIV2: begin
     Next_State = DST_15 ? `DIV3 : `DIV4;
  end
`DIV3:      Next_State = `DIV4;
```

```
‘DIV4:      Next_State = SRC_31 ? ‘DIV5 : ‘DIV8;
‘DIV5:      Next_State = ‘DIV6;
‘DIV6:      Next_State = ‘DIV7;
‘DIV7:      Next_State = ‘DIV8;
‘DIV8:      Next_State = DST_Zero ? ‘FETCH0 : ‘DIV9;
‘DIV9:      Next_State = WD_Neg ? ‘DIV10 : ‘FETCH0;
‘DIV10:     Next_State = ‘DIV11;
‘DIV11:     Next_State = Acc_Hi_0 ? ‘DIV12 : ‘DIV10;
‘DIV12:     Next_State = ‘DIV13;
‘DIV13:     Next_State = ‘DIV14;
‘DIV14:     Next_State = ‘DIV15;
‘DIV15:     Next_State = ‘FETCH0;

‘ASH0:      Next_State = DST_Zero ? ‘ASH3 : ‘ASH1;
‘ASH1:      Next_State = ‘ASH2;
‘ASH2:      Next_State = ‘ASH0;
‘ASH3:      Next_State = ‘REG_WRITE_BACK0;

‘ASHC0:     Next_State = SRC_Reg_0 ? ‘BROKEN : ‘ASHC1;
‘ASHC1:     Next_State = DST_Zero ? ‘ASHC4 : ‘ASHC2;
‘ASHC2:     Next_State = ‘ASHC3;
‘ASHC3:     Next_State = ‘ASHC1;
‘ASHC4:     Next_State = ‘ASHC5;
‘ASHC5:     Next_State = ‘ASHC6;
‘ASHC6:     Next_State = ‘FETCH0;
/* The rest of double ops will only take 1 cycle */
/* Subtract One and Branch if not Zero, SOB */
‘SOB0:      Next_State = ‘SOB1;
‘SOB1:      Next_State = WD_Zero ? ‘FETCH0 : ‘SOB2;
‘SOB2:      Next_State = ‘SOB3;
‘SOB3:      Next_State = ‘FETCH0;

/*Set Condition Codes, SCC */
/* Only 1 cycle to set condition codes */
‘SCC0:      Next_State = ‘FETCH0;

/* Jump and subroutine instructions */
/* Jump and traps should only take 1 cycle */
‘JMP0:      Next_State = ‘JMP1;
‘JMP1:      Next_State = ‘FETCH0;

//‘BPT0:      Next_State = ‘FETCH0;
//‘IOT0:      Next_State = ‘FETCH0;
‘TRAP0:     Next_State = ‘TRAP1;
‘TRAP1:     Next_State = ‘TRAP2;
‘TRAP2:     Next_State = ‘TRAP3;
‘TRAP3:     Next_State = ‘TRAP4;
‘TRAP4:     Next_State = ‘TRAP5;
‘TRAP5:     Next_State = ‘TRAP6;
‘TRAP6:     Next_State = ‘TRAP7;
‘TRAP7:     Next_State = ‘TRAP8;
‘TRAP8:     Next_State = ‘TRAP9;
‘TRAP9:     Next_State = ‘TRAP10;
‘TRAP10:    Next_State = ‘TRAP11;
‘TRAP11:    Next_State = ‘FETCH0;
```

```
    //‘EMT0:      Next_State = ‘FETCH0;
    /* JSR and return instructions will take multiple cycles */
    /* Jump Subroutine, or JSR */
    ‘JSR0:      Next_State = ‘JSR1;
    ‘JSR1:      Next_State = ‘JSR2;
    ‘JSR2:      Next_State = ‘JSR3;
    ‘JSR3:      Next_State = ‘JSR4;
    ‘JSR4:      Next_State = ‘JSR5;
    ‘JSR5:      Next_State = ‘JSR6;
    ‘JSR6:      Next_State = ‘FETCH0;
    /* Return from Subroutine, RTS */
    ‘RTS0:      Next_State = ‘RTS1;
    ‘RTS1:      Next_State = ‘RTS2;
    ‘RTS2:      Next_State = ‘RTS3;
    ‘RTS3:      Next_State = ‘RTS4;
    ‘RTS4:      Next_State = ‘FETCH0;

    //‘RTI0:      Next_State = ‘FETCH0;
    ‘RTT0:      Next_State = ‘RTT1;
    ‘RTT1:      Next_State = ‘RTT2;
    ‘RTT2:      Next_State = Instruction[2] ? ‘FETCH1 : ‘FETCH0;
    ‘SPL0:      Next_State = ‘FETCH0;

    /* Other Special Instructions */
    ‘HALT0:     Next_State = ‘HALT0;
    ‘WAIT0:     Next_State = ‘WAIT0;
    ‘MFPT0:     Next_State = ‘FETCH0;

    ‘WRITE_BACK0:   case (Byte)
                        1’b0: Next_State = Mem_DST ? ‘WRITE_BACK1 : ‘FETCH0;
                        1’b1: Next_State = ‘FETCH0;
                        default: Next_State = ‘FETCH0;
                    endcase
    ‘WRITE_BACK1:   Next_State = ‘FETCH0;

    ‘REG_WRITE_BACK0:   Next_State = ‘FETCH0;

    ‘BROKEN:    Next_State = ‘BROKEN;

    /* DONE!!! If need to default, broke!!!*/
    default:    Next_State = ‘BROKEN;
endcase
end // DONE Next State Logic

/* Select Logic */
always @(*) begin
    /* ALU Source A Select Logic */
    case (State)
        ‘FETCH1:        ALU_A_Sel = ‘alu_a_two;
        // DST Fetch Time
        ‘DST_INC0:      ALU_A_Sel = (Instruction[‘DST_REG] == ‘PC | Instruction[‘DST_REG] == ‘SP) ? ‘alu_a_two
        ‘DST_DINC0:     ALU_A_Sel = ‘alu_a_two;
        ‘DST_DEC0:      ALU_A_Sel = (Instruction[‘DST_REG] == ‘PC | Instruction[‘DST_REG] == ‘SP) ? ‘alu_a_two
        ‘DST_DDEC0:     ALU_A_Sel = ‘alu_a_two;
        ‘DST_IDX0:      ALU_A_Sel = ‘alu_a_two;
        ‘DST_IDX2:      ALU_A_Sel = ‘alu_a_dst;
```

```
‘DST_DIDX0:     ALU_A_Sel = ‘alu_a_two;
‘DST_DIDX2:     ALU_A_Sel = ‘alu_a_dst;
// SRC Fetch Time
‘SRC_INC0:      ALU_A_Sel = (Instruction[‘SRC_REG] == ‘PC | Instruction[‘SRC_REG] == ‘SP) ? ‘alu_a_two
‘SRC_DINC0:     ALU_A_Sel = ‘alu_a_two;
‘SRC_DEC0:      ALU_A_Sel = (Instruction[‘SRC_REG] == ‘PC | Instruction[‘SRC_REG] == ‘SP) ? ‘alu_a_two
‘SRC_DDEC0:     ALU_A_Sel = ‘alu_a_two;
‘SRC_IDX0:      ALU_A_Sel = ‘alu_a_two;
‘SRC_IDX2:      ALU_A_Sel = ‘alu_a_dst;
‘SRC_DIDX0:     ALU_A_Sel = ‘alu_a_two;
‘SRC_DIDX2:     ALU_A_Sel = ‘alu_a_dst;
// Execute Instructions
‘SINGLE_EX0:    case (Instruction[‘SINGLE_OP])
                    ‘CLR:   ALU_A_Sel = ‘alu_a_zero;
                    ‘COMP:  ALU_A_Sel = ‘alu_a_zero;
                    ‘INC:   ALU_A_Sel = ‘alu_a_one;
                    ‘DEC:   ALU_A_Sel = ‘alu_a_one;
                    ‘NEG:   ALU_A_Sel = ‘alu_a_zero;
                    ‘ADDC:  ALU_A_Sel = C_Cur ? ‘alu_a_one : ‘alu_a_zero;
                    ‘SUBC:  ALU_A_Sel = C_Cur ? ‘alu_a_one : ‘alu_a_zero;
                    ‘TEST:  ALU_A_Sel = ‘alu_a_zero;
                    ‘ROTR:  ALU_A_Sel = ‘alu_a_zero;
                    ‘ROTL:  ALU_A_Sel = ‘alu_a_zero;
                    ‘ASR:   ALU_A_Sel = ‘alu_a_zero;
                    ‘ASL:   ALU_A_Sel = ‘alu_a_zero;
                    ‘MARK:  ALU_A_Sel = ‘alu_a_zero;
                    ‘MFPI:  ALU_A_Sel = ‘alu_a_zero;
                    ‘MTPI:  ALU_A_Sel = ‘alu_a_zero;
                    ‘SXT:   ALU_A_Sel = N_Cur ? ‘alu_a_ones : ‘alu_a_zero;
                    ‘CSM:   ALU_A_Sel = ‘alu_a_zero;
                   default: ALU_A_Sel = ‘alu_a_zero;
                endcase

‘DOUBLE_EX0: case (Instruction[‘DOUBLE_OP1])
    ‘MOV:   ALU_A_Sel = ‘alu_a_zero;
    ‘CMP:   ALU_A_Sel = ‘alu_a_dst;
    ‘BIT:   ALU_A_Sel = ‘alu_a_dst;
    ‘BIC:   ALU_A_Sel = ‘alu_a_dst;
    ‘BIS:   ALU_A_Sel = ‘alu_a_dst;
    ‘ADD:   ALU_A_Sel = ‘alu_a_dst;
    ‘REG:   case (Instruction[‘DOUBLE_OP2])
                 ‘ASH: ALU_A_Sel = ‘alu_a_zero;
                 ‘ASHC: ALU_A_Sel = ‘alu_a_zero;
                 ‘XOR: ALU_A_Sel = ‘alu_a_dst;
                 default: ALU_A_Sel = ‘alu_a_zero;
            endcase
        default: ALU_A_Sel = ‘alu_a_zero;
    endcase
‘SINGLE_SFT0:   ALU_A_Sel = ‘alu_a_zero;
‘SWAB0:         ALU_A_Sel = ‘alu_a_zero;
‘SWAB3:         ALU_A_Sel = ‘alu_a_one;
‘SWAB4:         ALU_A_Sel = ‘alu_a_zero;
// Branch
‘BRANCH0:       ALU_A_Sel = ‘alu_a_rf;
// Jump
‘JMP0:          ALU_A_Sel = ‘alu_a_dst;
```

```
// JSR
`JSR0:          ALU_A_Sel = `alu_a_two;
`JSR2:          ALU_A_Sel = `alu_a_zero;
`JSR4:          ALU_A_Sel = `alu_a_zero;
`JSR5:          ALU_A_Sel = `alu_a_zero;
// RTS
`RTS0:          ALU_A_Sel = `alu_a_zero;
`RTS2:          ALU_A_Sel = `alu_a_two;
// MARK
`MARK0:         ALU_A_Sel = `alu_a_rf;
`MARK2:         ALU_A_Sel = `alu_a_rf;
`MARK4:         ALU_A_Sel = `alu_a_two;
// TRAP
`TRAP1:         ALU_A_Sel = `alu_a_two;
`TRAP4:         ALU_A_Sel = `alu_a_two;
`TRAP5:         ALU_A_Sel = `alu_a_zero;
`TRAP6:         ALU_A_Sel = `alu_a_zero;
`TRAP7:         ALU_A_Sel = `alu_a_two;
`TRAP8:         ALU_A_Sel = `alu_a_zero;
`TRAP9:         ALU_A_Sel = `alu_a_zero;
`TRAP10:        ALU_A_Sel = `alu_a_zero;
// Return from Trap (RTT/RTI)
`RTT0:          ALU_A_Sel = `alu_a_two;
`RTT1:          ALU_A_Sel = `alu_a_two;
// Subtract One and Branch if not zero
`SOB0:          ALU_A_Sel = `alu_a_one;
`SOB2:          ALU_A_Sel = `alu_a_rf;
// ASH
`ASH0:          ALU_A_Sel = `alu_a_zero;
`ASH2:          ALU_A_Sel = `alu_a_one;
`ASH3:          ALU_A_Sel = `alu_a_zero;
// ASHC
`ASHC3:         ALU_A_Sel = `alu_a_one;
`ASHC4:         ALU_A_Sel = `alu_a_zero;
`ASHC5:         ALU_A_Sel = `alu_a_zero;
// Multiply
`MUL0:          ALU_A_Sel = `alu_a_zero;
`MUL1:          ALU_A_Sel = `alu_a_zero;
`MUL2:          ALU_A_Sel = `alu_a_zero;
`MUL3:          ALU_A_Sel = `alu_a_acc_lo;
`MUL4:          ALU_A_Sel = `alu_a_acc_hi;
`MUL6:          ALU_A_Sel = `alu_a_zero;
`MUL8:          ALU_A_Sel = `alu_a_acc_hi;
`MUL9:          ALU_A_Sel = `alu_a_zero;
`MUL10:         ALU_A_Sel = C_Cur ? `alu_a_one : `alu_a_zero;
`MUL12:         ALU_A_Sel = `alu_a_zero;
// Divide
`DIV2:          ALU_A_Sel = `alu_a_zero;
`DIV3:          ALU_A_Sel = `alu_a_dst;
`DIV5:          ALU_A_Sel = `alu_a_zero;
`DIV6:          ALU_A_Sel = `alu_a_zero;
`DIV7:          ALU_A_Sel = C_Cur ? `alu_a_one : `alu_a_zero;
`DIV8:          ALU_A_Sel = `alu_a_dst;
`DIV9:          ALU_A_Sel = `alu_a_dst;
`DIV10:         ALU_A_Sel = `alu_a_dst;
`DIV11:         ALU_A_Sel = `alu_a_acc_hi;
```

```
    `DIV12:         ALU_A_Sel = `alu_a_acc_lo;
    `DIV14:         ALU_A_Sel = `alu_a_zero;
    // Memory Ops
    // Move from prev
    `MFPI0:         ALU_A_Sel = `alu_a_zero;
    `MFPI1:         ALU_A_Sel = `alu_a_two;
    `MFPI2:         ALU_A_Sel = `alu_a_zero;
    `MFPI3:         ALU_A_Sel = `alu_a_zero;
    // Move to prev
    `MTPI0:         ALU_A_Sel = `alu_a_two;
    `MTPI1:         ALU_A_Sel = `alu_a_zero;
    `MTPI2:         ALU_A_Sel = `alu_a_zero;
    `MTPI3:         ALU_A_Sel = `alu_a_zero;
    default:        ALU_A_Sel = `alu_a_zero;
endcase // End ALU Source A Select Logic

/* ALU Source B Select Logic */
case (State)
    `FETCH1:        ALU_B_Sel = `alu_b_rf;
    // DST Fetch Time
    `DST_INC0:      ALU_B_Sel = `alu_b_rf;
    `DST_DINC0:     ALU_B_Sel = `alu_b_rf;
    `DST_DEC0:      ALU_B_Sel = `alu_b_rf;
    `DST_DDEC0:     ALU_B_Sel = `alu_b_rf;
    `DST_DREG0:     ALU_B_Sel = `alu_b_rf;
    `DST_IDX0:      ALU_B_Sel = `alu_b_rf;
    `DST_IDX2:      ALU_B_Sel = `alu_b_rf;
    `DST_DIDX0:     ALU_B_Sel = `alu_b_rf;
    `DST_DIDX2:     ALU_B_Sel = `alu_b_rf;
    // SRC Fetch Time
    `SRC_INC0:      ALU_B_Sel = `alu_b_rf;
    `SRC_DINC0:     ALU_B_Sel = `alu_b_rf;
    `SRC_DEC0:      ALU_B_Sel = `alu_b_rf;
    `SRC_DDEC0:     ALU_B_Sel = `alu_b_rf;
    `SRC_DREG0:     ALU_B_Sel = `alu_b_rf;
    `SRC_IDX0:      ALU_B_Sel = `alu_b_rf;
    `SRC_IDX2:      ALU_B_Sel = `alu_b_rf;
    `SRC_DIDX0:     ALU_B_Sel = `alu_b_rf;
    `SRC_DIDX2:     ALU_B_Sel = `alu_b_rf;
    // Single Execute Instructions
    `SINGLE_EX0:    case (Instruction[`SINGLE_OP])
                        `SXT:   ALU_B_Sel = `alu_b_zero;
                        default: ALU_B_Sel = `alu_b_dst;
                    endcase
    // Double Execute Instructions
    `DOUBLE_EX0: case (Instruction[`DOUBLE_OP1])
        `MOV:   ALU_B_Sel = `alu_b_src_lo;
        `CMP:   ALU_B_Sel = `alu_b_src_lo;
        `BIT:   ALU_B_Sel = `alu_b_src_lo;
        `BIC:   ALU_B_Sel = `alu_b_src_lo;
        `BIS:   ALU_B_Sel = `alu_b_src_lo;
        `ADD:   ALU_B_Sel = `alu_b_src_lo;
        `REG:   case (Instruction[`DOUBLE_OP2])
                     `ASH: ALU_B_Sel = `alu_b_zero;
                     `XOR: ALU_B_Sel = `alu_b_src_lo;
                     default: ALU_B_Sel = `alu_b_zero;
```

```
            endcase
        default: ALU_B_Sel = `alu_b_zero;
    endcase
`SINGLE_SFT0:   ALU_B_Sel = `alu_b_src_lo;
`SWAB0:         ALU_B_Sel = `alu_b_dst;
`SWAB3:         ALU_B_Sel = `alu_b_dst;
`SWAB4:         ALU_B_Sel = `alu_b_src_lo;
// Branch
`BRANCH0:       ALU_B_Sel = `alu_b_branch;
// Jump
`JMP0:          ALU_B_Sel = `alu_b_zero;
// JSR
`JSR0:          ALU_B_Sel = `alu_b_rf;
`JSR2:          ALU_B_Sel = `alu_b_rf;
`JSR4:          ALU_B_Sel = `alu_b_rf;
`JSR5:          ALU_B_Sel = `alu_b_dst;
// RTS
`RTS0:          ALU_B_Sel = `alu_b_rf;
`RTS2:          ALU_B_Sel = `alu_b_rf;
// MARK
`MARK0:         ALU_B_Sel = `alu_b_sob;
`MARK2:         ALU_B_Sel = `alu_b_zero;
`MARK4:         ALU_B_Sel = `alu_b_rf;
// Trap
`TRAP1:         ALU_B_Sel = `alu_b_acc_lo;
`TRAP4:         ALU_B_Sel = `alu_b_rf;
`TRAP5:         ALU_B_Sel = `alu_b_dst;
`TRAP6:         ALU_B_Sel = `alu_b_dst;
`TRAP7:         ALU_B_Sel = `alu_b_acc_lo;
`TRAP8:         ALU_B_Sel = `alu_b_rf;
`TRAP9:         ALU_B_Sel = `alu_b_rf;
`TRAP10:        ALU_B_Sel = `alu_b_src_lo;
// Return from TRAP!!!!! (RTT/RTI)
`RTT0:          ALU_B_Sel = `alu_b_rf;
`RTT1:          ALU_B_Sel = `alu_b_acc_lo;
// SOB
`SOB0:          ALU_B_Sel = `alu_b_rf;
`SOB2:          ALU_B_Sel = `alu_b_sob;
// ASH
`ASH0:          ALU_B_Sel = `alu_b_zero;
`ASH2:          ALU_B_Sel = `alu_b_dst;
`ASH3:          ALU_B_Sel = `alu_b_src_lo;
// ASHC
`ASHC3:         ALU_B_Sel = `alu_b_dst;
`ASHC4:         ALU_B_Sel = `alu_b_src_lo;
`ASHC5:         ALU_B_Sel = `alu_b_src_hi;
// Multiply
`MUL0:          ALU_B_Sel = `alu_b_zero;
`MUL1:          ALU_B_Sel = `alu_b_dst;
`MUL2:          ALU_B_Sel = `alu_b_src_lo;
`MUL3:          ALU_B_Sel = `alu_b_src_lo;
`MUL4:          ALU_B_Sel = `alu_b_src_hi;
`MUL6:          ALU_B_Sel = `alu_b_acc_lo;
`MUL8:          ALU_B_Sel = `alu_b_zero;
`MUL9:          ALU_B_Sel = `alu_b_acc_lo;
`MUL10:         ALU_B_Sel = `alu_b_acc_lo;
```

```
    `MUL12:         ALU_B_Sel = `alu_b_acc_lo;
    // Divide
    `DIV2:          ALU_B_Sel = `alu_b_zero;
    `DIV3:          ALU_B_Sel = `alu_b_zero;
    `DIV5:          ALU_B_Sel = `alu_b_src_lo;
    `DIV6:          ALU_B_Sel = `alu_b_src_hi;
    `DIV7:          ALU_B_Sel = `alu_b_src_hi;
    `DIV8:          ALU_B_Sel = `alu_b_zero;
    `DIV9:          ALU_B_Sel = `alu_b_src_hi;
    `DIV10:         ALU_B_Sel = `alu_b_src_hi;
    `DIV11:         ALU_B_Sel = `alu_b_acc_lo;
    `DIV12:         ALU_B_Sel = `alu_b_zero;
    `DIV14:         ALU_B_Sel = `alu_b_src_hi;
    // Memory Ops
    `MFPI0:         ALU_B_Sel = `alu_b_dst;
    `MFPI1:         ALU_B_Sel = `alu_b_rf;
    `MFPI2:         ALU_B_Sel = `alu_b_src_lo;
    `MFPI3:         ALU_B_Sel = `alu_b_src_lo;
    // move to
    `MTPI0:         ALU_B_Sel = `alu_b_rf;
    `MTPI1:         ALU_B_Sel = `alu_b_dst;
    `MTPI2:         ALU_B_Sel = `alu_b_src_lo;
    `MTPI3:         ALU_B_Sel = `alu_b_src_lo;
    default:        ALU_B_Sel = `alu_b_zero;
endcase // End ALU Source B Select


/* ALU Control Selection Logic */
case (State)
    `FETCH1:        ALU_Ctrl = `alu_add;
    // DST Fetch
    `DST_INC0:      ALU_Ctrl = `alu_add;
    `DST_DINC0:     ALU_Ctrl = `alu_add;
    `DST_DEC0:      ALU_Ctrl = `alu_sub_a;
    `DST_DDEC0:     ALU_Ctrl = `alu_sub_a;
    `DST_IDX0:      ALU_Ctrl = `alu_add;
    `DST_IDX2:      ALU_Ctrl = `alu_add;
    `DST_DIDX0:     ALU_Ctrl = `alu_add;
    `DST_DIDX2:     ALU_Ctrl = `alu_add;
    // SRC Fetch
    `SRC_INC0:      ALU_Ctrl = `alu_add;
    `SRC_DINC0:     ALU_Ctrl = `alu_add;
    `SRC_DEC0:      ALU_Ctrl = `alu_sub_a;
    `SRC_DDEC0:     ALU_Ctrl = `alu_sub_a;
    `SRC_IDX0:      ALU_Ctrl = `alu_add;
    `SRC_IDX2:      ALU_Ctrl = `alu_add;
    `SRC_DIDX0:     ALU_Ctrl = `alu_add;
    `SRC_DIDX2:     ALU_Ctrl = `alu_add;
    // Single Execute Instructions
    `SINGLE_EX0: case (Instruction[`SINGLE_OP])
                        `CLR:   ALU_Ctrl = `alu_and;
                        `COMP:  ALU_Ctrl = `alu_comp;
                        `INC:   ALU_Ctrl = `alu_add;
                        `DEC:   ALU_Ctrl = `alu_sub_a;
                        `NEG:   ALU_Ctrl = `alu_sub_b;
                        `ADDC:  ALU_Ctrl = `alu_add;
```

```
                    `SUBC:  ALU_Ctrl = `alu_sub_a;
                    `TEST:  ALU_Ctrl = `alu_add;
                    `ROTR:  ALU_Ctrl = `alu_add;
                    `ROTL:  ALU_Ctrl = `alu_add;
                    `ASR:   ALU_Ctrl = `alu_add;
                    `ASL:   ALU_Ctrl = `alu_add;
                    `MARK:  ALU_Ctrl = `alu_add;
                    `MFPI:  ALU_Ctrl = `alu_add;
                    `MTPI:  ALU_Ctrl = `alu_add;
                    `SXT:   ALU_Ctrl = `alu_add;
                    `CSM:   ALU_Ctrl = `alu_add;
                    default: ALU_Ctrl = {`ALU_CTRL_LEN{1'bx}};
                endcase
`DOUBLE_EX0: case (Instruction[`DOUBLE_OP1])
    `MOV:   ALU_Ctrl = `alu_add;
    `CMP:   ALU_Ctrl = `alu_sub_a;
    `BIT:   ALU_Ctrl = `alu_and;
    `BIC:   ALU_Ctrl = `alu_bic;
    `BIS:   ALU_Ctrl = `alu_or;
    `ADD:   ALU_Ctrl = Instruction[15] ? `alu_sub_b : `alu_add;
    `REG:   case (Instruction[`DOUBLE_OP2])
                `ASH: ALU_Ctrl = `alu_add;
                `XOR: ALU_Ctrl = `alu_xor;
                default: ALU_Ctrl = {`ALU_CTRL_LEN{1'bx}};
            endcase
        default: ALU_Ctrl = {`ALU_CTRL_LEN{1'bx}};
    endcase
`SINGLE_SFT0:   ALU_Ctrl = `alu_add;
`SWAB0:         ALU_Ctrl = `alu_add;
`SWAB3:         ALU_Ctrl = `alu_sub_a;
`SWAB4:         ALU_Ctrl = `alu_add;
// Branch
`BRANCH0:       ALU_Ctrl = `alu_add;
// Jump
`JMP0:          ALU_Ctrl = `alu_add;
// JSR
`JSR0:          ALU_Ctrl = `alu_sub_a;
`JSR2:          ALU_Ctrl = `alu_add;
`JSR4:          ALU_Ctrl = `alu_add;
`JSR5:          ALU_Ctrl = `alu_add;
// RTS
`RTS0:          ALU_Ctrl = `alu_add;
`RTS2:          ALU_Ctrl = `alu_add;
// MARK
`MARK0:         ALU_Ctrl = `alu_add;
`MARK2:         ALU_Ctrl = `alu_add;
`MARK4:         ALU_Ctrl = `alu_add;
// TRAP
`TRAP1:         ALU_Ctrl = `alu_add;
`TRAP4:         ALU_Ctrl = `alu_sub_a;
`TRAP5:         ALU_Ctrl = `alu_add;
`TRAP6:         ALU_Ctrl = `alu_add;
`TRAP7:         ALU_Ctrl = `alu_sub_a;
`TRAP10:        ALU_Ctrl = `alu_add;
// Return from Trap (RTT/RTI)
`RTT0:          ALU_Ctrl = `alu_add;
```

```
    `RTT1:          ALU_Ctrl = `alu_add;
    // SOB
    `SOB0:          ALU_Ctrl = `alu_sub_a;
    `SOB2:          ALU_Ctrl = `alu_sub_b;
    // ASH
    `ASH0:          ALU_Ctrl = `alu_add;
    `ASH2:          ALU_Ctrl = DST_15 ? `alu_add : `alu_sub_a;
    `ASH3:          ALU_Ctrl = `alu_add;
    // ASHC
    `ASHC3:         ALU_Ctrl = DST_15 ? `alu_add : `alu_sub_a;
    `ASHC4:         ALU_Ctrl = `alu_add;
    `ASHC5:         ALU_Ctrl = `alu_add;
    // Multiply
    `MUL0:          ALU_Ctrl = `alu_add;
    `MUL1:          ALU_Ctrl = DST_15 ? `alu_sub_b : `alu_add;
    `MUL2:          ALU_Ctrl = SRC_15 ? `alu_sub_b : `alu_add;
    `MUL3:          ALU_Ctrl = `alu_add;
    `MUL4:          ALU_Ctrl = `alu_add;
    `MUL6:          ALU_Ctrl = mult_div_sign ? `alu_sub_b : `alu_add;
    `MUL8:          ALU_Ctrl = `alu_add;
    `MUL9:          ALU_Ctrl = `alu_comp;
    `MUL10:         ALU_Ctrl = `alu_add;
    `MUL12:         ALU_Ctrl = mult_div_sign ? `alu_comp : `alu_add;
    // Divide
    `DIV2:          ALU_Ctrl = `alu_add;
    `DIV3:          ALU_Ctrl = `alu_sub_a;
    `DIV5:          ALU_Ctrl = `alu_sub_b;
    `DIV6:          ALU_Ctrl = `alu_comp;
    `DIV7:          ALU_Ctrl = `alu_add;
    `DIV8:          ALU_Ctrl = `alu_add;
    `DIV9:          ALU_Ctrl = `alu_sub_a;
    `DIV10:         ALU_Ctrl = `alu_sub_a;
    `DIV11:         ALU_Ctrl = `alu_or;
    `DIV12:         ALU_Ctrl = mult_div_sign ? `alu_sub_a : `alu_add;
    `DIV14:         ALU_Ctrl = remainder_sign ? `alu_sub_b : `alu_add;
    // Memory Ops
    `MFPI0:         ALU_Ctrl = `alu_add;
    `MFPI1:         ALU_Ctrl = `alu_sub_a;
    `MFPI2:         ALU_Ctrl = `alu_add;
    `MFPI3:         ALU_Ctrl = `alu_add;
    // Move to
    `MTPI0:         ALU_Ctrl = `alu_add;
    `MTPI1:         ALU_Ctrl = `alu_add;
    `MTPI2:         ALU_Ctrl = `alu_add;
    `MTPI3:         ALU_Ctrl = `alu_add;
    default:        ALU_Ctrl = `alu_add;
endcase // End ALU Control Select Logic

/* SRC Low Select Logic */
case (State)
    `SRC_REG0:      SRC_Lo_Sel = `src_lo_rf;
    `SRC_DREG1:     SRC_Lo_Sel = `src_lo_mem;
    `SRC_INC1:      SRC_Lo_Sel = `src_lo_mem;
    `SRC_DINC2:     SRC_Lo_Sel = `src_lo_mem;
    `SRC_DEC1:      SRC_Lo_Sel = `src_lo_mem;
    `SRC_DDEC2:     SRC_Lo_Sel = `src_lo_mem;
```

```
    `SRC_IDX3:      SRC_Lo_Sel = `src_lo_mem;
    `SRC_DIDX4:     SRC_Lo_Sel = `src_lo_mem;
    `SINGLE_EX0:    case(Instruction[`SINGLE_OP])
                        `ROTR:  SRC_Lo_Sel = `src_lo_wd;
                        `ROTL:  SRC_Lo_Sel = `src_lo_wd;
                        `ASR:   SRC_Lo_Sel = `src_lo_wd;
                        `ASL:   SRC_Lo_Sel = `src_lo_wd;
                       default: SRC_Lo_Sel = `src_lo_same;
                    endcase
    `SWAB0:         SRC_Lo_Sel = `src_lo_wd;
    `SWAB2:         SRC_Lo_Sel = `src_lo_ls;
    `ROTR0:         SRC_Lo_Sel = `src_lo_rs;
    `ROTL0:         SRC_Lo_Sel = `src_lo_ls;
    `ASR0:          SRC_Lo_Sel = `src_lo_rs;
    `ASL0:          SRC_Lo_Sel = `src_lo_ls;
    // ASH
    `ASH1:          SRC_Lo_Sel = DST_15 ? `src_lo_rs : `src_lo_ls;
    // ASHC
    `ASHC0:         SRC_Lo_Sel = `src_lo_rf;
    `ASHC2:         SRC_Lo_Sel = DST_15 ? `src_lo_rs : `src_lo_ls;
    // Multiply
    `MUL2:          SRC_Lo_Sel = `src_lo_wd;
    `MUL5:          SRC_Lo_Sel = `src_lo_ls;
    // Divide
    `DIV1:          SRC_Lo_Sel = `src_lo_rf;
    `DIV5:          SRC_Lo_Sel = `src_lo_wd;
    `DIV9:          SRC_Lo_Sel = `src_lo_ls;
    `DIV10:         SRC_Lo_Sel = `src_lo_same;
    `DIV11:         SRC_Lo_Sel = Acc_Hi_0 ? `src_lo_same : `src_lo_ls;
    // TRAP
    `TRAP2:         SRC_Lo_Sel = `src_lo_wd;
    // Memory Ops
    `MFPI1:         SRC_Lo_Sel = `src_lo_mem;
    // Move to
    `MTPI1:         SRC_Lo_Sel = `src_lo_mem;
    default:        SRC_Lo_Sel = `src_lo_same;
endcase // End SRC Low Select Logic

/* SRC High Select Logic */
case (State)
    `FETCH0:        SRC_Hi_Sel = `src_hi_same; // Place holder
    `DOUBLE_EX0:    case (Instruction[`DOUBLE_OP1])
                        `REG: case (Instruction[`DOUBLE_OP2])
                                `ASHC:  SRC_Hi_Sel = `src_hi_rf;
                                default: SRC_Hi_Sel = `src_hi_same;
                              endcase
                       default: SRC_Hi_Sel = `src_hi_same;
                    endcase
    // ASHC
    `ASHC2:         SRC_Hi_Sel = DST_15 ? `src_hi_rs : `src_hi_ls;
    // Multiply
    `MUL0:          SRC_Hi_Sel = `src_hi_wd;
    `MUL5:          SRC_Hi_Sel = `src_hi_ls;
    // Divide
    `DIV0:          SRC_Hi_Sel = `src_hi_rf;
    `DIV6:          SRC_Hi_Sel = `src_hi_wd;
```

```
    `DIV7:          SRC_Hi_Sel = `src_hi_wd;
    `DIV9:          SRC_Hi_Sel = `src_hi_ls;
    `DIV10:         SRC_Hi_Sel = WD_Neg ? `src_hi_same : `src_hi_wd;
    `DIV11:         SRC_Hi_Sel = Acc_Hi_0 ? `src_hi_same : `src_hi_ls;
    default:        SRC_Hi_Sel = `src_hi_same;
endcase // End SRC High Select Logic

/* DST Select Logic */
case (State)
    // DST Fetch
    `DST_REG0:      DST_Sel = `dst_rf;
    `DST_DREG1:     DST_Sel = `dst_mem;
    `DST_INC1:      DST_Sel = `dst_mem;
    `DST_DINC2:     DST_Sel = `dst_mem;
    `DST_DEC1:      DST_Sel = `dst_mem;
    `DST_DDEC2:     DST_Sel = `dst_mem;
    `DST_IDX1:      DST_Sel = `dst_mem;
    `DST_IDX3:      DST_Sel = `dst_mem;
    `DST_DIDX1:     DST_Sel = `dst_mem;
    `DST_DIDX4:     DST_Sel = `dst_mem;
    // SRC Fetch
    `SRC_IDX1:      DST_Sel = `dst_mem;
    `SRC_DIDX1:     DST_Sel = `dst_mem;
    // ASH
    `DOUBLE_EX0:     case(Instruction[`DOUBLE_OP1])
                        `REG: case (Instruction[`DOUBLE_OP2])
                                `ASH: DST_Sel = `dst_shamt;
                                `ASHC: DST_Sel = `dst_shamt;
                                default: DST_Sel = `dst_same;
                              endcase
                        default: DST_Sel = `dst_same;
                     endcase
    // SWAB
    `SWAB0:         DST_Sel = `dst_shamt;
    `SWAB3:         DST_Sel = `dst_wd;
    //`ASH0:          DST_Sel = `dst_shamt;
    `ASH2:          DST_Sel = `dst_wd;
    // ASHC
    `ASHC3:         DST_Sel = `dst_wd;
    // Multiply
    `MUL1:          DST_Sel = `dst_wd;
    `MUL5:          DST_Sel = `dst_rs;
    // Divide
    `DIV3:          DST_Sel = `dst_wd;
    // TRAP
    `TRAP3:         DST_Sel = `dst_wd;
    default:        DST_Sel = `dst_same;
endcase // End DST Select Logic

/* Accumulator Low Select Logic */
case (State)
    `FETCH1:        ACC_Lo_Sel = `acc_lo_alu_out;
    // DST Fetch
    `DST_INC0:      ACC_Lo_Sel = `acc_lo_alu_out;
    `DST_DINC0:     ACC_Lo_Sel = `acc_lo_alu_out;
    `DST_DEC0:      ACC_Lo_Sel = `acc_lo_alu_out;
```

```
‘DST_DDEC0:     ACC_Lo_Sel = ‘acc_lo_alu_out;
‘DST_IDX0:      ACC_Lo_Sel = ‘acc_lo_alu_out;
‘DST_DIDX0:     ACC_Lo_Sel = ‘acc_lo_alu_out;
// SRC Fetch
‘SRC_INC0:      ACC_Lo_Sel = ‘acc_lo_alu_out;
‘SRC_DINC0:     ACC_Lo_Sel = ‘acc_lo_alu_out;
‘SRC_DEC0:      ACC_Lo_Sel = ‘acc_lo_alu_out;
‘SRC_DDEC0:     ACC_Lo_Sel = ‘acc_lo_alu_out;
‘SRC_IDX0:      ACC_Lo_Sel = ‘acc_lo_alu_out;
‘SRC_DIDX0:     ACC_Lo_Sel = ‘acc_lo_alu_out;
// Single Op Execution
‘SINGLE_EX0:    case (Instruction[‘SINGLE_OP])
                    ‘CLR:   ACC_Lo_Sel = ‘acc_lo_alu_out;
                    ‘COMP:  ACC_Lo_Sel = ‘acc_lo_alu_out;
                    ‘INC:   ACC_Lo_Sel = ‘acc_lo_alu_out;
                    ‘DEC:   ACC_Lo_Sel = ‘acc_lo_alu_out;
                    ‘NEG:   ACC_Lo_Sel = ‘acc_lo_alu_out;
                    ‘ADDC:  ACC_Lo_Sel = ‘acc_lo_alu_out;
                    ‘SUBC:  ACC_Lo_Sel = ‘acc_lo_alu_out;
                    ‘TEST:  ACC_Lo_Sel = ‘acc_lo_alu_out;
                    //‘ROTR:  ACC_Lo_Sel = ‘acc_lo_sft_out;
                    //‘ROTL:  ACC_Lo_Sel = ‘acc_lo_sft_out;
                    //‘ASR:   ACC_Lo_Sel = ‘acc_lo_sft_out;
                    //‘ASL:   ACC_Lo_Sel = ‘acc_lo_sft_out;
                    ‘SXT:   ACC_Lo_Sel = ‘acc_lo_alu_out;
                    default: ACC_Lo_Sel = ‘acc_lo_same;
                endcase
// Double Op Execution
‘DOUBLE_EX0: case (Instruction[‘DOUBLE_OP1])
    ‘MOV:   ACC_Lo_Sel = ‘acc_lo_alu_out;
    ‘CMP:   ACC_Lo_Sel = ‘acc_lo_alu_out;
    ‘BIT:   ACC_Lo_Sel = ‘acc_lo_alu_out;
    ‘BIC:   ACC_Lo_Sel = ‘acc_lo_alu_out;
    ‘BIS:   ACC_Lo_Sel = ‘acc_lo_alu_out;
    ‘ADD:   ACC_Lo_Sel = ‘acc_lo_alu_out;
    ‘REG:   case (Instruction[‘DOUBLE_OP2])
                ‘ASH: ACC_Lo_Sel = ‘acc_lo_sft_out;
                ‘XOR: ACC_Lo_Sel = ‘acc_lo_alu_out;
                default: ACC_Lo_Sel = ‘acc_lo_same;
            endcase
        default: ACC_Lo_Sel = ‘acc_lo_same;
    endcase
‘SINGLE_SFT0:   ACC_Lo_Sel = ‘acc_lo_alu_out;
// Branch Instructions
‘BRANCH0:   ACC_Lo_Sel = ‘acc_lo_alu_out;
// Jump
‘JMP0:      ACC_Lo_Sel = ‘acc_lo_alu_out;
// SWAB
‘SWAB4:     ACC_Lo_Sel = ‘acc_lo_alu_out;
// JSR
‘JSR0:      ACC_Lo_Sel = ‘acc_lo_alu_out;
‘JSR2:      ACC_Lo_Sel = ‘acc_lo_alu_out;
‘JSR4:      ACC_Lo_Sel = ‘acc_lo_alu_out;
‘JSR5:      ACC_Lo_Sel = ‘acc_lo_alu_out;
// RTS
‘RTS0:      ACC_Lo_Sel = ‘acc_lo_alu_out;
```

```
    `RTS2:      ACC_Lo_Sel = `acc_lo_alu_out;
    // MARK
    `MARK0:     ACC_Lo_Sel = `acc_lo_alu_out;
    `MARK2:     ACC_Lo_Sel = `acc_lo_alu_out;
    `MARK4:     ACC_Lo_Sel = `acc_lo_alu_out;
    // TRAP
    `TRAP0:     ACC_Lo_Sel = `acc_lo_trap;
    `TRAP1:     ACC_Lo_Sel = `acc_lo_alu_out;
    `TRAP4:     ACC_Lo_Sel = `acc_lo_alu_out;
    `TRAP7:     ACC_Lo_Sel = `acc_lo_alu_out;
    // Return from Trap (RTT/RTI)
    `RTT0:      ACC_Lo_Sel = `acc_lo_alu_out;
    `RTT1:      ACC_Lo_Sel = `acc_lo_alu_out;
    // SOB
    `SOB0:      ACC_Lo_Sel = `acc_lo_alu_out;
    `SOB2:      ACC_Lo_Sel = `acc_lo_alu_out;
    // ASH
    `ASH3:      ACC_Lo_Sel = `acc_lo_alu_out;
    // ASHC
    //`ASHC2:     ACC_Lo_Sel = SRC_Reg_5 ? `acc_lo_sft_out : `acc_lo_same;
    `ASHC4:     ACC_Lo_Sel = `acc_lo_alu_out;
    `ASHC5:     ACC_Lo_Sel = `acc_lo_alu_out;
    // Multiply
    `MUL0:      ACC_Lo_Sel = `acc_lo_alu_out;
    `MUL3:      ACC_Lo_Sel = DST_0 ? `acc_lo_alu_out : `acc_lo_same;
    `MUL6:      ACC_Lo_Sel = `acc_lo_alu_out;
    `MUL8:      ACC_Lo_Sel = `acc_lo_alu_out;
    `MUL9:      ACC_Lo_Sel = `acc_lo_alu_out;
    `MUL10:     ACC_Lo_Sel = `acc_lo_alu_out;
    // Divide
    `DIV2:      ACC_Lo_Sel = `acc_lo_alu_out;
    `DIV11:     ACC_Lo_Sel = N_Cur ? `acc_lo_same : `acc_lo_alu_out;
    `DIV12:     ACC_Lo_Sel = `acc_lo_alu_out;
    `DIV14:     ACC_Lo_Sel = `acc_lo_alu_out;
    // Memory Ops
    `MFPI1:     ACC_Lo_Sel = `acc_lo_alu_out;
    // Move to
    `MTPI1:     ACC_Lo_Sel = `acc_lo_alu_out;
    default:        ACC_Lo_Sel = `acc_lo_same;
endcase // End Accumulator Low Select Logic

/* Accumulator High Select Logic */
case (State)
    `FETCH0:        ACC_Hi_Sel = `acc_hi_same; // Place holder
    // ASHC
    //`ASHC2:         ACC_Hi_Sel = SRC_Reg_5 ? `acc_hi_same : `acc_hi_sft_out;
    //`ASHC3:         ACC_Hi_Sel = SRC_Reg_5 ? `acc_hi_sft_out : `acc_hi_same;
    // Multiply
    `MUL0:          ACC_Hi_Sel = `acc_hi_alu_out;
    `MUL4:          ACC_Hi_Sel = DST_0 ? `acc_hi_alu_out : `acc_hi_same;
    // Divide
    `DIV9:          ACC_Hi_Sel = `acc_hi_div;
    `DIV11:         ACC_Hi_Sel = `acc_hi_rs;
    default:        ACC_Hi_Sel = `acc_hi_same;
endcase
```

```
/* Shift Amount Select Logic */
case (State)
    `SINGLE_EX0:    case (Instruction[`SINGLE_OP])
                        `ROTR:  Shamt_Sel = `shamt_R1;
                        `ROTL:  Shamt_Sel = `shamt_L1;
                        `ASR:   Shamt_Sel = `shamt_R1;
                        `ASL:   Shamt_Sel = `shamt_L1;
                        default: Shamt_Sel = `shamt_zero;
                    endcase
    `DOUBLE_EX0: case (Instruction[`DOUBLE_OP1])
            `REG:   case (Instruction[`DOUBLE_OP2])
                        `ASH: Shamt_Sel = `shamt_src;
                        `ASHC: Shamt_Sel = `shamt_src;
                    default: Shamt_Sel = `shamt_zero;
                endcase
             default: Shamt_Sel = `shamt_zero;
          endcase
    `SWAB0:         Shamt_Sel = `shamt_8;
    //`ASH1:          Shamt_Sel = `shamt_src;
    `ASHC2:         Shamt_Sel = `shamt_src;
    `ASHC3:         Shamt_Sel = `shamt_src;
    default:        Shamt_Sel = `shamt_zero;
endcase // End Shift Amount Select Logic

/* Rotate Select Logic */
case (State)
    `SINGLE_EX0:    case (Instruction[`SINGLE_OP])
                        `ROTR:  Rotate = 1'b1;
                        `ROTL:  Rotate = 1'b1;
                        default: Rotate = 1'b0;
                    endcase
    `SWAB0:  Rotate = 1'b1;
    `ASHC2:  Rotate = 1'b1;
    `ASHC3:  Rotate = 1'b0;
    default: Rotate = 1'b0;
endcase // End Rotate Select Logic

/* Shift Invert Shamt */
case (State)
    `ASH1:  SFT_Inv_Shamt = 1'b1;
    default: SFT_Inv_Shamt = 1'b0;
endcase // End SFT_Inv_Shamt

/* Shift SRC High select Logic */
case (State)
    `FETCH0:        Shift_SRC_Hi = `shift_hi_lo; // Place holder
    `SINGLE_EX0:    Shift_SRC_Hi = `shift_hi_dst;
    `SWAB0:         Shift_SRC_Hi = `shift_hi_dst;
    `ASH1:          Shift_SRC_Hi = `shift_hi_dst;
    `ASHC2:         Shift_SRC_Hi = Instruction[5] ^ Instruction[4] ? `shift_hi_lo : `shift_hi_hi;
    `ASHC3:         Shift_SRC_Hi = SRC_Reg_5 ? `shift_hi_hi : `shift_hi_lo; // Use High SRC for shift if ne
    default:        Shift_SRC_Hi = `shift_hi_lo;
endcase // End Shift SRC Hi Select Logic

/* Shift SRC Low Select Logic */
case (State)
```

```
    `FETCH0:        Shift_SRC_Lo = `shift_lo_lo;
    `SINGLE_EX0:    Shift_SRC_Lo = `shift_lo_dst;
    `SWAB0:         Shift_SRC_Lo = `shift_lo_dst;
    `ASH1:          Shift_SRC_Lo = `shift_lo_dst;
    `ASHC2:         Shift_SRC_Lo = Instruction[5] ^ Instruction[4] ? `shift_lo_hi : `shift_lo_lo;
    `ASHC3:         Shift_SRC_Lo = SRC_Reg_5 ? `shift_lo_hi : `shift_lo_lo; // Use High SRC for shift if ne
    default:        Shift_SRC_Lo = `shift_lo_lo;
endcase


/* Mem Addr Selection */
case (State)
    `FETCH1:        Mem_Addr_Sel = `mem_addr_alu_b;
    // DST Fetch
    `DST_DREG0:     Mem_Addr_Sel = `mem_addr_alu_b;
    `DST_INC0:      Mem_Addr_Sel = `mem_addr_alu_b;
    `DST_DINC0:     Mem_Addr_Sel = `mem_addr_alu_b;
    `DST_DINC1:     Mem_Addr_Sel = `mem_addr_mem;
    `DST_DEC0:      Mem_Addr_Sel = `mem_addr_alu_out;
    `DST_DDEC0:     Mem_Addr_Sel = `mem_addr_alu_out;
    `DST_DDEC1:     Mem_Addr_Sel = `mem_addr_mem;
    `DST_IDX0:      Mem_Addr_Sel = `mem_addr_alu_b;
    `DST_IDX2:      Mem_Addr_Sel = `mem_addr_alu_out;
    `DST_DIDX0:     Mem_Addr_Sel = `mem_addr_alu_b;
    `DST_DIDX2:     Mem_Addr_Sel = `mem_addr_alu_out;
    `DST_DIDX3:     Mem_Addr_Sel = `mem_addr_mem;
    // SRC Fetch
    `SRC_DREG0:     Mem_Addr_Sel = `mem_addr_alu_b;
    `SRC_INC0:      Mem_Addr_Sel = `mem_addr_alu_b;
    `SRC_DINC0:     Mem_Addr_Sel = `mem_addr_alu_b;
    `SRC_DINC1:     Mem_Addr_Sel = `mem_addr_mem;
    `SRC_DEC0:      Mem_Addr_Sel = `mem_addr_alu_out;
    `SRC_DDEC0:     Mem_Addr_Sel = `mem_addr_alu_out;
    `SRC_DDEC1:     Mem_Addr_Sel = `mem_addr_mem;
    `SRC_IDX0:      Mem_Addr_Sel = `mem_addr_alu_b;
    `SRC_IDX2:      Mem_Addr_Sel = `mem_addr_alu_out;
    `SRC_DIDX0:     Mem_Addr_Sel = `mem_addr_alu_b;
    `SRC_DIDX2:     Mem_Addr_Sel = `mem_addr_alu_out;
    `SRC_DIDX3:     Mem_Addr_Sel = `mem_addr_mem;
    // Write Back
    // Write Back already has correct address
    // left over from DST Fetch, if it's a memory op

    // JSR
    `JSR0:          Mem_Addr_Sel = `mem_addr_alu_out;
    // RTS
    `RTS2:          Mem_Addr_Sel = `mem_addr_alu_b;
    // MARK
    `MARK4:         Mem_Addr_Sel = `mem_addr_alu_b;
    // TRAP
    `TRAP1:         Mem_Addr_Sel = `mem_addr_acc_lo;
    `TRAP2:         Mem_Addr_Sel = `mem_addr_acc_lo;
    `TRAP4:         Mem_Addr_Sel = `mem_addr_alu_out;
    `TRAP7:         Mem_Addr_Sel = `mem_addr_alu_out;
    // Return from TRAP! (RTT/RTI)
    `RTT0:          Mem_Addr_Sel = `mem_addr_alu_b;
```

```
    `RTT1:          Mem_Addr_Sel = `mem_addr_acc_lo;
    // Memory Ops
    `MFPI0:         Mem_Addr_Sel = `mem_addr_alu_out;
    `MFPI1:         Mem_Addr_Sel = `mem_addr_alu_out;
    // Move to
    `MTPI0:         Mem_Addr_Sel = `mem_addr_alu_b;
    `MTPI1:         Mem_Addr_Sel = `mem_addr_alu_out;
    default:        Mem_Addr_Sel = `mem_addr_same;
endcase // End Mem Addr Selection

/* Mem Enable Selection */
case (State)
    `FETCH2:        Mem_Enable = 1'b1;
    // DST Fetch
    `DST_DREG1:     Mem_Enable = 1'b1;
    `DST_INC1:      Mem_Enable = 1'b1;
    `DST_DINC1:     Mem_Enable = 1'b1;
    `DST_DINC2:     Mem_Enable = 1'b1;
    `DST_DEC1:      Mem_Enable = 1'b1;
    `DST_DDEC1:     Mem_Enable = 1'b1;
    `DST_DDEC2:     Mem_Enable = 1'b1;
    `DST_IDX1:      Mem_Enable = 1'b1;
    `DST_IDX3:      Mem_Enable = 1'b1;
    `DST_DIDX1:     Mem_Enable = 1'b1;
    `DST_DIDX3:     Mem_Enable = 1'b1;
    `DST_DIDX4:     Mem_Enable = 1'b1;
    // SRC Fetch
    `SRC_DREG1:     Mem_Enable = 1'b1;
    `SRC_INC1:      Mem_Enable = 1'b1;
    `SRC_DINC1:     Mem_Enable = 1'b1;
    `SRC_DINC2:     Mem_Enable = 1'b1;
    `SRC_DEC1:      Mem_Enable = 1'b1;
    `SRC_DDEC1:     Mem_Enable = 1'b1;
    `SRC_DDEC2:     Mem_Enable = 1'b1;
    `SRC_IDX1:      Mem_Enable = 1'b1;
    `SRC_IDX3:      Mem_Enable = 1'b1;
    `SRC_DIDX1:     Mem_Enable = 1'b1;
    `SRC_DIDX3:     Mem_Enable = 1'b1;
    `SRC_DIDX4:     Mem_Enable = 1'b1;
    // JSR
    `JSR3:          Mem_Enable = 1'b1;
    `JSR4:          Mem_Enable = 1'b1;
    // RTS
    `RTS4:          Mem_Enable = 1'b1;
    // MARK
    `MARK6:         Mem_Enable = 1'b1;
    // TRAP
    `TRAP2:         Mem_Enable = 1'b1;
    `TRAP3:         Mem_Enable = 1'b1;
    `TRAP5:         Mem_Enable = 1'b1;
    `TRAP6:         Mem_Enable = 1'b1;
    `TRAP8:         Mem_Enable = 1'b1;
    `TRAP9:         Mem_Enable = 1'b1;
    // Return From Trap
    `RTT1:          Mem_Enable = 1'b1;
    `RTT2:          Mem_Enable = 1'b1;
```

```
    // Memory Ops
    `MFPI1:         Mem_Enable = 1'b1;
    `MFPI2:         Mem_Enable = 1'b1;
    `MFPI3:         Mem_Enable = 1'b1;
    // Move to
    `MTPI1:         Mem_Enable = 1'b1;
    `MTPI2:         Mem_Enable = 1'b1;
    `MTPI3:         Mem_Enable = 1'b1;
    // Write Back
    `WRITE_BACK0:   Mem_Enable = 1'b1;
    `WRITE_BACK1:   Mem_Enable = 1'b1;
    default:        Mem_Enable = 1'b0;
endcase // End Mem Enable

/* Mem Write Enable Logic */
case (State)
    // JSR
    `JSR3:          Mem_Write_En = 1'b1;
    `JSR4:          Mem_Write_En = 1'b1;
    // TRAP
    `TRAP5:         Mem_Write_En = 1'b1;
    `TRAP6:         Mem_Write_En = 1'b1;
    `TRAP8:         Mem_Write_En = 1'b1;
    `TRAP9:         Mem_Write_En = 1'b1;
    // Memory Ops
    `MFPI2:         Mem_Write_En = 1'b1;
    `MFPI3:         Mem_Write_En = 1'b1;
    // Move to
    `MTPI2:         Mem_Write_En = 1'b1;
    `MTPI3:         Mem_Write_En = 1'b1;
    // Write Back
    `WRITE_BACK0:   Mem_Write_En = Mem_DST;
    `WRITE_BACK1:   Mem_Write_En = Mem_DST;
    default:        Mem_Write_En = 1'b0;
endcase // End Mem Write Enable Logic

/* Reg File Address Selection */
if (reset) begin
    RF_Addr_Sel = `rf_addr_pc;
end
else begin
    case (State)
        `FETCH1:        RF_Addr_Sel = `rf_addr_pc;
        `FETCH2:        RF_Addr_Sel = `rf_addr_pc;
        // DST Fetch
        `DST_REG0:      RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_DREG0:     RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_INC0:      RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_INC1:      RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_DINC0:     RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_DINC1:     RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_DEC0:      RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_DEC1:      RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_DDEC0:     RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_DDEC1:     RF_Addr_Sel = `rf_addr_dst_reg;
        `DST_IDX0:      RF_Addr_Sel = `rf_addr_pc;
```

```
‘DST_IDX1:      RF_Addr_Sel = ‘rf_addr_pc;
‘DST_IDX2:      RF_Addr_Sel = ‘rf_addr_dst_reg;
‘DST_DIDX0:     RF_Addr_Sel = ‘rf_addr_pc;
‘DST_DIDX1:     RF_Addr_Sel = ‘rf_addr_pc;
‘DST_DIDX2:     RF_Addr_Sel = ‘rf_addr_dst_reg;
// SRC Fetch
‘SRC_REG0:      RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_DREG0:     RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_INC0:      RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_INC1:      RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_DINC0:     RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_DINC1:     RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_DEC0:      RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_DEC1:      RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_DDEC0:     RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_DDEC1:     RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_IDX0:      RF_Addr_Sel = ‘rf_addr_pc;
‘SRC_IDX1:      RF_Addr_Sel = ‘rf_addr_pc;
‘SRC_IDX2:      RF_Addr_Sel = ‘rf_addr_src_reg;
‘SRC_DIDX0:     RF_Addr_Sel = ‘rf_addr_pc;
‘SRC_DIDX1:     RF_Addr_Sel = ‘rf_addr_pc;
‘SRC_DIDX2:     RF_Addr_Sel = ‘rf_addr_src_reg;
// For ASHC
‘DOUBLE_EX0:    case (Instruction[‘DOUBLE_OP1])
                    ‘REG: case (Instruction[‘DOUBLE_OP2])
                            ‘ASHC:  RF_Addr_Sel = ‘rf_addr_src_reg;
                            default: RF_Addr_Sel = ‘rf_addr_pc;
                          endcase
                    default: RF_Addr_Sel = ‘rf_addr_pc;
                endcase
// Branch
‘BRANCH_EX0:    RF_Addr_Sel = ‘rf_addr_pc;
// JMP
‘JMP1:          RF_Addr_Sel = ‘rf_addr_pc;
// JSR
‘JSR0:          RF_Addr_Sel = ‘rf_addr_sp;
‘JSR1:          RF_Addr_Sel = ‘rf_addr_sp;
‘JSR2:          RF_Addr_Sel = ‘rf_addr_src_reg;
‘JSR4:          RF_Addr_Sel = ‘rf_addr_pc;
‘JSR5:          RF_Addr_Sel = ‘rf_addr_src_reg;
‘JSR6:          RF_Addr_Sel = ‘rf_addr_pc;
// RTS
‘RTS0:          RF_Addr_Sel = ‘rf_addr_dst_reg;
‘RTS1:          RF_Addr_Sel = ‘rf_addr_pc;
‘RTS2:          RF_Addr_Sel = ‘rf_addr_sp;
‘RTS3:          RF_Addr_Sel = ‘rf_addr_sp;
‘RTS4:          RF_Addr_Sel = ‘rf_addr_dst_reg;
// MARK
‘MARK0:         RF_Addr_Sel = ‘rf_addr_pc;
‘MARK1:         RF_Addr_Sel = ‘rf_addr_sp;
‘MARK2:         RF_Addr_Sel = ‘rf_addr_r5;
‘MARK3:         RF_Addr_Sel = ‘rf_addr_pc;
‘MARK4:         RF_Addr_Sel = ‘rf_addr_sp;
‘MARK5:         RF_Addr_Sel = ‘rf_addr_sp;
‘MARK6:         RF_Addr_Sel = ‘rf_addr_r5;
// TRAP
```

```
        `TRAP4:         RF_Addr_Sel = `rf_addr_sp;
        `TRAP8:         RF_Addr_Sel = `rf_addr_pc;
        `TRAP9:         RF_Addr_Sel = `rf_addr_pc;
        `TRAP10:        RF_Addr_Sel = `rf_addr_pc;
        `TRAP11:        RF_Addr_Sel = `rf_addr_sp;
        // RTT
        `RTT0:          RF_Addr_Sel = `rf_addr_sp;
        `RTT1:          RF_Addr_Sel = `rf_addr_pc;
        `RTT2:          RF_Addr_Sel = `rf_addr_sp;
        // SOB
        `SOB0:          RF_Addr_Sel = `rf_addr_src_reg;
        `SOB1:          RF_Addr_Sel = `rf_addr_src_reg;
        `SOB2:          RF_Addr_Sel = `rf_addr_pc;
        `SOB3:          RF_Addr_Sel = `rf_addr_pc;
        //ASHC
        `ASHC0:         RF_Addr_Sel = `rf_addr_rv1;
        `ASHC5:         RF_Addr_Sel = `rf_addr_rv1;
        `ASHC6:         RF_Addr_Sel = `rf_addr_src_reg;
        // Multiply
        `MUL7:          RF_Addr_Sel = `rf_addr_rv1;
        `MUL11:         RF_Addr_Sel = `rf_addr_src_reg;
        // Divide
        `DIV0:          RF_Addr_Sel = `rf_addr_src_reg;
        `DIV1:          RF_Addr_Sel = `rf_addr_rv1;
        `DIV13:         RF_Addr_Sel = `rf_addr_src_reg;
        `DIV15:         RF_Addr_Sel = `rf_addr_rv1;
        // Memory Ops
        `MFPI1:         RF_Addr_Sel = `rf_addr_sp;
        `MFPI4:         RF_Addr_Sel = `rf_addr_sp;
        // Move to
        `MTPI0:         RF_Addr_Sel = `rf_addr_sp;
        `MTPI4:         RF_Addr_Sel = `rf_addr_sp;
        // Other
        `REG_WRITE_BACK0:   RF_Addr_Sel = `rf_addr_src_reg;
        `WRITE_BACK0:   RF_Addr_Sel = `rf_addr_dst_reg;
        default:        RF_Addr_Sel = `rf_addr_pc;
    endcase // End Reg File Address Selection
end

/* Reg File Enable Selection */
if (reset) begin
    RF_Enable = 1'b1;
end
else begin
    case (State)
        `FETCH1:        RF_Enable = 1'b1;
        `FETCH2:        RF_Enable = 1'b1;
        // DST Fetch
        `DST_REG0:      RF_Enable = 1'b1;
        `DST_DREG0:     RF_Enable = 1'b1;
        `DST_INC0:      RF_Enable = 1'b1;
        `DST_INC1:      RF_Enable = 1'b1;
        `DST_DINC0:     RF_Enable = 1'b1;
        `DST_DINC1:     RF_Enable = 1'b1;
        `DST_DEC0:      RF_Enable = 1'b1;
        `DST_DEC1:      RF_Enable = 1'b1;
```

```
‘DST_DDEC0:     RF_Enable = 1’b1;
‘DST_DDEC1:     RF_Enable = 1’b1;
‘DST_IDX0:      RF_Enable = 1’b1;
‘DST_IDX1:      RF_Enable = 1’b1;
‘DST_IDX2:      RF_Enable = 1’b1;
‘DST_DIDX0:     RF_Enable = 1’b1;
‘DST_DIDX1:     RF_Enable = 1’b1;
‘DST_DIDX2:     RF_Enable = 1’b1;
// SRC  Fetch
‘SRC_REG0:      RF_Enable = 1’b1;
‘SRC_DREG0:     RF_Enable = 1’b1;
‘SRC_INC0:      RF_Enable = 1’b1;
‘SRC_INC1:      RF_Enable = 1’b1;
‘SRC_DINC0:     RF_Enable = 1’b1;
‘SRC_DINC1:     RF_Enable = 1’b1;
‘SRC_DEC0:      RF_Enable = 1’b1;
‘SRC_DEC1:      RF_Enable = 1’b1;
‘SRC_DDEC0:     RF_Enable = 1’b1;
‘SRC_DDEC1:     RF_Enable = 1’b1;
‘SRC_IDX0:      RF_Enable = 1’b1;
‘SRC_IDX1:      RF_Enable = 1’b1;
‘SRC_IDX2:      RF_Enable = 1’b1;
‘SRC_DIDX0:     RF_Enable = 1’b1;
‘SRC_DIDX1:     RF_Enable = 1’b1;
‘SRC_DIDX2:     RF_Enable = 1’b1;
‘DOUBLE_EX0:    case (Instruction[‘DOUBLE_OP1])
                    ‘REG: case (Instruction[‘DOUBLE_OP2])
                            ‘ASHC:  RF_Enable = 1’b1;
                            default: RF_Enable = 1’b0;
                          endcase
                    default: RF_Enable = 1’b0;
                endcase
// Branch
‘BRANCH0:       RF_Enable = 1’b1;
‘BRANCH_EX0:    RF_Enable = 1’b1;
// Jump
‘JMP1:          RF_Enable = 1’b1;
// JSR
‘JSR0:          RF_Enable = 1’b1;
‘JSR1:          RF_Enable = 1’b1;
‘JSR2:          RF_Enable = 1’b1;
‘JSR4:          RF_Enable = 1’b1;
‘JSR5:          RF_Enable = 1’b1;
‘JSR6:          RF_Enable = 1’b1;
// RTS
‘RTS0:          RF_Enable = 1’b1;
‘RTS1:          RF_Enable = 1’b1;
‘RTS2:          RF_Enable = 1’b1;
‘RTS3:          RF_Enable = 1’b1;
‘RTS4:          RF_Enable = 1’b1;
// MARK
‘MARK0:         RF_Enable = 1’b1;
‘MARK1:         RF_Enable = 1’b1;
‘MARK2:         RF_Enable = 1’b1;
‘MARK3:         RF_Enable = 1’b1;
‘MARK4:         RF_Enable = 1’b1;
```

```
        `MARK5:         RF_Enable = 1'b1;
        `MARK6:         RF_Enable = 1'b1;
        // TRAP
        `TRAP4:         RF_Enable = 1'b1;
        `TRAP8:         RF_Enable = 1'b1;
        `TRAP9:         RF_Enable = 1'b1;
        `TRAP10:        RF_Enable = 1'b1;
        `TRAP11:        RF_Enable = 1'b1;
        // RTT
        `RTT0:          RF_Enable = 1'b1;
        `RTT1:          RF_Enable = 1'b1;
        `RTT2:          RF_Enable = 1'b1;
        // SOB
        `SOB0:          RF_Enable = 1'b1;
        `SOB1:          RF_Enable = 1'b1;
        `SOB2:          RF_Enable = 1'b1;
        `SOB3:          RF_Enable = 1'b1;
        // ASHC
        `ASHC0:         RF_Enable = 1'b1;
        `ASHC5:         RF_Enable = 1'b1;
        `ASHC6:         RF_Enable = 1'b1;
        // Multiply
        `MUL7:          RF_Enable = 1'b1;
        `MUL11:         RF_Enable = 1'b1;
        // Divide
        `DIV0:          RF_Enable = 1'b1;
        `DIV1:          RF_Enable = 1'b1;
        `DIV13:         RF_Enable = 1'b1;
        `DIV15:         RF_Enable = 1'b1;
        // Memory Ops
        `MFPI1:         RF_Enable = 1'b1;
        `MFPI4:         RF_Enable = 1'b1;
        // Move to
        `MTPI0:         RF_Enable = 1'b1;
        `MTPI4:         RF_Enable = 1'b1;
        // Others
        `REG_WRITE_BACK0:   RF_Enable = 1'b1;
        `WRITE_BACK0:    RF_Enable = 1'b1;
        default:        RF_Enable = 1'b0;
    endcase // End Register File Enable Selection
end // not reset


/* Reg File Write Enable */
if (reset) begin
    RF_Write_En = 1'b1;
end
else begin
    case (State)
        `FETCH2:        RF_Write_En = 1'b1;
        // DST Fetch
        `DST_INC1:      RF_Write_En = 1'b1;
        `DST_DINC1:     RF_Write_En = 1'b1;
        `DST_DEC1:      RF_Write_En = 1'b1;
        `DST_DDEC1:     RF_Write_En = 1'b1;
        `DST_IDX1:      RF_Write_En = 1'b1;
```

```
`DST_DIDX1:     RF_Write_En = 1'b1;
// SRC Fetch
`SRC_INC1:      RF_Write_En = 1'b1;
`SRC_DINC1:     RF_Write_En = 1'b1;
`SRC_DEC1:      RF_Write_En = 1'b1;
`SRC_DDEC1:     RF_Write_En = 1'b1;
`SRC_IDX1:      RF_Write_En = 1'b1;
`SRC_DIDX1:     RF_Write_En = 1'b1;
// Branches
`BRANCH_EX0:    if (~Instruction[15]) begin
                    case (Instruction[`BRANCH_OP])
                        `SYSB:  RF_Write_En = 1'b0;
                        `BR:    RF_Write_En = 1'b1;
                        `BNE:   RF_Write_En = ~Z_Cur;
                        `BEQ:   RF_Write_En = Z_Cur;
                        `BGE:   RF_Write_En = ~(N_Cur ^ V_Cur);
                        `BLT:   RF_Write_En = N_Cur ^ V_Cur;
                        `BGT:   RF_Write_En = ~Z_Cur & ~(N_Cur ^ V_Cur);
                        `BLE:   RF_Write_En = Z_Cur | (N_Cur ^ V_Cur);
                    endcase
                end
                else begin
                    case (Instruction[`BRANCH_OP])
                        `BPL:   RF_Write_En = ~N_Cur;
                        `BMI:   RF_Write_En = N_Cur;
                        `BHI:   RF_Write_En = ~C_Cur & ~Z_Cur;
                        `BLOS:  RF_Write_En = C_Cur | Z_Cur;
                        `BVC:   RF_Write_En = ~V_Cur;
                        `BVS:   RF_Write_En = V_Cur;
                        `BCC:   RF_Write_En = ~C_Cur;
                        `BCS:   RF_Write_En = C_Cur;
                    endcase
                end
// Jump
`JMP1:          RF_Write_En = 1'b1;
// JSR
`JSR1:          RF_Write_En = 1'b1;
`JSR5:          RF_Write_En = 1'b1;
`JSR6:          RF_Write_En = 1'b1;
// RTS
`RTS1:          RF_Write_En = 1'b1;
`RTS3:          RF_Write_En = 1'b1;
`RTS4:          RF_Write_En = 1'b1;
// MARK
`MARK1:         RF_Write_En = 1'b1;
`MARK3:         RF_Write_En = 1'b1;
`MARK5:         RF_Write_En = 1'b1;
`MARK6:         RF_Write_En = 1'b1;
// TRAP
`TRAP10:         RF_Write_En = 1'b1;
`TRAP11:         RF_Write_En = 1'b1;
// RTT
`RTT1:          RF_Write_En = 1'b1;
`RTT2:          RF_Write_En = 1'b1;
// SOB
`SOB1:          RF_Write_En = 1'b1;
```

```
        `SOB3:          RF_Write_En = 1'b1;
        // ASHC
        `ASHC5:         RF_Write_En = 1'b1;
        `ASHC6:         RF_Write_En = 1'b1;
        // Multiply
        `MUL7:          RF_Write_En = 1'b1;
        `MUL11:         RF_Write_En = ~SRC_Reg_0;
        // Divide
        `DIV13:         RF_Write_En = 1'b1;
        `DIV15:         RF_Write_En = 1'b1;
        // Memory Ops
        `MFPI4:         RF_Write_En = 1'b1;
        // Move to
        `MTPI4:         RF_Write_En = 1'b1;
        // Other
        `REG_WRITE_BACK0:   RF_Write_En = 1'b1;
        `WRITE_BACK0:   RF_Write_En = ~Mem_DST;
        default:        RF_Write_En = 1'b0;
    endcase // End Reg File Write Enable
end // not reset


/* Write Back Data Select Logic */
if (reset) begin
    WD_Sel = `wd_mem;
end
else begin
    case (State)
        `FETCH2:        WD_Sel = `wd_acc_lo;
        // DST Fetch
        `DST_INC1:      WD_Sel = `wd_acc_lo;
        `DST_DINC1:     WD_Sel = `wd_acc_lo;
        `DST_DEC1:      WD_Sel = `wd_acc_lo;
        `DST_DDEC1:     WD_Sel = `wd_acc_lo;
        `DST_IDX1:      WD_Sel = `wd_acc_lo;
        `DST_DIDX1:     WD_Sel = `wd_acc_lo;
        // SRC Fetch
        `SRC_INC1:      WD_Sel = `wd_acc_lo;
        `SRC_DINC1:     WD_Sel = `wd_acc_lo;
        `SRC_DEC1:      WD_Sel = `wd_acc_lo;
        `SRC_DDEC1:     WD_Sel = `wd_acc_lo;
        `SRC_IDX1:      WD_Sel = `wd_acc_lo;
        `SRC_DIDX1:     WD_Sel = `wd_acc_lo;
        // Double Instrs
        `DOUBLE_EX0:    WD_Sel = `wd_alu_out;
        `SINGLE_EX0:    WD_Sel = `wd_alu_out;
        // SWAB
        `SWAB0:         WD_Sel = `wd_alu_out;
        `SWAB3:         WD_Sel = `wd_alu_out;
        // Branches
        `BRANCH_EX0:    WD_Sel = `wd_acc_lo;
        // Jump
        `JMP1:          WD_Sel = `wd_acc_lo;
        // JSR
        `JSR1:          WD_Sel = `wd_acc_lo;
        `JSR3:          WD_Sel = `wd_acc_lo;
```

```
`JSR4:          WD_Sel = `wd_acc_lo;
`JSR5:          WD_Sel = `wd_acc_lo;
`JSR6:          WD_Sel = `wd_acc_lo;
// RTS
`RTS1:          WD_Sel = `wd_acc_lo;
`RTS3:          WD_Sel = `wd_acc_lo;
`RTS4:          WD_Sel = `wd_mem;
// SOB
`SOB1:          WD_Sel = `wd_acc_lo;
`SOB3:          WD_Sel = `wd_acc_lo;
// MARK
`MARK1:         WD_Sel = `wd_acc_lo;
`MARK3:         WD_Sel = `wd_acc_lo;
`MARK5:         WD_Sel = `wd_acc_lo;
`MARK6:         WD_Sel = `wd_mem;
// TRAP
`TRAP2:         WD_Sel = `wd_mem;
`TRAP3:         WD_Sel = `wd_psw;
`TRAP5:         WD_Sel = `wd_alu_out;
`TRAP6:         WD_Sel = `wd_alu_out;
`TRAP8:         WD_Sel = `wd_alu_out;
`TRAP9:         WD_Sel = `wd_alu_out;
`TRAP10:        WD_Sel = `wd_alu_out;
`TRAP11:        WD_Sel = `wd_acc_lo;
// RTT
`RTT1:          WD_Sel = `wd_mem;
`RTT2:          WD_Sel = `wd_acc_lo;
// ASH
//`ASH0:          WD_Sel = `wd_alu_out;
`ASH2:          WD_Sel = `wd_alu_out;
// ASHC
`ASHC3:         WD_Sel = `wd_alu_out;
`ASHC5:         WD_Sel = `wd_acc_lo;
`ASHC6:         WD_Sel = `wd_acc_lo;
// Multiply
`MUL3:          WD_Sel = `wd_alu_out;
`MUL6:          WD_Sel = `wd_alu_out;
`MUL7:          WD_Sel = `wd_acc_lo;
`MUL11:         WD_Sel = `wd_acc_lo;
`MUL12:         WD_Sel = `wd_alu_out;
// Divide
`DIV3:          WD_Sel = `wd_alu_out;
`DIV5:          WD_Sel = `wd_alu_out;
`DIV6:          WD_Sel = `wd_alu_out;
`DIV7:          WD_Sel = `wd_alu_out;
`DIV8:          WD_Sel = `wd_alu_out;
`DIV9:          WD_Sel = `wd_alu_out;
`DIV10:         WD_Sel = `wd_alu_out;
`DIV13:         WD_Sel = `wd_acc_lo;
`DIV15:         WD_Sel = `wd_acc_lo;
// Memory Ops
`MFPI1:         WD_Sel = `wd_mem;
`MFPI2:         WD_Sel = `wd_alu_out;
`MFPI3:         WD_Sel = `wd_alu_out;
`MFPI4:         WD_Sel = `wd_acc_lo;
// Move to
```

```
        `MTPI1:         WD_Sel = `wd_mem;
        `MTPI2:         WD_Sel = `wd_alu_out;
        `MTPI3:         WD_Sel = `wd_alu_out;
        `MTPI4:         WD_Sel = `wd_acc_lo;
        // Write Back
        `REG_WRITE_BACK0:   WD_Sel = `wd_acc_lo;
        `WRITE_BACK0:   WD_Sel = MOVB ? `wd_movb : `wd_acc_lo;
        `WRITE_BACK1:   WD_Sel = `wd_acc_lo;
        default: WD_Sel = `wd_alu_out;
    endcase // End Write Back Data Select Logic
end //not reset

/* Read Byte Select Logic */
case (State)
    // DST Fetch
    `DST_REG0:      Read_Byte = Byte;
    `DST_DREG1:     Read_Byte = Byte;
    `DST_INC1:      Read_Byte = Byte;
    `DST_DINC2:     Read_Byte = Byte;
    `DST_DEC1:      Read_Byte = Byte;
    `DST_DDEC2:     Read_Byte = Byte;
    `DST_IDX3:      Read_Byte = Byte;
    `DST_DIDX4:     Read_Byte = Byte;
    // SRC Fetch
    `SRC_REG0:      Read_Byte = Byte;
    `SRC_DREG1:     Read_Byte = Byte;
    `SRC_INC1:      Read_Byte = Byte;
    `SRC_DINC2:     Read_Byte = Byte;
    `SRC_DEC1:      Read_Byte = Byte;
    `SRC_DDEC2:     Read_Byte = Byte;
    `SRC_IDX3:      Read_Byte = Byte;
    `SRC_DIDX4:     Read_Byte = Byte;
    //
    `WRITE_BACK0:   Read_Byte = Byte & ~MOVB;
    default:        Read_Byte = 1'b0;
endcase

/* Use Cin Select Logic */
case (State)
    `FETCH0:        Use_Cin = 1'b0;     // Place holder
    `SINGLE_EX0:    case (Instruction[`SINGLE_OP])
                        `ROTR:   Use_Cin = 1'b0;
                        `ROTL:   Use_Cin = 1'b0;
                        default: Use_Cin = 1'b0;
                    endcase
    `MUL4:          Use_Cin = 1'b1;
    default:        Use_Cin = 1'b0;
endcase // End Cin Select Logic

/* Lo_Hi_Bit */
case (State)
    `ROTR0:  Lo_Hi_Bit = C_Cur;
    `ASR0:   Lo_Hi_Bit = SRC_15;
    `ASH1:   Lo_Hi_Bit = SRC_15;
    `ASHC2:  Lo_Hi_Bit = SRC_16;
    default: Lo_Hi_Bit = 1'b0;
```

```
endcase

/* Lo_Lo_Bit */
case (State)
    `ROTL0:  Lo_Lo_Bit = C_Cur;
    `ASL0:   Lo_Lo_Bit = 1'b0;
    `SWAB2:  Lo_Lo_Bit = SRC_15 ? 1'b1 : 1'b0;
    default: Lo_Lo_Bit = 1'b0;
endcase

/* Lo_Mid_Bit */
case (State)
    `ROTR0: Lo_Mid_Bit = Byte ? C_Cur : SRC_8;
    `ASR0:  Lo_Mid_Bit = Byte ? DST_7 : SRC_8;
    default: Lo_Mid_Bit = SRC_8;
endcase

/* Write Data Byte Select Logic */
case (State)
    //JSR
    `JSR3:          WD_Byte_Sel = 1'b0;
    `JSR4:          WD_Byte_Sel = 1'b1;
    // TRAP
    `TRAP5:         WD_Byte_Sel = 1'b0;
    `TRAP6:         WD_Byte_Sel = 1'b1;
    `TRAP8:         WD_Byte_Sel = 1'b0;
    `TRAP9:         WD_Byte_Sel = 1'b1;
    // Memory Ops
    `MFPI2:         WD_Byte_Sel = 1'b0;
    `MFPI3:         WD_Byte_Sel = 1'b1;
    // Move to
    `MTPI2:         WD_Byte_Sel = 1'b0;
    `MTPI3:         WD_Byte_Sel = 1'b1;
    // Write Back
    `WRITE_BACK0:   WD_Byte_Sel = 1'b0;
    `WRITE_BACK1:   WD_Byte_Sel = 1'b1;
    default:        WD_Byte_Sel = 1'b0;
endcase // End Write Data Byte Select

/* Memory Address Out Base Select Logic */
case (State)
    // JSR
    `JSR3:          Mem_Addr_Out_Base = 1'b1;
    `JSR4:          Mem_Addr_Out_Base = 1'b0;
    // TRAP
    `TRAP5:         Mem_Addr_Out_Base = 1'b1;
    `TRAP6:         Mem_Addr_Out_Base = 1'b0;
    `TRAP8:         Mem_Addr_Out_Base = 1'b1;
    `TRAP9:         Mem_Addr_Out_Base = 1'b0;
    // Memory Ops
    `MFPI2:         Mem_Addr_Out_Base = 1'b1;
    `MFPI3:         Mem_Addr_Out_Base = 1'b0;
    // Move to
    `MTPI2:         Mem_Addr_Out_Base = 1'b1;
    `MTPI3:         Mem_Addr_Out_Base = 1'b0;
    // Write Back
```

```
    `WRITE_BACK0:   Mem_Addr_Out_Base = 1'b1;
    `WRITE_BACK1:   Mem_Addr_Out_Base = 1'b0;
    default:        Mem_Addr_Out_Base = 1'b1;
endcase // End Memory Address out Base Select Logic

/* C Select Logic */
// C must be selected during Execute stages
case (State)
    `FETCH0:        C_Sel = `c_same;
    //`WRITE_BACK0:   case (Instr_type)
    `SINGLE_EX0: case (Instruction[`SINGLE_OP])
                    `CLR:   C_Sel = `c_clear;
                    `COMP:  C_Sel = `c_set;
                    `INC:   C_Sel = `c_same;
                    `DEC:   C_Sel = `c_same;
                    `NEG:   C_Sel = `c_not_z;
                    `ADDC:  C_Sel = `c_alu;
                    `SUBC:  C_Sel = DST_Zero & C_Cur ? `c_set : `c_clear;
                    `TEST:  C_Sel = `c_clear;
                    //`ROTR:  C_Sel = `c_sft;
                    //`ROTL:  C_Sel = `c_sft;
                    //`ASR:   C_Sel = `c_sft;
                    //`ASL:   C_Sel = `c_sft;
                    `SXT:   C_Sel = `c_same;
                    default: C_Sel = `c_same;
                endcase
   `DOUBLE_EX0: case (Instruction[`DOUBLE_OP1])
                    `MOV:   C_Sel = `c_same;
                    `CMP:   C_Sel = `c_alu_not;
                    `BIT:   C_Sel = `c_same;
                    `BIC:   C_Sel = `c_same;
                    `BIS:   C_Sel = `c_same;
                    `ADD:   C_Sel = Instruction[15] ? `c_alu_not : `c_alu;
                    `REG:   case (Instruction[`DOUBLE_OP2])
                                `ASH:   C_Sel = `c_clear;
                                `ASHC:  C_Sel = `c_clear;
                                `XOR:   C_Sel = `c_same;
                                default: C_Sel = `c_same;
                            endcase
                    default: C_Sel = `c_same;
                endcase
    `ROTR0:     C_Sel = SRC_0 ? `c_set : `c_clear;
    `ROTL0:     C_Sel = Byte ? (DST_7 ? `c_set : `c_clear) : (SRC_15 ? `c_set : `c_clear);
    `ASR0:      C_Sel = SRC_0 ? `c_set : `c_clear;
    `ASL0:      C_Sel = Byte ? (DST_7 ? `c_set : `c_clear) : (SRC_15 ? `c_set : `c_clear);
    `SCC0:      C_Sel = Instruction[0] ? (Instruction[4] ? `c_set : `c_clear) : `c_same;
    // SWAB
    `SWAB0:     C_Sel = `c_clear;
    `ASH1:      C_Sel = DST_15 ? (SRC_0 ? `c_set : `c_clear) : (SRC_15 ? `c_set : `c_clear);
    // ASHC
    `ASHC2:     C_Sel = DST_15 ? (SRC_0 ? `c_set : `c_clear) : (SRC_31 ? `c_set : `c_clear);
    //`ASHC2:     C_Sel = `c_sft;
    // Multiply
    `MUL3:      C_Sel = `c_alu;
    `MUL6:      C_Sel = `c_alu;
    `MUL12:     C_Sel = Z_Cur ? `c_clear : (WD_Zero ? `c_clear : `c_set);
```

```
    // Divide
    `DIV5:      C_Sel = `c_alu;
    `DIV8:      C_Sel = DST_Zero ? `c_set : `c_clear;
    `DIV9:      C_Sel = `c_clear;
    default:    C_Sel = `c_same;
endcase // End C Select Logic

/* V Select Logic */
case (State)
    `FETCH0:        V_Sel = `v_same;
    `DOUBLE_EX0:    case (Instruction[`DOUBLE_OP1])
                        `BIT: V_Sel = `v_clear;
                        `CMP: V_Sel = `v_double_cmp;
                        `REG:   case (Instruction[`DOUBLE_OP2])
                                    `ASH:   V_Sel = `v_clear;
                                    `ASHC:  V_Sel = `v_clear;
                                    default: V_Sel = `v_same;
                                endcase
                        default: V_Sel = `v_same;
                    endcase
    `WRITE_BACK0:   case (Instr_type)
                        `INSTR_SINGLE: case (Instruction[`SINGLE_OP])
                                           `CLR:   V_Sel = `v_clear;
                                           `COMP:  V_Sel = `v_clear;
                                           `INC:   V_Sel = ~DST_Neg & WD_Neg ? `v_set : `v_clear;
                                           `DEC:   V_Sel = DST_Neg & ~WD_Neg ? `v_set : `v_clear;
                                           `NEG:   V_Sel = `v_neg;
                                           `ADDC:  V_Sel = `v_single_pos;
                                           `SUBC:  V_Sel = `v_single_neg;
                                           `TEST:  V_Sel = `v_clear;
                                           `ROTR:  V_Sel = `v_sft;
                                           `ROTL:  V_Sel = `v_sft;
                                           `ASR:   V_Sel = `v_sft;
                                           `ASL:   V_Sel = `v_sft;
                                           `SXT:   V_Sel = `v_clear;
                                           default: V_Sel = `v_same;
                                       endcase
                        `INSTR_DOUBLE:  case (Instruction[`DOUBLE_OP1])
                                           `MOV:   V_Sel = `v_clear;
                                           `CMP:   V_Sel = `v_double_cmp;
                                           `BIT:   V_Sel = `v_clear;
                                           `BIC:   V_Sel = `v_clear;
                                           `BIS:   V_Sel = `v_clear;
                                           `ADD:   V_Sel = Instruction[15] ? `v_double_sub : `v_double_add
                                           `REG:   case (Instruction[`DOUBLE_OP2])
                                                       `ASH:   V_Sel = `v_clear;
                                                       `XOR:   V_Sel = `v_clear;
                                                       default: V_Sel = `v_same;
                                                   endcase
                                           default: V_Sel = `v_same;
                                       endcase
                            default:    V_Sel = `v_same;
                        endcase
    `REG_WRITE_BACK0:   case (Instr_type)
                            `INSTR_DOUBLE: case (Instruction[`DOUBLE_OP1])
                                               `REG: case (Instruction[`DOUBLE_OP2])
```

```
                                                    //`ASH: V_Sel = WD_Neg ^ SRC_15 ? `v_set : `v_clear;
                                                    //`ASH: V_Sel = `v_clear;
                                                    `XOR: V_Sel = `v_clear;
                                                    default: V_Sel = `v_same;
                                                endcase
                                            default: V_Sel = `v_same;
                                        endcase
                                default:    V_Sel = `v_same;
                            endcase
        `SCC0:      V_Sel = Instruction[1] ? (Instruction[4] ? `v_set : `v_clear) : `v_same;
        // SWAB
        `SWAB0:     V_Sel = `v_clear;
        // ASH
        `ASH2:      V_Sel = (V_Cur | (~DST_15 & (C_Cur ^ SRC_15))) ? `v_set : `v_clear;
        // ASHC
        `ASHC2:     V_Sel = (V_Cur | (~DST_15 & (C_Cur ^ SRC_31))) ? `v_set : `v_clear;
        // Multiply
        `MUL12:     V_Sel = `v_clear;
        // Divide
        `DIV8:      V_Sel = DST_Zero ? `v_set : `v_clear;
        `DIV9:      V_Sel = WD_Neg ? `v_clear : `v_set;
        // Memory Ops
        `MFPI1:     V_Sel = `v_clear;
        // Move to
        `MTPI1:     V_Sel = `v_clear;
        default:    V_Sel = `v_same;
    endcase // End V Select Logic

    /* Z Select Logic */
    case (State)
        `FETCH0:        Z_Sel = `z_same;
        `DOUBLE_EX0:    case (Instruction[`DOUBLE_OP1])
                            `BIT:   Z_Sel = `z_zero;
                            `CMP:   Z_Sel = `z_zero;
                            default: Z_Sel = `z_same;
                        endcase
        `WRITE_BACK0:   case (Instr_type)
                            `INSTR_SINGLE: case (Instruction[`SINGLE_OP])
                                               `CLR:   Z_Sel = `z_zero;
                                               `COMP:  Z_Sel = `z_zero;
                                               `INC:   Z_Sel = `z_zero;
                                               `DEC:   Z_Sel = `z_zero;
                                               `NEG:   Z_Sel = `z_zero;
                                               `ADDC:  Z_Sel = `z_zero;
                                               `SUBC:  Z_Sel = `z_zero;
                                               `TEST:  Z_Sel = `z_zero;
                                               `ROTR:  Z_Sel = `z_zero;
                                               `ROTL:  Z_Sel = `z_zero;
                                               `ASR:   Z_Sel = `z_zero;
                                               `ASL:   Z_Sel = `z_zero;
                                               `SXT:   Z_Sel = `z_zero;
                                               default: Z_Sel = `z_same;
                                           endcase
                            `INSTR_DOUBLE:  case (Instruction[`DOUBLE_OP1])
                                               `MOV:   Z_Sel = `z_zero;
                                               `CMP:   Z_Sel = `z_zero;
```

```
                                        `BIT:   Z_Sel = `z_zero;
                                        `BIC:   Z_Sel = `z_zero;
                                        `BIS:   Z_Sel = `z_zero;
                                        `ADD:   Z_Sel = `z_zero;
                                        `REG:   case (Instruction[`DOUBLE_OP2])
                                                    `ASH:   Z_Sel = `z_zero;
                                                    `XOR:   Z_Sel = `z_zero;
                                                    default: Z_Sel = `z_same;
                                                endcase
                                        default: Z_Sel = `z_same;
                                    endcase
                            `INSTR_SPECIAL_BRANCH:  Z_Sel = Instruction[7] ? (WD_Lo_Zero ? `z_set : `z_clear) :
                                default:    Z_Sel = `z_same;
                            endcase
     `REG_WRITE_BACK0:   case (Instr_type)
                            `INSTR_DOUBLE: case (Instruction[`DOUBLE_OP1])
                                        `REG: case (Instruction[`DOUBLE_OP2])
                                                `ASH: Z_Sel = `z_zero;
                                                `XOR: Z_Sel = `z_zero;
                                                default: Z_Sel = `z_same;
                                              endcase
                                        default: Z_Sel = `z_same;
                                    endcase
                            default:    Z_Sel = `z_same;
                            endcase
     `SCC0:      Z_Sel = Instruction[2] ? (Instruction[4] ? `z_set : `z_clear) : `z_same;
     // ASHC
     `ASHC5:     Z_Sel = `z_zero;
     `ASHC6:     Z_Sel = WD_Zero & Z_Cur ? `z_set : `z_clear;
     // Multiply
     `MUL7:      Z_Sel = `z_zero;
     `MUL11:     Z_Sel = WD_Zero & Z_Cur ? `z_set : `z_clear;
     // Divide
     `DIV13:     Z_Sel = `z_zero;
     // Memory Ops
     `MFPI1:     Z_Sel = `z_zero;
     // Move to
     `MTPI1:     Z_Sel = `z_zero;
     default:    Z_Sel = `z_same;
 endcase // End Z Select Logic

/* N Select Logic */
case (State)
     `FETCH0:        N_Sel = `n_same;
     `DOUBLE_EX0:    case (Instruction[`DOUBLE_OP1])
                        `BIT: N_Sel = `n_neg;
                        `CMP: N_Sel = `n_neg;
                        default: N_Sel = `n_same;
                     endcase
     `WRITE_BACK0:   case (Instr_type)
                        `INSTR_SINGLE: case (Instruction[`SINGLE_OP])
                                        `CLR:   N_Sel = `n_neg;
                                        `COMP:  N_Sel = `n_neg;
                                        `INC:   N_Sel = `n_neg;
                                        `DEC:   N_Sel = `n_neg;
                                        `NEG:   N_Sel = `n_neg;
```

```
                                        `ADDC:  N_Sel = `n_neg;
                                        `SUBC:  N_Sel = `n_neg;
                                        `TEST:  N_Sel = `n_neg;
                                        `ROTR:  N_Sel = `n_neg;
                                        `ROTL:  N_Sel = `n_neg;
                                        `ASR:   N_Sel = `n_neg;
                                        `ASL:   N_Sel = `n_neg;
                                        `SXT:   N_Sel = `n_same;
                                       default: N_Sel = `n_same;
                                    endcase
                            `INSTR_DOUBLE:  case (Instruction[`DOUBLE_OP1])
                                        `MOV:   N_Sel = `n_neg;
                                        `CMP:   N_Sel = `n_neg;
                                        `BIT:   N_Sel = `n_neg;
                                        `BIC:   N_Sel = `n_neg;
                                        `BIS:   N_Sel = `n_neg;
                                        `ADD:   N_Sel = `n_neg;
                                        `REG:   case (Instruction[`DOUBLE_OP2])
                                                    `ASH:   N_Sel = `n_neg;
                                                    `XOR:   N_Sel = `n_neg;
                                                    default: N_Sel = `n_same;
                                                endcase
                                       default: N_Sel = `n_same;
                                    endcase
                            `INSTR_SPECIAL_BRANCH:  N_Sel = Instruction[7] ? (WD_7 ? `n_set : `n_clear) : `n_sa
                            default:    N_Sel = `n_same;
                        endcase
        `REG_WRITE_BACK0:   case (Instr_type)
                                `INSTR_DOUBLE: case (Instruction[`DOUBLE_OP1])
                                                `REG: case (Instruction[`DOUBLE_OP2])
                                                        `ASH: N_Sel = `n_neg;
                                                        `XOR: N_Sel = `n_neg;
                                                        default: N_Sel = `n_same;
                                                      endcase
                                               default: N_Sel = `n_same;
                                             endcase
                                default:    N_Sel = `n_same;
                            endcase
        `SCC0:          N_Sel = Instruction[3] ? (Instruction[4] ? `n_set : `n_clear) : `n_same;
        // ASHC
        `ASHC6:         N_Sel = `n_neg;
        // Multiply
        `MUL11:         N_Sel = `n_neg;
        // Divide
        `DIV10:         N_Sel = `n_neg;
        `DIV13:         N_Sel = `n_neg;
        // Memory Ops
        `MFPI1:         N_Sel = `n_neg;
        // Move to
        `MTPI1:         N_Sel = `n_neg;
        default:        N_Sel = `n_same;
    endcase // End N Select Logic

    /* ASHC Select */
    case (State)
        `ASHC2:         Ashc = 1'b1;
```

```
    default:        Ashc = 1'b0;
endcase // End ASHC Select logic

/* PSW Select Logic */
case (State)
    `FETCH0:        PSW_Sel = `psw_same;    // Place Holder
    `TRAP3:         PSW_Sel = `psw_mem;
    `RTT2:          PSW_Sel = `psw_mem;
    default:        PSW_Sel = `psw_same;
endcase // End PSW Select Logic

/* Trap Vector Select Logic */
case (State)
    `FETCH0:        Trap_Sel = `trap_broken;
    `TRAP0:         if (Interrupt) begin
                        Trap_Sel = `trap_mem_data;
                    end
                    else if (Trace_Trap) begin
                        Trap_Sel = `trap_bpt;
                    end
                    else if (Instruction[15:8] == 8'h88) begin
                        Trap_Sel = `trap_emt;
                    end
                    else if (Instruction[15:8] == 8'h89) begin
                        Trap_Sel = `trap_trap;
                    end
                    else if (Instruction == 16'h0004) begin
                        Trap_Sel = `trap_io;
                    end
                    else if (Instruction == 16'h0003) begin
                        Trap_Sel = `trap_bpt;
                    end
                    else begin
                        Trap_Sel = `trap_broken;
                    end
    default:        Trap_Sel = `trap_broken;
endcase // End Trap Vector Select Logic

/* Select Priority
 * 1'b1 = select from SPL Instruction
 * 1'b0 = select the current priority
 */
case (State)
    `FETCH0:        Priority_Sel = 1'b0;
    `SPL0:          Priority_Sel = 1'b1;
    default:        Priority_Sel = 1'b0;
endcase // End Priority Select Logic

// Interrupt Clear logic
case (State)
    `FETCH0:        Interrupt_Clear = 1'b0;
    `TRAP0:         Interrupt_Clear = Interrupt;
    default: Interrupt_Clear = 1'b0;
endcase //End Interrupt Clear logic

/* Select the Current or Past Mem Space
```

```
     * 1'b1 is current mem space
     * 1'b0 is previous mem space
     */
    case (State)
        `FETCH0:        Mem_Space_Cur = 1'b1;
        `MFPI1:         Mem_Space_Cur = 1'b0;
        // Move to
        `MTPI2:         Mem_Space_Cur = 1'b0;
        `MTPI3:         Mem_Space_Cur = 1'b0;
        default:        Mem_Space_Cur = 1'b1;
    endcase // End Current Mem Space Select

    /* Select Instruction versus Data Space
     * 1'b1 is Instruction Space
     * 1'b0 is Data Space
     *
    case (State)
        `FETCH0:        Mem_Space_I = 1'b1;
        default:        Mem_Space_I = 1'b1;
    endcase // End Instruction v Data Space Selection */

end // End combinational Select Logic

assign Instr_Update_En = State == `FETCH2;


/* Masssive Combinational logic based on the current State
 * This is where all the control really happens */



/* Select what goes into the SRC Lo reg */
//assign SRC_Lo_Reg = 0;
//assign SRC_Lo_Mem = 0;
//assign SRC_Lo_SRC_LS = 0;
// All zero implies keep same

/* Select what goes into the SRC Hi reg */
//assign SRC_Hi_Reg = 0;
//assign SRC_Hi_Mem = 0;
//assign SRC_Hi_SRC_LS = 0;
// All Zeros implies keep same


/* Select what goes into DST reg */
//assign DST_Reg = 0;
//assign DST_Mem = 0;
//assign DST_DST_RS = 0;
// Zeros implies DST keeps same

/* Select what goes onto Write Data bus */
//assign WD_ACC_Lo = (State == `FETCH1);
//assign WD_ACC_Hi = 0;
//assign WD_ALU_Out = 0;
//assign WD_Shift_Out = 0;
//assign WD_Mem = 0;
```

```
    /* Select what goes into the ACC_Lo register, ACCL */
    //assign ACCL_ALU_Out = (State == `FETCH0);
    //assign ACCL_Shift_Out = 0;
    // Zeros implies that ACC_Lo keeps same value

    /* Select what goes into the ACC_Hi register, ACCH */
    //assign ACCH_ALU_Out = 0;
    //assign ACCH_Shift_Out = 0;
    // Zeros implies that ACC_Hi keeps same value

    //assign shift_use_src_hi = 0;
    //assign instruction_update = (State == `FETCH1);

endmodule
```

A.7 core.v

```
`timescale 1ns / 100ps
`include "pdp11.h"

module core(
    input           clk, clkb, sdi, scan,
                    reset, Interrupt,
    input   [15:0]  MDI,
    output  [15:0]  MAO,
    output  [7:0]   WD,
    output          MemWrite, MemEn, Kernal,
                    ReadByte, InterruptClear, sdo
    );
    wire    [15:0]  Instruction;
    wire    [8:0]   RegSelect, RegSelect_b,
                    HiEnable, HiEnable_b,
                    LowEnable, LowEnable_b;
    wire    [7:0]   ALUASel, ALUASel_b,
                    ALUBSel, ALUBSel_b;
    wire    [5:0]   DSTSel, DSTSel_b,
                    SRCLoSel, SRCLoSel_b;
    wire    [4:0]   WDWSel, WDWSel_b,
                    SRCHiSel, SRCHiSel_b,
                    MANSel, MANSel_b;
    wire    [3:0]   Flags, FlagsNext,
                    ACCLoSel, ACCLoSel_b,
                    ACCHiSel, ACCHiSel_b,
                    SFTLoSel, SFTLoSel_b;
    wire    [2:0]   Priority, PriorityNext,
                    PSWSel, PSWSel_b,
                    SFTConst, TrapVal,
                    SFTHiSel, SFTHiSel_b,
                    ALUAConst;
    wire    [1:0]   CurPrivilege, PrevPrivilege;
    wire            WDW_Zero, WDW_15, WDW_Lo_Zero, WDW_7,
```

```
                    SRC_Lo_15, SRC_Lo_7, SRC_Lo_0, SRC_Hi_15, SRC_Hi_0,
                    Acc_Hi_0, MemPSW, L7Z, L15Z, DST_0,
                    DST_7, DST_8, DST_15, CSDTOut, CALUOut,
                    TraceTrap, SFTAmt, MemReadPSW,
                    MemReadPSW_b, isByte, isByte_b,
                    WDSel, WDSel_b, Update, Update_b,
                    SFTAmt_b, rotate,
                    Ashc, InvShamt, MemHigh, MANen,
                    MANen_b, sdm, scan_b, reset_b,
                    ALUinvertB, ALUinvertA, add_b,
                    ALUC_in, enALU, enALU_b, ALUenAND, ALUenAND_b,
                    ALUenOR, ALUenOR_b, ALUenCOMP, ALUenCOMP_b,
                    Lo_Lo_Bit, Lo_Hi_Bit, Lo_Mid_Bit, clk_b, clkb_b;

//    always@(*) $display("MDI %h %h, MemReadPSW %b, MemReadPSW_b %b, MRD %h, WDW %h", MDI, dp.RB.MDI, dp.RB.MemRea
    SuperController sc(
        .Instruction(Instruction), .Flags(Flags), .Priority(Priority),
        .CurPrivilege(CurPrivilege), .PrevPrivilege(PrevPrivilege),
        .WDW_Zero(WDW_Zero), .WDW_15(WDW_15), .WDW_Lo_Zero(WDW_Lo_Zero), .WDW_7(WDW_7),
        .SRC_Lo_15(SRC_Lo_15), .SRC_Lo_7(SRC_Lo_7), .SRC_Lo_0(SRC_Lo_0), .SRC_Hi_15(SRC_Hi_15), .SRC_Hi_0(SRC_Hi_0)
        .Acc_Hi_0(Acc_Hi_0), .MemPSW(MemPSW),
        .L7Z(L7Z), .L15Z(L15Z), .DST_0(DST_0), .DST_7(DST_7), .SRC_8(SRC_8), .DST_15(DST_15),
        .CWALUOut(CWALUOut), .CBALUOut(CBALUOut), .TraceTrap(TraceTrap), .sdi(sdi),
        .clk(clk), .clkb(clkb), .Interrupt(Interrupt), .reset(reset), .scan(scan),
        .RegSelect(RegSelect), .RegSelect_b(RegSelect_b), .HiEnable(HiEnable),
        .HiEnable_b(HiEnable_b), .LowEnable(LowEnable), .LowEnable_b(LowEnable_b),
        .ALUASel(ALUASel), .ALUASel_b(ALUASel_b), .ALUBSel(ALUBSel), .ALUBSel_b(ALUBSel_b),
        .ACCLoSel(ACCLoSel), .DSTSel(DSTSel), .DSTSel_b(DSTSel_b), .SRCLoSel(SRCLoSel), .SRCLoSel_b(SRCLoSel_b),
        .ACCLoSel_b(ACCLoSel_b), .SRCHiSel(SRCHiSel), .SRCHiSel_b(SRCHiSel_b),
        .WDWSel(WDWSel), .WDWSel_b(WDWSel_b),
        .ACCHiSel(ACCHiSel), .ACCHiSel_b(ACCHiSel_b),
        .FlagsNext(FlagsNext),
        .SFTConst(SFTConst), .TrapVal(TrapVal),
        .PriorityNext(PriorityNext), .PSWSel(PSWSel), .PSWSel_b(PSWSel_b),
        .ALUAConst(ALUAConst), .MANSel(MANSel), .MANSel_b(MANSel_b), .SFTAmt(SFTAmt),
        .MemReadPSW(MemReadPSW), .MemReadPSW_b(MemReadPSW_b),
        .WDSel(WDSel), .WDSel_b(WDSel_b),
        .Update(Update), .Update_b(Update_b), .SFTAmt_b(SFTAmt_b),
        .MemHigh(MemHigh), .MANen(MANen), .MANen_b(MANen_b),
        .sdo(sdm), .scan_b(scan_b), .reset_b(reset_b),
        .MemEn(MemEn), .MemWrite(MemWrite), .Kernal(Kernal), .ReadByte(ReadByte), .ReadByte_b(ReadByte_b),
        .InterruptClear(InterruptClear), .Lo_Lo_Bit(Lo_Lo_Bit), .Lo_Hi_Bit(Lo_Hi_Bit), .Lo_Mid_Bit(Lo_Mid_Bit),
        .ALUinvertB(ALUinvertB), .ALUinvertA(ALUinvertA), .add_b(add_b),
        .ALUC_in(ALUC_in), .enALU(enALU), .enALU_b(enALU_b), .ALUenAND(ALUenAND), .ALUenAND_b(ALUenAND_b),
        .ALUenOR(ALUenOR), .ALUenOR_b(ALUenOR_b), .ALUenCOMP(ALUenCOMP), .ALUenCOMP_b(ALUenCOMP_b),
        .clk_b(clk_b), .clkb_b(clkb_b)
    );
    Datapath dp(
        .MDI(MDI),
        .RegSelect(RegSelect), .RegSelect_b(RegSelect_b),
        .HiEnable(HiEnable),  .HiEnable_b(HiEnable_b),
        .LowEnable(LowEnable), .LowEnable_b(LowEnable_b),
        .ALUASel(ALUASel), .ALUASel_b(ALUASel_b),
        .ALUBSel(ALUBSel), .ALUBSel_b(ALUBSel_b),
        .DSTSel(DSTSel), .DSTSel_b(DSTSel_b),
        .WDWSel(WDWSel), .WDWSel_b(WDWSel_b),
```

```
        .SRCLoSel(SRCLoSel), .SRCLoSel_b(SRCLoSel_b),
        .ACCLoSel(ACCLoSel), .ACCLoSel_b(ACCLoSel_b),
        .ACCHiSel(ACCHiSel), .ACCHiSel_b(ACCHiSel_b),
        .FlagsNext(FlagsNext),
        .SRCHiSel(SRCHiSel), .SRCHiSel_b(SRCHiSel_b),
        .PriorityNext(PriorityNext),
        .PSWSel(PSWSel), .PSWSel_b(PSWSel_b),
        .ALUAConst(ALUAConst), .SFTConst(SFTConst),
        .TrapVal(TrapVal), .MANSel(MANSel), .MANSel_b(MANSel_b),
        .sdi(sdi),
        .clk(clk), .clkb(clkb), .clk_b(clk_b), .clkb_b(clkb_b),
        .scan(scan), .scan_b(scan_b),
        .reset(reset),
        .MemReadPSW(MemReadPSW), .MemReadPSW_b(MemReadPSW_b),
        .ReadByte_b(ReadByte_b),
        .WDSel(WDSel), .WDSel_b(WDSel_b),
        .Update(Update), .Update_b(Update_b),
        .SFTAmt(SFTAmt), .SFTAmt_b(SFTAmt_b),
        .MemHigh(MemHigh), .MANen(MANen), .MANen_b(MANen_b),
        .ALUinvertB(ALUinvertB), .ALUinvertA(ALUinvertA), .add_b(add_b),
        .ALUC_in(ALUC_in), .enALU(enALU), .enALU_b(enALU_b), .ALUenAND(ALUenAND), .ALUenAND_b(ALUenAND_b),
        .ALUenOR(ALUenOR), .ALUenOR_b(ALUenOR_b), .ALUenCOMP(ALUenCOMP), .ALUenCOMP_b(ALUenCOMP_b),
        .Lo_Lo_Bit(Lo_Lo_Bit), .Lo_Hi_Bit(Lo_Hi_Bit), .Lo_Mid_Bit(Lo_Mid_Bit),
        .Instruction(Instruction), .MAO(MAO),
        .WD(WD),
        .Flags(Flags),
        .Priority(Priority),
        .CurPrivilege(CurPrivilege), .PrevPrivilege(PrevPrivilege),
        .sdo(sdo),
        .WDW_Zero(WDW_Zero), .WDW_15(WDW_15), .WDW_Lo_Zero(WDW_Lo_Zero), .WDW_7(WDW_7),
        .SRC_Lo_15(SRC_Lo_15), .SRC_Lo_7(SRC_Lo_7), .SRC_Lo_0(SRC_Lo_0), .SRC_Hi_15(SRC_Hi_15), .SRC_Hi_0(SRC_Hi_0)
        .L7Z(L7Z), .L15Z(L15Z), .DST_0(DST_0), .DST_7(DST_7), .SRC_8(SRC_8), .DST_15(DST_15),
        .CWALUOut(CWALUOut), .CBALUOut(CBALUOut),
        .TraceTrap(TraceTrap), .MemPSW(MemPSW), .ACCHi0(Acc_Hi_0)
    );

endmodule
```

A.8 Datapath.v

```
`timescale 1ns / 100ps
`include "pdp11.h"
module Datapath(
    input       [15:0]  MDI,
    input       [8:0]   RegSelect, RegSelect_b,
                        HiEnable,  HiEnable_b,
                        LowEnable, LowEnable_b,
    input       [7:0]   ALUASel, ALUASel_b,
                        ALUBSel, ALUBSel_b,
    input       [5:0]   DSTSel, DSTSel_b,
    input       [4:0]   WDWSel, WDWSel_b,
                        MANSel, MANSel_b,
    input       [5:0]   SRCLoSel, SRCLoSel_b,
    input       [4:0]   SRCHiSel, SRCHiSel_b,
    input       [3:0]   FlagsNext,
                        ACCLoSel, ACCLoSel_b,
```

```
                    ACCHiSel, ACCHiSel_b,
input       [2:0]   PriorityNext,
                    PSWSel, PSWSel_b,
                    ALUAConst, SFTConst,
                    TrapVal,
input               sdi,
                    clk, clkb, clk_b, clkb_b,
                    scan, scan_b,
                    reset,
                    MemReadPSW, MemReadPSW_b,
                    ReadByte_b,
                    WDSel, WDSel_b,
                    Update, Update_b,
                    SFTAmt, SFTAmt_b,
                    MemHigh, MANen, MANen_b,
                    ALUinvertB, ALUinvertA, add_b,
                    ALUC_in, enALU, enALU_b, ALUenAND, ALUenAND_b,
                    ALUenOR, ALUenOR_b, ALUenCOMP, ALUenCOMP_b,
                    Lo_Lo_Bit, Lo_Hi_Bit, Lo_Mid_Bit,
output      [15:0]  Instruction, MAO,
output      [7:0]   WD,
output      [3:0]   Flags,
output      [2:0]   Priority,
output      [1:0]   CurPrivilege, PrevPrivilege,
output              sdo,
                    WDW_Zero, WDW_15, WDW_Lo_Zero, WDW_7,
                    SRC_Lo_15, SRC_Lo_7, SRC_Lo_0, SRC_Hi_15, SRC_Hi_0,
                    L7Z, L15Z, DST_0, DST_7, SRC_8, DST_15,
                    CWALUOut, CBALUOut,
                    TraceTrap, MemPSW, ACCHi0
);
wire        [15:0]  WDW, MRD, RF, MAN,
                    ALUA, ALUB;
wire        [5:0]   Shamt;
// these need to be put somewhere
    WDW_Block WB(.WDW(WDW),.WDW_Zero(WDW_Zero), .WDW_15(WDW_15), .WDW_Lo_Zero(WDW_Lo_Zero), .WDW_7(WDW_7));
RegPSW_Block RB(
    .WDW(WDW), .MDI(MDI),
    .RegSelect(RegSelect), .RegSelect_b(RegSelect_b),
    .HiEnable(HiEnable), .HiEnable_b(HiEnable_b),
    .LowEnable(LowEnable), .LowEnable_b(LowEnable_b),
    .MemReadPSW(MemReadPSW), .MemReadPSW_b(MemReadPSW_b),
    .WD_MEM(WDWSel[2]), .WD_MEM_b(WDWSel_b[2]),
    .WD_PSW(WDWSel[1]), .WD_PSW_b(WDWSel_b[1]),
    .sdi(sdi), .scan(scan), .scan_b(scan_b),
    .clk(clk), .clkb(clkb), .clk_b(clk_b), .clkb_b(clkb_b), .reset(reset),
    .byte_b(ReadByte_b),
    .WDSel(WDSel_b), .WDSel_b(WDSel),
    .PriorityNext(PriorityNext),
    .FlagsNext(FlagsNext), .PSWSel(PSWSel), .PSWSel_b(PSWSel_b),
    .TraceTrap(TraceTrap),
    .MRD(MRD), .RF(RF), .WD(WD), .Flags(Flags), .Priority(Priority),
    .CurPrivilege(CurPrivilege), .PrevPrivilege(PrevPrivilege)
);
SD_Block SB(
    .WDW(WDW), .MRD(MRD), .RF(RF),
```

```
    .Lo_Lo_Bit(Lo_Lo_Bit), .Lo_Hi_Bit(Lo_Hi_Bit),
    .Lo_Mid_Bit(Lo_Mid_Bit),
    .DST_Sel(DSTSel), .DST_Sel_b(DSTSel_b),
    .SRC_Lo_Sel(SRCLoSel), .SRC_Lo_Sel_b(SRCLoSel_b),
    .SRC_Hi_Sel(SRCHiSel), .SRC_Hi_Sel_b(SRCHiSel_b),
    .ALUBSel(ALUBSel[3:0]), .ALUBSel_b(ALUBSel_b[3:0]),
    .sdi(RF[0]), .scan(scan), .scan_b(scan_b),
    .clk(clk), .clkb(clkb), .clk_b(clk_b), .clkb_b(clkb_b), .reset(reset), .Instruction(Instruction[5:0]),
    .ALUASel(ALUASel[1:0]), .ALUASel_b(ALUASel_b[1:0]),
    .ALUA(ALUA), .ALUB(ALUB),  .SFTConst(SFTConst),
    .SFTAmt(SFTAmt), .SFTAmt_b(SFTAmt_b),
    .SRC_Lo_15(SRC_Lo_15), .SRC_Lo_7(SRC_Lo_7), .SRC_Lo_0(SRC_Lo_0),
    .SRC_Hi_15(SRC_Hi_15), .SRC_Hi_0(SRC_Hi_0), .L7Z(L7Z), .L15Z(L15Z),
    .DST_0(DST_0), .DST_7(DST_7), .SRC_8(SRC_8), .DST_15(DST_15)

);
Instruction_Block IB(
    .MRD(MRD),
    .sdi(DST_0), .scan(scan), .scan_b(scan_b),
    .clk(clk), .clkb(clkb),
    .clk_b(clk_b), .clkb_b(clkb_b),
    .reset(reset),
    .Update(Update), .Update_b(Update_b),
    .Instruction(Instruction)
);
Ari_Block AB(
    .ALUA(ALUA), .ALUB(ALUB),
    .WDW(WDW), .MRD(MRD),
    .Instruction(Instruction[7:0]),
    .ACCLoSel(ACCLoSel), .ACCLoSel_b(ACCLoSel_b),
    .ACCHiSel(ACCHiSel), .ACCHiSel_b(ACCHiSel_b),
    .TrapVal(TrapVal),
    .MANSel(MANSel[3:0]), .MANSel_b(MANSel_b[3:0]),
    .ALUAConst(ALUAConst),
    .ALUASel(ALUASel[4:2]), .ALUASel_b(ALUASel_b[4:2]),
    .WDWSel({WDWSel[4:3],WDWSel[0]}), .WDWSel_b({WDWSel_b[4:3],WDWSel_b[0]}),
    .clk(clk), .clkb(clkb), .clk_b(clk_b), .clkb_b(clkb_b), .scan(scan), .scan_b(scan_b),
    .reset(reset), .sdi(Instruction[0]),
    .ALUBSel(ALUBSel[7:4]), .ALUBSel_b(ALUBSel_b[7:4]),
    .ALUinvertB(ALUinvertB), .ALUinvertA(ALUinvertA), .add_b(add_b),
    .ALUC_in(ALUC_in), .enALU(enALU), .enALU_b(enALU_b), .ALUenAND(ALUenAND), .ALUenAND_b(ALUenAND_b),
    .ALUenOR(ALUenOR), .ALUenOR_b(ALUenOR_b), .ALUenCOMP(ALUenCOMP), .ALUenCOMP_b(ALUenCOMP_b),
    .MAN(MAN),
    .ACCHi0(ACCHi0),
    .CWALUOut(CWALUOut), .CBALUOut(CBALUOut)
);
MemAddr_Block MAB(
    .MAN(MAN),
    .clk(clk), .clkb(clkb), .clk_b(clk_b), .clkb_b(clkb_b),
    .scan(scan), .scan_b(scan_b),
    .reset(reset), .sdi(ACCHi0),
    .MemHigh(MemHigh),
    .en(MANSel[4]), .en_b(MANSel_b[4]),
    .MAO(MAO),
    .sdo(sdo),.MemPSW(MemPSW)
);
```

```
endmodule
```

A.9 Dram.v

```
‘timescale 1ns / 100ps
module Dram(

    output          ReadOut,
    input           WriteData,
    input           RegSelect, WriteEnable
    );

    wire            Value, net40;
    assign  Value   =   WriteEnable ? WriteData     : Value ;
    assign  ReadOut =   RegSelect   ? Value         : 1’bz  ;

    assign  net40   =   Value;
endmodule
```

A.10 DRegister.v

```
‘timescale 1ns / 100ps
module DRegister(
    input           RegSelect, HiEnable, LowEnable,
    input   [15:0]  WriteData,
    output  [15:0]  ReadOut
    );
    Dram reg_hi[15:8] (ReadOut[15:8], WriteData[15:8], RegSelect, HiEnable );
    Dram reg_low[7:0] (ReadOut[7:0],  WriteData[7:0] , RegSelect, LowEnable);
endmodule
```

A.11 DST_Block.v

```
‘timescale 1ns / 100ps
‘include "pdp11.h"
module DST_Block(
    input       [15:0]  WDW, MRD, RF,
    input               sdi, scan, scan_b, clk, clk_b, clkb, clkb_b, reset,
    input       [5:0]   DST_Sel, DST_Sel_b,
    input       [5:0]   Shamt,
    input               ALUASel, ALUASel_b, ALUBSel, ALUBSel_b,
    output      [15:0]  ALUA, ALUB,
    output              L7Z, L15Z,
                        DST_0, DST_7, DST_15
    );

    wire        [15:0]  DST, DST_Next;

    assign  DST_0   =    DST[0]          ;
    assign  L7Z     =  ~|DST[6:0]        ;
    assign  DST_7   =    DST[7]          ;
    assign  L15Z    = (~|DST[14:7]) & L7Z;
    assign  DST_15  =    DST[15]         ;
```

```
    Mux6_dp #(.WIDTH(16)) dst_mux(
        .d0(RF[15:0])           ,
        .d1(MRD[15:0])          ,
        .d2({1'b0,DST[15:1]})   ,
        .d3(WDW[15:0])          ,
        .d4(DST[15:0])          ,
        .d5({{10{Shamt[5]}},Shamt}),
        .sel(DST_Sel)      ,
        .sel_b(DST_Sel_b)  ,
        .q(DST_Next[15:0]));

    scanflop_r #(.WIDTH(16)) DST_SR(
        .d(DST_Next[15:0]),
        .reset(reset),
        .clk(clk), .clk_b(clk_b),
        .clkb(clkb), .clkb_b(clkb_b),
        .sdi(sdi),
        .scan(scan), .scan_b(scan_b),
        .q(DST[15:0])
    );
    Tri_dp #(.WIDTH(16)) ALUA_Tri (.d(DST[15:0]), .en(ALUASel ), .en_b(ALUASel_b ), .q(ALUA[15:0]) );
    Tri_dp #(.WIDTH(16)) ALUB_Tri (.d(DST[15:0]), .en(ALUBSel ), .en_b(ALUBSel_b ), .q(ALUB[15:0]) );
endmodule
```

A.12 Instruction_Block.v

```
`timescale 1ns / 100ps
`include "pdp11.h"
module Instruction_Block(
    input   [15:0]  MRD,
    input           sdi, scan, scan_b, clk, clk_b, clkb, clkb_b, reset,
    input           Update, Update_b,
    output  [15:0]  Instruction
    );
    wire    [15:0]  Instr_Next;
    Mux2_OneHot #(.WIDTH(16)) dst_mux(
        .d0(Instruction[15:0]) ,
        .d1(MRD[15:0]),
        .sel0(Update_b), .sel1(Update),
        .q(Instr_Next[15:0]));

    scanflop_r #(.WIDTH(16)) INSTR_SR(
        .d(Instr_Next[15:0]),
        .reset(reset),
        .clk(clk), .clk_b(clk_b),
        .clkb(clkb), .clkb_b(clkb_b),
        .sdi(sdi),
        .scan(scan), .scan_b(scan_b),
        .q(Instruction[15:0])
    );
endmodule
```

A.13 MemAddr_Block.v

```
‘timescale 1ns / 100ps
module MemAddr_Block(
        inout   [15:0]  MAN,
        input           clk, clk_b, clkb, clkb_b,
                        scan, scan_b,
                        reset, sdi,
                        MemHigh,
                        en, en_b,
        output  [15:0]  MAO,
        output          sdo, MemPSW
    );
    wire    [15:0] MAB;
    Tri_dp #(.WIDTH(16)) MAN_Tri(
        .d(MAB),
        .en(en), .en_b(en_b),
        .q(MAN)
    );
    scanflop_r   #(.WIDTH(16)) MAB_SR(
        .d(MAN),
        .clk(clk), .clk_b(clk_b),
        .clkb(clkb), .clkb_b(clkb_b),
        .reset(reset), .sdi(sdi),
        .scan(scan), .scan_b(scan_b),
        .q(MAB)
    );
    assign MAO    = {MAB[15:1], MAB[0]|MemHigh};
    assign MemPSW = &{MAB[15:1], ~MAB[0]};
    assign sdo    =  MAB[0];
endmodule
```

A.14 Mux.v

```
‘timescale 1ns / 100ps
module Tri #(parameter WIDTH=16) (
    input [WIDTH-1:0] d,
    input en,
    output [WIDTH-1:0] q
    );

    assign q[WIDTH-1:0] = en ? d[WIDTH-1:0] : {WIDTH{1’bz}};
endmodule

module Tri_dp #(parameter WIDTH=16) (
    input [WIDTH-1:0] d,
    input en, en_b,
    output [WIDTH-1:0] q
    );

    assign q[WIDTH-1:0] = (en & ~en_b) ? d[WIDTH-1:0] : {WIDTH{1’bz}};
endmodule

module Mux2 #(parameter WIDTH=16) (input  [WIDTH-1:0] d0, d1,
                                   input  sel,
                                   output reg [WIDTH-1:0] q);
```

```
    always @(*) begin
        case (sel)
            1'b0: q = d0;
            1'b1: q = d1;
        endcase
    end

endmodule

module Mux2_OneHot #(parameter WIDTH=16) (input  [WIDTH-1:0] d0, d1,
                                          input  sel0, sel1,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        if (sel0 & ~sel1)
            q = d0;
        else if (sel1 & ~sel0)
            q = d1;
        else if (~(sel1 | sel0))
            q = {(WIDTH){1'bz}};
        else
            q = {(WIDTH){1'bx}};
    end

endmodule

module Mux2_dp #(parameter WIDTH=16) (input  [WIDTH-1:0] d0, d1,
                                          input[1:0]  sel, sel_b,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        case (sel & ~sel_b)
            2'b01: q = d0;
            2'b10: q = d1;
            2'b00: q = {(WIDTH){1'bz}};
            default: q = {(WIDTH){1'bx}};
        endcase
    end
endmodule

module Mux3_OneHot #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,
                                          input  sel0,sel1,sel2,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        case ({sel0,sel1,sel2})
            3'b100: q = d0;
            3'b010: q = d1;
            3'b001: q = d2;
            default: q = {(WIDTH){1'bx}};
        endcase
    end

endmodule

module Mux3_dp #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,
```

```
                                    input     [2:0] sel, sel_b,
                                    output reg [WIDTH-1:0] q);
    always @(*) begin
        case (sel & ~sel_b)
            3'b001: q = d0;
            3'b010: q = d1;
            3'b100: q = d2;
            3'b000: q = {(WIDTH){1'bz}};
            default: q = {(WIDTH){1'bx}};
        endcase
    end
endmodule

module Mux4_OneHot #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,d3,
                                          input  sel0,sel1,sel2,sel3,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        case ({sel0,sel1,sel2,sel3})
            4'b1000: q = d0;
            4'b0100: q = d1;
            4'b0010: q = d2;
            4'b0001: q = d3;
            default: q = {(WIDTH){1'bx}};
        endcase
    end

endmodule

module Mux4_dp #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,d3,
                                      input  [3:0] sel, sel_b,
                                      output reg [WIDTH-1:0] q);
    always @(*) begin
        case (sel & ~sel_b)
            4'b0001: q = d0;
            4'b0010: q = d1;
            4'b0100: q = d2;
            4'b1000: q = d3;
            4'b0000: q = {(WIDTH){1'bz}};
            default: q = {(WIDTH){1'bx}};
        endcase
    end
endmodule

module Mux5_OneHot #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,d3,d4,
                                          input  sel0,sel1,sel2,sel3,sel4,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        case ({sel0,sel1,sel2,sel3,sel4})
            5'b1_0000: q = d0;
            5'b0_1000: q = d1;
            5'b0_0100: q = d2;
            5'b0_0010: q = d3;
            5'b0_0001: q = d4;
            default: q = {(WIDTH){1'bx}};
```

```
        endcase
    end

endmodule

module Mux5_dp #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,d3,d4,
                                      input     [4:0] sel, sel_b,
                                      output reg [WIDTH-1:0] q);
    always @(*) begin
        case (sel & ~sel_b)
            5'b0_0001: q = d0;
            5'b0_0010: q = d1;
            5'b0_0100: q = d2;
            5'b0_1000: q = d3;
            5'b1_0000: q = d4;
            5'b0_0000: q = {(WIDTH){1'bz}};
            default: q = {(WIDTH){1'bx}};
        endcase
    end
endmodule

module Mux6_dp #(parameter WIDTH=16) (input [WIDTH-1:0] d0,d1,d2,d3,d4,d5,
                                      input [5:0] sel, sel_b,
                                      output reg [WIDTH-1:0] q);

    always @(*) begin
        case (sel & ~sel_b)
            6'b00_0001: q = d0;
            6'b00_0010: q = d1;
            6'b00_0100: q = d2;
            6'b00_1000: q = d3;
            6'b01_0000: q = d4;
            6'b10_0000: q = d5;
            default: q = {(WIDTH){1'bx}};
        endcase
    end
endmodule

module Mux6_OneHot #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,d3,d4,d5,
                                          input  sel0,sel1,sel2,sel3,sel4,sel5,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        case ({sel0,sel1,sel2,sel3,sel4,sel5})
            6'b10_0000: q = d0;
            6'b01_0000: q = d1;
            6'b00_1000: q = d2;
            6'b00_0100: q = d3;
            6'b00_0010: q = d4;
            6'b00_0001: q = d5;
            default: q = {(WIDTH){1'bx}};
        endcase
    end

endmodule
```

```
module Mux7_OneHot #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,d3,d4,d5,d6,
                                          input  sel0,sel1,sel2,sel3,sel4,sel5,sel6,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        case ({sel0,sel1,sel2,sel3,sel4,sel5,sel6})
            7'b100_0000: q = d0;
            7'b010_0000: q = d1;
            7'b001_0000: q = d2;
            7'b000_1000: q = d3;
            7'b000_0100: q = d4;
            7'b000_0010: q = d5;
            7'b000_0001: q = d6;
            default: q = {(WIDTH){1'bx}};
        endcase
    end

endmodule

module Mux8_OneHot #(parameter WIDTH=16) (input  [WIDTH-1:0] d0,d1,d2,d3,d4,d5,d6,d7,
                                          input  sel0,sel1,sel2,sel3,sel4,sel5,sel6,sel7,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        case ({sel0,sel1,sel2,sel3,sel4,sel5,sel6,sel7})
            8'b1000_0000: q = d0;
            8'b0100_0000: q = d1;
            8'b0010_0000: q = d2;
            8'b0001_0000: q = d3;
            8'b0000_1000: q = d4;
            8'b0000_0100: q = d5;
            8'b0000_0010: q = d6;
            8'b0000_0001: q = d7;
            default: q = {(WIDTH){1'bx}};
        endcase
    end

endmodule

module Mux9_OneHot #(parameter WIDTH=16) (input [WIDTH-1:0] d0,d1,d2,d3,d4,d5,d6,d7,d8,
                                          input sel0,sel1,sel2,sel3,sel4,sel5,sel6,sel7,sel8,
                                          output reg [WIDTH-1:0] q);

    always @(*) begin
        case({sel0,sel1,sel2,sel3,sel4,sel5,sel6,sel7,sel8})
            9'b1_0000_0000: q = d0;
            9'b0_1000_0000: q = d1;
            9'b0_0100_0000: q = d2;
            9'b0_0010_0000: q = d3;
            9'b0_0001_0000: q = d4;
            9'b0_0000_1000: q = d5;
            9'b0_0000_0100: q = d6;
            9'b0_0000_0010: q = d7;
            9'b0_0000_0001: q = d8;
            default: q = {(WIDTH){1'bx}};
        endcase
```

```
    end

endmodule

module Mux11_OneHot #(parameter WIDTH=16) (input [WIDTH-1:0] d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,
                                           input sel0,sel1,sel2,sel3,sel4,sel5,sel6,sel7,sel8,sel9,sel10,
                                           output reg [WIDTH-1:0] q);

    always @(*) begin
        case({sel0,sel1,sel2,sel3,sel4,sel5,sel6,sel7,sel8,sel9,sel10})
            11'b100_0000_0000: q = d0;
            11'b010_0000_0000: q = d1;
            11'b001_0000_0000: q = d2;
            11'b000_1000_0000: q = d3;
            11'b000_0100_0000: q = d4;
            11'b000_0010_0000: q = d5;
            11'b000_0001_0000: q = d6;
            11'b000_0000_1000: q = d7;
            11'b000_0000_0100: q = d8;
            11'b000_0000_0010: q = d9;
            11'b000_0000_0001: q = d10;
            default: q = {(WIDTH){1'bx}};
        endcase
    end

endmodule
```

A.15 pdp11.h

```
/* PDP-11 Defines for a simple PDP-11 Processor
 * Includes op code definitions for decoding
 * and clock definitions for timing
 *
 * William Koven (wkoven@hmc.edu)
 * Harvey Mudd College
 * 18 February 2011
 *
 */


// Clock Parameters
`define cycle   10
`define phase   5
`define TICK    #2
`define CYCLE   #`cycle
`define PHASE   #`phase

`define PSW_RESET       16'h0000
`define DEFAULT_USER_STACK  16'h0200
`define DEFAULT_SUPER_STACK 16'h0000
`define DEFUALT_KERNAL_STACK 16'h0200

`define PSW_MEM_ADDR    16'hFFFE
```

```
`define STATE_LEN       8

`define PC              3'b111
`define SP              3'b110

/* OPCODE detect */
`define DOUBLE_OP1      14:12
`define DOUBLE_OP2      11:9
`define PREFIX          14:11
`define SINGLE_OP       10:6
`define JUMP_BITS       10:9
`define TRAP_BIT        8
`define BRANCH_OP       10:8

`define DST_REG         2:0
`define DST_MODE        5:3
`define SRC_REG         8:6
`define SRC_MODE        11:9

`define PREFIX_SINGLE   4'b0001
`define PREFIX_BRANCH   4'b0000
`define JUMP_INST       2'b00

/* Single Operand Instruction Opcodes */

`define SWAB            5'b00011    // Swab bytes, or rotate 8 bits
`define JSR             5'b00100    // Jump to subroutine, if instr[15] then Emulator Trap
`define CLR             5'b01000    // Clear
`define COMP            5'b01001    // Complement
`define INC             5'b01010    // Increment
`define DEC             5'b01011    // Decrement
`define NEG             5'b01100    // Negate
`define ADDC            5'b01101    // Add Carry
`define SUBC            5'b01110    // Subtract Carry
`define TEST            5'b01111    // Test
`define ROTR            5'b10000    // Rotate Right
`define ROTL            5'b10001    // Rotate Left
`define ASR             5'b10010    // Arithmetic Shift Right
`define ASL             5'b10011    // Arithmetic Shift Left
`define MARK            5'b10100    // MARK, if instr[15] then MTPS
`define MFPI            5'b10101    // MFPI, if instr[15] then MFPD
`define MTPI            5'b10110    // MTPI, if instr[15] then MTPD
`define SXT             5'b10111    // Sign Extend
`define CSM             5'b11000    // Call to Supervisor

/* Double Operand Instruction, src and dest, Opcodes */
`define MOV             3'b001      // MOVE
`define CMP             3'b010      // Compare, src - dst, set flags only
`define BIT             3'b011      // Bit test, src & dst, set flags only
`define BIC             3'b100      // Bit clear, dst &= ~src
`define BIS             3'b101      // Bit Set, dst |= src
`define ADD             3'b110      // Add, dst += src, bit 15 means Subtract instead
`define REG             3'b111      // Src must be a register only, see below


/* Double Operand Instructions, src (R) must be reg only, Opcodes */
```

```
`define MUL             3'b000      // Signed multiply, R*dst, result in R,R+1 if R even
                                    // if R is odd only low 16 bits of result in R
`define DIV             3'b001      // Signed divide, R,R+1/dest, result in R, remainder
                                    // in R+1, remainder has same sign as R, R must be even
`define ASH             3'b010      // Arithmetic shift, -32 to 31, negative is right
`define ASHC            3'b011      // Arithmetic shift combined, R,R+1 are treated as
                                    // a single 32 bit operand and shifted -32 to 31
`define XOR             3'b100      // XOR, dst ^= R, word only
`define FLOP            3'b101      // FLoating point OPeration, not supported
`define SYS             3'b110      // System instructions
`define SOB             3'b111      // Decrement R (R -= 1), branch not equal 0
                                    // branch backwards 0..63 words, or
                                    // PC = PC - 2(offset)

/* Branch Instructions, instr[15] == 0 */
`define SYSB            3'b000      // System instructions
`define BR              3'b001      // Branch unconditionally
`define BNE             3'b010      // Branch if not equal (Z=0)
`define BEQ             3'b011      // Branch if equal (Z=1)
`define BGE             3'b100      // Branch greater or equal (N|V = 0)
`define BLT             3'b101      // Branch if less than (N|V = 1)
`define BGT             3'b110      // Branch if greater than (N^V = 1)
`define BLE             3'b111      // Branch if less than or equal (N^V = 0)

/* Branch Instructions, instr[15] == 1 */
`define BPL             3'b000      // Branch if plus (N=0)
`define BMI             3'b001      // Branch if minus (N=1)
`define BHI             3'b010      // Branch if higher than (C|Z = 0)
`define BLOS            3'b011      // Branch if lower or same (C|Z = 1)
`define BVC             3'b100      // Branch if overflow clear (V = 0)
`define BVS             3'b101      // Branch if overflow set (V = 1)
`define BCC             3'b110      // Branch if carry clear or
                                    // Branch if higher or same (BHIS) (C = 0)
`define BCS             3'b111      // Branch if carry set or
                                    // Branch if lower than (BLO) (C = 1)
/* Special Instructions, instr[2:0] */
`define HALT            3'b000      // HALT instruction
`define WAIT            3'b001      // WAIT instruction
`define RTI             3'b010      // Return from Interrupt
`define BPT             3'b011      // Break Point Trap
`define IOT             3'b100      // I/O Trap
`define NSI             3'b101      // No Such Instruction
`define RTT             3'b110      // Return from Trap
`define MFPT            3'b111      // Move From Processor PDP-11/44 Only

/* ALU Control Defines */
`define ALU_CTRL_LEN    8           // Width of the ALU control bus
`define ALU_AND         7           // Bit to determine if the ALU is ANing
`define ALU_ADD_SUB     6:4         // Bits to determine if the ALU is adding or subtracting
`define ALU_ADD         6           // Add
`define ALU_SUB_B       5           // Subtract, A + (-B)
`define ALU_SUB_A       4           // Subtract, B + (-A)
`define ALU_BIC         3           // Bit to determine if the ALU is BICing (A & ~B)
`define ALU_OR          2           // Bit to determine if the ALU is ORing
`define ALU_XOR         1           // Bit to determine if the ALU is XORing
`define ALU_COMP        0           // Bit to determine if the ALU is complementing (ALU_B is complemented)
```

```
‘define alu_and         8’b1000_0000
‘define alu_add         8’b0100_0000
‘define alu_sub_b       8’b0010_0000
‘define alu_sub_a       8’b0001_0000
‘define alu_bic         8’b0000_1000
‘define alu_or          8’b0000_0100
‘define alu_xor         8’b0000_0010
‘define alu_comp        8’b0000_0001

/* ALU Source A Select Defines */
‘define ALU_A_SEL_LEN   8           // Length of the ALU Source A Select Bus
‘define ALU_A_DST       0           // Use DST
‘define ALU_A_ACC_LO    1           // Use the Accumulator Low bits
‘define ALU_A_RF        2           // Use the output of the Register File
‘define ALU_A_ONE       3           // Select 1 as the input
‘define ALU_A_TWO       4           // Select 2 as the input
‘define ALU_A_ACC_HI    5           // Use the Accumulator High bits
‘define ALU_A_ONES      6           // Select -1 as the input (all ones)
‘define ALU_A_ZERO      7

‘define alu_a_dst       8’b0000_0001
‘define alu_a_rf        8’b0000_0010
‘define alu_a_acc_lo    8’b0000_0100
‘define alu_a_acc_hi    8’b0000_1000
‘define alu_a_zero      8’b0001_0000
‘define alu_a_one       8’b0011_0000
‘define alu_a_two       8’b0101_0000
‘define alu_a_ones      8’b1111_0000

/* ALU Source B Select Defines */
‘define ALU_B_SEL_LEN   8           // Wdith of the ALU Source B Select Bus
‘define ALU_B_SRC_HI    0           // Use SRC High bits
‘define ALU_B_SRC_LO    1           // Use SRC Low bits
‘define ALU_B_ACC_LO    2           // Use the Accumulator Low Bits
‘define ALU_B_RF        3           // Use output of the Register File as input
‘define ALU_B_DST       4           // Use DST
‘define ALU_B_SOB       5           // Use the Memory Read Data
‘define ALU_B_BRANCH    6           // Use the sign extended branch offset
‘define ALU_B_ZERO      7

‘define alu_b_src_lo    8’b0000_0001
‘define alu_b_src_hi    8’b0000_0010
‘define alu_b_dst       8’b0000_0100
‘define alu_b_rf        8’b0000_1000
‘define alu_b_acc_lo    8’b0001_0000
‘define alu_b_sob       8’b0010_0000
‘define alu_b_branch    8’b1100_0000
‘define alu_b_zero      8’b0100_0000

/* SRC Low Select Defines */
‘define SRC_LO_SEL_LEN  6           // Width of SRC Low Select Bus
‘define SRC_LO_RF       0           // SRC Low gets RF
‘define SRC_LO_MEM      1           // SRC Low gets Memory Read Data
‘define SRC_LO_LS       2           // SRC Low gets SRC Low left shifted by 1
‘define SRC_LO_WD       3           // SRC Low gets Write Data Bus
```

```
`define SRC_LO_SAME     4
`define SRC_LO_RS        5


`define src_lo_rf       6'b000001
`define src_lo_mem      6'b000010
`define src_lo_ls       6'b000100
`define src_lo_wd       6'b001000
`define src_lo_same     6'b010000
`define src_lo_rs        6'b100000

/* SRC High Select Defines */
`define SRC_HI_SEL_LEN  5           // Width of SRC High Select Bus
`define SRC_HI_RF       0           // SRC High gets RF
`define SRC_HI_WD       1           // SRC High gets Memory Read Data
`define SRC_HI_LS       2           // SRC High gets SRC High left shifted by 1
`define SRC_HI_SAME     3
`define SRC_HI_RS        4


`define src_hi_rf       5'b00001
`define src_hi_wd       5'b00010
`define src_hi_ls       5'b00100
`define src_hi_same     5'b01000
`define src_hi_rs        5'b10000

/* DST Select Defines */
`define DST_SEL_LEN     6           // Width of DST Select Bus
`define DST_RF          0           // DST gets RF
`define DST_MEM         1           // DST gets Memory Read Data
`define DST_RS          2           // DST gets DST right shifted by 1
`define DST_WD          3           // DST gets Write Data Bus
`define DST_SAME        4
`define DST_SHAMT        5


`define dst_rf          6'b000001
`define dst_mem         6'b000010
`define dst_rs          6'b000100
`define dst_wd          6'b001000
`define dst_same        6'b010000
`define dst_shamt        6'b100000

/* Write Data Select Defines */
`define WD_SEL_LEN      5           // Width of WD Select Bus
`define WD_ACC_LO       0           // WD is Accumulator Low bits
`define WD_PSW          1           // WD is the PSW
`define WD_MEM          2           // WD is the Memory read data
`define WD_MOVB         3
`define WD_ALU_OUT      4
                                    // Default is the ALU OUT


`define wd_acc_lo       5'b00001
`define wd_psw          5'b00010
`define wd_mem          5'b00100
```

```
‘define wd_movb         5’b01000
‘define wd_alu_out      5’b10000

/* Memory Address Select Defines */
‘define MEM_ADDR_SEL_LEN    5       // Width of WD Select Bus
‘define MEM_ADDR_ALU_OUT    0       // Memory Address gets ALU Out
‘define MEM_ADDR_ACC_LO     1       // Memory Address gets Accumulator Low bits
‘define MEM_ADDR_MEM        2       // Memory Address gets the Memory Read Data
‘define MEM_ADDR_ALU_B      3       // Memory Address gest the Register File Read Data
‘define MEM_ADDR_SAME       4


‘define mem_addr_acc_lo         5’b00001
‘define mem_addr_mem            5’b00010
‘define mem_addr_alu_b          5’b00100
‘define mem_addr_alu_out        5’b01000
‘define mem_addr_same           5’b10000

/* Accumulator Low Select Defines */
‘define ACC_LO_SEL_LEN  4           // Width of the Accumulator Low Select Bus
‘define ACC_LO_ALU_OUT  0           // Accumulator Low gets ALU Out
‘define ACC_LO_TRAP     1           //Accumulator Low gets a trap vector
‘define ACC_LO_MEM      2           //Accumulator Low gets a memory
‘define ACC_LO_SAME     3

‘define acc_lo_alu_out  4’b0001
‘define acc_lo_trap     4’b0010
‘define acc_lo_mem      4’b0100
‘define acc_lo_same     4’b1000
‘define acc_lo_sft_out  4’b0000

/* Accumulator High Select Defines */
‘define ACC_HI_SEL_LEN  4           // Width of the Accumulator High Select Bus
‘define ACC_HI_ALU_OUT  0           // Accumulator High gets ALU Out
‘define ACC_HI_DIV      1           // Acc Hi gets 16’h8000
‘define ACC_HI_RS       2           // Acc Hi >> 1
‘define ACC_HI_SAME     3


‘define acc_hi_alu_out  4’b0001
‘define acc_hi_div      4’b0010
‘define acc_hi_rs       4’b0100
‘define acc_hi_same     4’b1000

/* Register File Address Select Defines */
‘define RF_ADDR_SEL_LEN 6           // Width of Register File Address Select Bus
‘define RF_ADDR_SP      0           // RF Address is the Stack Pointer
‘define RF_ADDR_R5      1           // RF Address is R5
‘define RF_ADDR_DST_REG 2           // RF Address is the Instruction DST Reg
‘define RF_ADDR_SRC_REG 3           // RF Address is the Instruction SRC Reg
‘define RF_ADDR_RV1     4           // RF Address is SRC Reg + 1
‘define RF_ADDR_PC      5
                                    // DEFAULT IS THE PC!!!


‘define rf_addr_sp      6’b00_0001
```

```
`define rf_addr_r5      6'b00_0010
`define rf_addr_dst_reg 6'b00_0100
`define rf_addr_src_reg 6'b00_1000
`define rf_addr_rv1     6'b01_0000
`define rf_addr_pc      6'b10_0000

/* Shift Amount Select Defines */
`define SHAMT_SEL_LEN   5           // Width of the Shift Amount Select Bus
`define SHAMT_SRC       0           // Shift Amount is the SRC part of the Instruction
`define SHAMT_L1        1           // Shift Amount is Left by 1 or +1
`define SHAMT_R1        2           // Shift Amount is Right by 1 or -1
`define SHAMT_8         3           // Used for SWAB
`define SHAMT_ZERO      4
                                    // Default is Zero
`define shamt_src       5'b00001
`define shamt_L1        5'b00010
`define shamt_R1        5'b00100
`define shamt_8         5'b01000
`define shamt_zero      5'b10000

`define SFT_SRC_HI_LEN  3
`define SHIFT_HI_LO     0
`define SHIFT_HI_HI     1
`define SHIFT_HI_DST    2

`define shift_hi_lo     3'b001
`define shift_hi_hi     3'b010
`define shift_hi_dst    3'b100

`define SFT_SRC_LO_LEN  4
`define SHIFT_LO_LO     0
`define SHIFT_LO_HI     1
`define SHIFT_LO_BYTE   2
`define SHIFT_LO_DST    3

`define shift_lo_lo     4'b0001
`define shift_lo_hi     4'b0010
`define shift_lo_byte   4'b0100
`define shift_lo_dst    4'b1000

/* PSW Select Defines */
`define PSW_SEL_LEN     1           // Widht of PSW select bus
`define PSW_MEM         0           // PSW gets the current WD

`define psw_same        1'b0
`define psw_mem         1'b1

/* Condition Code Defines */
`define C_SEL_LEN       7           // Width of C code select bus
`define C_ALU           0           // Take C as Carry out from ALU
`define C_ALU_NOT       1           // Take C as ~Carry Out
`define C_SFT           2           // Take C from the shifter
`define C_SET           3           // Set C
`define C_CLEAR         4           // Clear C
`define C_NOT_Z         5           // C = 0 if result == 0, 1 otherwise
`define C_SAME          6
```

```
`define c_alu           7'b000_0001
`define c_alu_not       7'b000_0010
`define c_sft           7'b000_0100
`define c_set           7'b000_1000
`define c_clear         7'b001_0000
`define c_not_z         7'b010_0000
`define c_same          7'b100_0000

`define V_SEL_LEN       11          // Width of V code select bus
`define V_SINGLE        0           // V from Single Ops
`define V_DOUBLE_ADD    1           // V from Double Ops
`define V_SFT           2           // V from Shifter
`define V_CLEAR         3           // V Cleared
`define V_SET           4           // V Set
`define V_NEG           5           // V for negate, or no sign change
`define V_DOUBLE_SUB    6           // V for subtract (dst - src)
`define V_DOUBLE_CMP    7           // V for Compare (src - dst)
`define V_SINGLE_POS    8           // V for addc
`define V_DST_MIN       9           // V for subc
`define V_SAME          10

`define v_single        11'b000_0000_0001
`define v_double_add    11'b000_0000_0010
`define v_sft           11'b000_0000_0100
`define v_clear         11'b000_0000_1000
`define v_set           11'b000_0001_0000
`define v_neg           11'b000_0010_0000
`define v_double_sub    11'b000_0100_0000
`define v_double_cmp    11'b000_1000_0000
`define v_single_pos    11'b001_0000_0000
`define v_single_neg    11'b010_0000_0000
`define v_same          11'b100_0000_0000

`define Z_SEL_LEN       4           // Width of Z code select bus
`define Z_ZERO          0           // Check if out is Zero
`define Z_SET           1           // Set Z
`define Z_CLEAR         2           // Clear Z
`define Z_SAME          3

`define z_zero          4'b0001
`define z_set           4'b0010
`define z_clear         4'b0100
`define z_same          4'b1000

`define N_SEL_LEN       4           // Width of N code select bus
`define N_NEG           0           // Check if out is negative
`define N_CLEAR         1           // Clear N
`define N_SET           2           // Set N
`define N_SAME          3

`define n_neg           4'b0001
`define n_clear         4'b0010
`define n_set           4'b0100
`define n_same          4'b1000
```

```
`define IO_VECTOR       16'h0010    // Octal 20 = 0x10 = 16 decimal
`define TRAP_VECTOR     16'h001c    // Octal 34 = 0x1c = 28 decimal
`define EMT_VECTOR      16'h0018    // Octal 30 = 0x18 = 24 decimal
`define BPT_VECTOR      16'h000c    // Octal 14 = 0xc = 12 decimal
`define ILLEGAL_VECTOR  16'h0004    // Octal 4 = 0x4 = 4 decimal

`define IO_CONST        3'b100
`define TRAP_CONST      3'b111
`define EMT_CONST       3'b110
`define BPT_CONST       3'b011
`define ILLEGAL_CONST   3'b001


`define TRAP_SEL_LEN    6           // Width of Trap Vector Select bus
`define TRAP_IO         0           // I/O Trap Vector
`define TRAP_TRAP       1           // Trap Trap Vector
`define TRAP_EMT        2           // EMT Trap Vector
`define TRAP_BPT        3           // Break Point Trap
`define TRAP_MEM_DATA   4           // Trap to Vector on Mem Data
`define TRAP_BROKEN     5

`define trap_io         6'b00_0001
`define trap_trap       6'b00_0010
`define trap_emt        6'b00_0100
`define trap_bpt        6'b00_1000
`define trap_mem_data   6'b01_0000
`define trap_broken     6'b10_0000
```

A.16 PSW_Block.v

```
`timescale 1ns / 100ps
`include "pdp11.h"
module PSW_Block(
    inout       [15:0]  WDW, MRD,
    input       [3:0]   FlagsNext,
    input       [2:0]   PriorityNext,
                        PSWSel, PSWSel_b,
    input               WD_PSW, WD_PSW_b, MemPSW, MemPSW_b,
                        sdi, scan, scan_b, clk, clkb, clk_b, clkb_b, reset,
    output      [3:0]   Flags,
    output      [2:0]   Priority,
    output      [1:0]   CurPrivilege, PrevPrivilege,
    output              TraceTrap
    );

    wire        [15:0]  PSW, PSWNext;

    // Select what goes into the PSW Next
    Mux3_dp #(.WIDTH(16)) PSW_Mux(
        .d0(MRD[15:0]),
        .d1({PSW[15:8],PriorityNext[2:0],PSW[4], FlagsNext[3:0]}),
        .d2(WDW[15:0]),
        .sel(PSWSel[2:0]), .sel_b(PSWSel_b[2:0]),
        .q(PSWNext)
```

```
    );
    scanflop_r #(.WIDTH(16)) PSW_SR(
        .clk(clk), .clk_b(clk_b), .clkb(clkb), .clkb_b(clkb_b),
        .reset(reset),
        .scan(scan), .scan_b(scan_b),
        .d(PSWNext[15:0]),
        .sdi(sdi), .q(PSW[15:0])
    );

    Tri_dp #(.WIDTH(16)) WDW_Tri(.d(PSW[15:0]), .en(WD_PSW), .en_b(WD_PSW_b), .q(WDW[15:0]));
    Tri_dp #(.WIDTH(16)) MEM_Tri(.d(PSW[15:0]), .en(MemPSW), .en_b(MemPSW_b), .q(MRD[15:0]));

    // 3 : N, 2 : Z, 1 : V, 0 : C
    assign Flags = PSW[3:0];
    assign TraceTrap = PSW[4];
    assign Priority = PSW[7:5];
    assign CurPrivilege = PSW[15:14];
    assign PrevPrivilege = PSW[13:12];
endmodule
```

A.17 Reg_Block.v

```
‘timescale 1ns / 100ps
module Reg_Block(
    input   [8:0]   RegSelect, RegSelect_b,
                    HiEnable,  HiEnable_b,
                    LowEnable, LowEnable_b,
    input           Byte_b,
    input           clk, clkb, clk_b, clkb_b, scan, scan_b, sdi,
    input   [15:0]  WriteData,
    output  [15:0]  LatchOut
    );
    wire    [15:0]  ReadOut;
/*    RegFile RF(
        .RegSelect(RegSelect), .RegSelect_b(RegSelect_b),
        .HiEnable(HiEnable), .HiEnable_b(HiEnable_b),
        .LowEnable(LowEnable), .LowEnable_b(LowEnable_b),
        .Byte_b(Byte_b), .WriteData(WriteData),
        .ReadOut(ReadOut)
    );*/
    RegFile RF(
        ReadOut, Byte_b, HiEnable, HiEnable_b, LowEnable, LowEnable_b, RegSelect, RegSelect_b, WriteData
    );
    scanlatch #(.WIDTH(16)) RF_SL(
        .clk(clk), .clkb(clkb), .clk_b(clk_b), .clkb_b(clkb_b),
        .scan(scan), .scan_b(scan_b), .sdi(sdi),
        .d(ReadOut), .q(LatchOut)
    );
endmodule
```

A.18 RegFile.v

```
‘timescale 1ns / 100ps
module RegFile(
    output  [15:0]  readout,
    input           Byte_b,
```

```
input   [8:0]   HiEnable,  HiEnable_b,
                LowEnable, LowEnable_b,
                RegSelect, RegSelect_b,
input   [15:0]  WriteData
);
wire    [15:0]  RegOut;
DRegister r0  (
    .RegSelect(RegSelect[0]),
    .HiEnable(HiEnable[0]),
    .LowEnable(LowEnable[0]),
    .WriteData(WriteData),
    .ReadOut(RegOut)
);
DRegister r1  (
    .RegSelect(RegSelect[1]),
    .HiEnable(HiEnable[1]), .LowEnable(LowEnable[1]),
    .WriteData(WriteData),
    .ReadOut(RegOut)
);
DRegister r2  (
    .RegSelect(RegSelect[2]),
    .HiEnable(HiEnable[2]), .LowEnable(LowEnable[2]),
    .WriteData(WriteData),
    .ReadOut(RegOut)
);
DRegister r3  (
    .RegSelect(RegSelect[3]),
    .HiEnable(HiEnable[3]), .LowEnable(LowEnable[3]),
    .WriteData(WriteData),
    .ReadOut(RegOut)
);
DRegister r4  (
    .RegSelect(RegSelect[4]),
    .HiEnable(HiEnable[4]), .LowEnable(LowEnable[4]),
    .WriteData(WriteData),
    .ReadOut(RegOut)
);
DRegister r5  (
    .RegSelect(RegSelect[5]),
    .HiEnable(HiEnable[5]), .LowEnable(LowEnable[5]),
    .WriteData(WriteData),
    .ReadOut(RegOut)
);
DRegister r6_u(
    .RegSelect(RegSelect[6]),
    .HiEnable(HiEnable[6]), .LowEnable(LowEnable[6]),
    .WriteData(WriteData),
    .ReadOut(RegOut)
);
DRegister r6_k(
    .RegSelect(RegSelect[7]),
    .HiEnable(HiEnable[7]), .LowEnable(LowEnable[7]),
    .WriteData(WriteData),
    .ReadOut(RegOut)
);
DRegister r7  (
```

```
        .RegSelect(RegSelect[8]),
        .HiEnable(HiEnable[8]), .LowEnable(LowEnable[8]),
        .WriteData(WriteData),
        .ReadOut(RegOut)
    );
    assign readout[15:0] = {RegOut[15:8] & {8{Byte_b}}, RegOut[7:0]};
endmodule
```

A.19 RegPSW_Block.v

```
‘timescale 1ns / 100ps
‘include "pdp11.h"
module RegPSW_Block(
    inout   [15:0]  WDW,
    input   [15:0]  MDI,
    input   [8:0]   RegSelect, RegSelect_b,
                    HiEnable,  HiEnable_b,
                    LowEnable, LowEnable_b,
    input   [3:0]   FlagsNext,
    input   [2:0]   PriorityNext, PSWSel, PSWSel_b,
    input           MemReadPSW , MemReadPSW_b ,
                    WD_MEM, WD_MEM_b,
                    WD_PSW, WD_PSW_b,
                    sdi, scan, scan_b, clk, clk_b, clkb, clkb_b,
                    byte_b,
                    WDSel, WDSel_b, reset,
    output  [15:0]  MRD, RF,
    output  [7:0]   WD,
    output  [3:0]   Flags,
    output  [2:0]   Priority,
    output  [1:0]   CurPrivilege, PrevPrivilege,
    output          TraceTrap
    );

    wire    [15:0]  MA;
    Tri_dp #(.WIDTH(16)) MDI_tri(
        .d(MDI),
        .en(MemReadPSW_b), .en_b(MemReadPSW),
        .q(MRD)
    );
    Tri_dp #(.WIDTH(16)) WDW_tri(
        .d(MRD),
        .en(WD_MEM), .en_b(WD_MEM_b),
        .q(WDW)
    );
    PSW_Block PSWB(
        .WDW(WDW), .MRD(MRD),
        .PriorityNext(PriorityNext),
        .FlagsNext(FlagsNext),
        .PSWSel(PSWSel), .PSWSel_b(PSWSel_b),
        .WD_PSW(WD_PSW), .WD_PSW_b(WD_PSW_b),
        .MemPSW(MemReadPSW), .MemPSW_b(MemReadPSW_b),
        .sdi(sdi), .scan(scan), .scan_b(scan_b),
        .clk(clk), .clk_b(clk_b), .clkb(clkb), .clkb_b(clkb_b),
        .Flags(Flags), .Priority(Priority), .reset(reset),
        .CurPrivilege(CurPrivilege), .PrevPrivilege(PrevPrivilege),
```

```
        .TraceTrap(TraceTrap)
    );
    Mux2_OneHot #(.WIDTH(8)) WDMux(
        .d0(WDW[7:0]),
        .d1(WDW[15:8]),
        .sel0(WDSel),
        .sel1(WDSel_b),
        .q(WD)
    );

    Reg_Block RFB(
        .RegSelect(RegSelect), .RegSelect_b(RegSelect_b),
        .HiEnable(HiEnable), .HiEnable_b(HiEnable_b),
        .LowEnable(LowEnable), .LowEnable_b(LowEnable_b),
        .Byte_b(byte_b),
        .clk(clk), .clk_b(clk_b), .scan(scan), .scan_b(scan_b), .sdi(Flags[0]),
        .clkb(clkb), .clkb_b(clkb_b),
        .WriteData(WDW),
        .LatchOut(RF)
    );
endmodule
```

A.20 scanflop.v

```
‘timescale 1ns / 100ps
module scanflop_r #(parameter WIDTH=16) (
    input clk, clkb,
    input clk_b, clkb_b,
    input scan, scan_b,
    input reset,
    input sdi,
    input [WIDTH-1:0] d,
    output reg [WIDTH-1:0] q
    );
    reg [WIDTH-1:0] mid;
    always @ (*)
        begin
        if(reset)     mid[WIDTH-1:0] <= {WIDTH{1'b0}};
        else if(clkb)mid[WIDTH-1:0] <= (scan & ~scan_b) ? {sdi, q[WIDTH-1:1]} : d[WIDTH-1:0];
        else          mid[WIDTH-1:0] <= mid[WIDTH-1:0];

        if(clk)      q[WIDTH-1:0]   <= mid[WIDTH-1:0];
        else         q[WIDTH-1:0]   <= q[WIDTH-1:0];
    end
endmodule

module scanlatch #(parameter WIDTH=16) (
    input clk, clk_b, clkb, clkb_b,
    input scan, scan_b ,
    input sdi,
    input [WIDTH-1:0] d,
    output [WIDTH-1:0] q
    );
    wire    [WIDTH-1:0] mid;
    // getting lazier
    assign mid = clkb ? {sdi, q[WIDTH-1:1]} : mid[WIDTH-1:0];
```

```
    assign q   = clk ? (scan ? mid : d) : q;
endmodule
```

A.21 SD_Block.v

```
‘timescale 1ns / 100ps
module SD_Block(
    input       [15:0]  WDW, MRD, RF,
    input       [5:0]   DST_Sel, DST_Sel_b,
                        SRC_Lo_Sel, SRC_Lo_Sel_b,
                        Instruction,
    input       [4:0]   SRC_Hi_Sel, SRC_Hi_Sel_b,
    input       [3:0]   ALUBSel, ALUBSel_b,
    input       [2:0]   SFTConst,
    input       [1:0]   ALUASel, ALUASel_b,
    input               Lo_Lo_Bit, Lo_Hi_Bit, Lo_Mid_Bit,
    input               sdi, scan, scan_b, clk, clk_b, clkb, clkb_b, reset,
                        SFTAmt, SFTAmt_b,
    output      [15:0]  ALUA, ALUB,
    output              SRC_Lo_15, SRC_Lo_7, SRC_Lo_0, SRC_Hi_15, SRC_Hi_0,
                        L7Z, L15Z, DST_0, DST_7, SRC_8, DST_15
    );
    wire        [5:0]   Shamt;
    SRC_Lo_Block SLB(
        .WDW(WDW), .MRD(MRD), .RF(RF),
        .sdi(sdi), .scan(scan), .scan_b(scan_b), .clk(clk), .clk_b(clk_b), .reset(reset),
        .clkb(clkb), .clkb_b(clkb_b),
        .Lo_Lo_Bit(Lo_Lo_Bit), .Lo_Hi_Bit(Lo_Hi_Bit), .Lo_Mid_Bit(Lo_Mid_Bit),
        .SRC_Lo_Sel(SRC_Lo_Sel), .SRC_Lo_Sel_b(SRC_Lo_Sel_b),
        .ALUBSel(ALUBSel[0])  , .ALUBSel_b(ALUBSel_b[0])  ,

        .ALUB(ALUB),
        .SRC_Lo_15(SRC_Lo_15), .SRC_Lo_7(SRC_Lo_7), .SRC_Lo_8(SRC_8), .SRC_Lo_0(SRC_Lo_0)
    );

    SRC_Hi_Block SHB(
        .WDW(WDW), .RF(RF),
        .sdi(SRC_Lo_0), .scan(scan), .scan_b(scan_b), .clk(clk), .clk_b(clk_b), .reset(reset),
        .clkb(clkb), .clkb_b(clkb_b),
        .SRC_Lo_15(SRC_Lo_15),
        .SRC_Hi_Sel(SRC_Hi_Sel), .SRC_Hi_Sel_b(SRC_Hi_Sel_b),
        .ALUBSel(ALUBSel[1])  , .ALUBSel_b(ALUBSel_b[1])  ,

        .ALUB(ALUB), .SRC_Hi_15(SRC_Hi_15), .SRC_Hi_0(SRC_Hi_0)
    );

    DST_Block DB(
        .WDW(WDW), .MRD(MRD), .RF(RF),
        .sdi(SRC_Hi_0), .scan(scan), .scan_b(scan_b), .clk(clk), .clk_b(clk_b), .reset(reset),
        .clkb(clkb), .clkb_b(clkb_b),
        .DST_Sel(DST_Sel), .DST_Sel_b(DST_Sel_b),
        .ALUASel(ALUASel[0])  , .ALUASel_b(ALUASel_b[0])  ,
        .ALUBSel(ALUBSel[2])  , .ALUBSel_b(ALUBSel_b[2])  ,
        .Shamt(Shamt),

        .ALUA(ALUA), .ALUB(ALUB),
```

```
        .L7Z(L7Z), .L15Z(L15Z),
        .DST_15(DST_15), .DST_7(DST_7), .DST_0(DST_0)
    );
    Mux2_OneHot #(.WIDTH(6)) SHAMTMux(
        .d0(Instruction[5:0]),
        .d1({{2{SFTConst[0]}}, SFTConst[1], {2{SFTConst[0]}}, SFTConst[2]}),
        .sel0(SFTAmt), .sel1(SFTAmt_b),
        .q(Shamt)
    );

    Tri_dp #(.WIDTH(16)) ALUATri(
        .d(RF),
        .en(ALUASel[1]), .en_b(ALUASel_b[1]),
        .q(ALUA)
    );
    Tri_dp #(.WIDTH(16)) ALUBTri(
        .d(RF),
        .en(ALUBSel[3]), .en_b(ALUBSel_b[3]),
        .q(ALUB)
    );
endmodule
```

A.22 SRC_Hi_Block.v

```
‘timescale 1ns / 100ps
‘include "pdp11.h"
module SRC_Hi_Block(
    input       [15:0]  WDW, RF,
    input               sdi, scan, scan_b, clk, clk_b, clkb, clkb_b, reset,
                        SRC_Lo_15,
    input       [4:0]   SRC_Hi_Sel, SRC_Hi_Sel_b,
    input               ALUBSel, ALUBSel_b,

    output      [15:0]  ALUB,
    output              SRC_Hi_15, SRC_Hi_0);

    wire        [15:0]  SRC_Hi, SRC_Hi_Next;
    assign              SRC_Hi_15 = SRC_Hi[15];
    assign              SRC_Hi_0  = SRC_Hi[0];
    // Select the next SRC_Hi value
    Mux5_dp #(.WIDTH(16)) SRC_Hi_Mux(
        .d0(RF[15:0] ),
        .d1(WDW[15:0]),
        .d2({SRC_Hi[14:0], SRC_Lo_15}),
        .d3(SRC_Hi[15:0]),
        .d4({SRC_Hi[15],SRC_Hi[15:1]}),
        .sel(SRC_Hi_Sel), .sel_b(SRC_Hi_Sel_b),
        .q(SRC_Hi_Next[15:0])
    );

    scanflop_r #(.WIDTH(16)) SRC_Hi_SR(
        .d(SRC_Hi_Next[15:0]),
        .reset(reset),
        .clk(clk), .clk_b(clk_b),
        .clkb(clkb), .clkb_b(clkb_b),
        .sdi(sdi),
```

```
        .scan(scan), .scan_b(scan_b),
        .q(SRC_Hi[15:0])
    );

    Tri_dp #(.WIDTH(16)) ALUB_Tri (.d(SRC_Hi[15:0]), .en(ALUBSel ), .en_b(ALUBSel_b ), .q(ALUB[15:0]) );
endmodule
```

A.23 SRC_Lo_Block.v

```
‘timescale 1ns / 100ps
‘include "pdp11.h"
module SRC_Lo_Block(
    input       [15:0]  WDW, MRD, RF,
    input               sdi, scan, scan_b, clk, clk_b, clkb, clkb_b, reset,
    input               Lo_Hi_Bit, Lo_Lo_Bit, Lo_Mid_Bit,
    input       [5:0]   SRC_Lo_Sel, SRC_Lo_Sel_b,
    input               ALUBSel, ALUBSel_b,

    output      [15:0]  ALUB,
    output              SRC_Lo_15, SRC_Lo_8, SRC_Lo_7, SRC_Lo_0
    );

    wire        [15:0] SRC_Lo, SRC_Lo_Next;
    assign             SRC_Lo_15 =  SRC_Lo[15];
    assign             SRC_Lo_7  =  SRC_Lo[7] ;
    assign             SRC_Lo_8  =  SRC_Lo[8] ;
    assign             SRC_Lo_0  =  SRC_Lo[0] ;
    // Select the Next SRC_Lo value
    Mux6_dp #(.WIDTH(16)) SRC_Lo_Mux(
        .d0(RF[15:0]),
        .d1(MRD[15:0]),
        .d2({SRC_Lo[14:0],Lo_Lo_Bit}),
        .d3(WDW[15:0]),
        .d4(SRC_Lo[15:0]),
        .d5({Lo_Hi_Bit,SRC_Lo[15:9],Lo_Mid_Bit,SRC_Lo[7:1]}),
        .sel(SRC_Lo_Sel), .sel_b(SRC_Lo_Sel_b),
        .q(SRC_Lo_Next[15:0])
    );

    scanflop_r #(.WIDTH(16)) SRC_Lo_SR(
        .d(SRC_Lo_Next[15:0]),
        .reset(reset),
        .clk(clk), .clk_b(clk_b),
        .clkb(clkb), .clkb_b(clkb_b),
        .sdi(sdi),
        .scan(scan), .scan_b(scan_b),
        .q(SRC_Lo[15:0])
    );
    Tri_dp #(.WIDTH(16)) ALUB_Tri (.d(SRC_Lo[15:0]), .en(ALUBSel ), .en_b(ALUBSel_b ), .q(ALUB[15:0]) );
endmodule
```

A.24 States.h

```
/* Defines for the Controller module */

‘define FETCH0          8’h00
```

```
`define FETCH1          8'h01
`define FETCH2          8'h02
`define FETCH3          8'h03

`define SINGLE0         8'h04
`define DOUBLE0         8'h05
`define DOUBLE1         8'h06

`define DST_REG0        8'h0a
`define DST_DREG0       8'h0b
`define DST_DREG1       8'h0c
`define DST_INC0        8'h0d
`define DST_INC1        8'h0e
`define DST_DINC0       8'h0f
`define DST_DINC1       8'h10
`define DST_DINC2       8'h11
`define DST_DEC0        8'h12
`define DST_DEC1        8'h13
`define DST_DDEC0       8'h14
`define DST_DDEC1       8'h15
`define DST_DDEC2       8'h16
`define DST_IDX0        8'h17
`define DST_IDX1        8'h18
`define DST_IDX2        8'h19
`define DST_IDX3        8'h1a
`define DST_DIDX0       8'h1b
`define DST_DIDX1       8'h1c
`define DST_DIDX2       8'h1d
`define DST_DIDX3       8'h1e
`define DST_DIDX4       8'h1f

`define SRC_REG0        8'h20
`define SRC_DREG0       8'h21
`define SRC_DREG1       8'h22
`define SRC_INC0        8'h23
`define SRC_INC1        8'h24
`define SRC_DINC0       8'h25
`define SRC_DINC1       8'h26
`define SRC_DINC2       8'h27
`define SRC_DEC0        8'h28
`define SRC_DEC1        8'h29
`define SRC_DDEC0       8'h2a
`define SRC_DDEC1       8'h2b
`define SRC_DDEC2       8'h2c
`define SRC_IDX0        8'h2d
`define SRC_IDX1        8'h2e
`define SRC_IDX2        8'h2f
`define SRC_IDX3        8'h30
`define SRC_DIDX0       8'h31
`define SRC_DIDX1       8'h32
`define SRC_DIDX2       8'h33
`define SRC_DIDX3       8'h34
`define SRC_DIDX4       8'h35

//`define SWAB0           8'h36
```

```
‘define EXECUTE0        8’h38

‘define SINGLE_EX0      8’h39

‘define DOUBLE_EX0      8’h3b

‘define BRANCH0         8’h3d
‘define BRANCH_EX0      8’h3e

‘define SPECIAL_BRANCH0 8’h3f

‘define MARK0           8’h40
‘define MARK1           8’h41
‘define MARK2           8’h42
‘define MARK3           8’h43
‘define MARK4           8’h44
‘define MARK5           8’h45
‘define MARK6           8’h46

‘define MFPI0           8’h48
‘define MFPI1            8’h49
‘define MFPI2            8’h4a
‘define MFPI3            8’h4b
‘define MFPI4            8’h4c

‘define MTPI0           8’h50
‘define MTPI1            8’h51
‘define MTPI2            8’h52
‘define MTPI3            8’h53
‘define MTPI4            8’h54

‘define MUL0            8’h57
‘define MUL1            8’h58
‘define MUL2            8’h59
‘define MUL3            8’h5a
‘define MUL4            8’h5b
‘define MUL5            8’h5c
‘define MUL6            8’h5d
‘define MUL7            8’h5e
‘define MUL8            8’h5f
‘define MUL9            8’h60
‘define MUL10           8’h61
‘define MUL11           8’h62
‘define MUL12           8’h63
‘define ASHC0           8’h64
‘define ASHC1           8’h65
‘define ASHC2           8’h66
‘define ASHC3           8’h67
‘define ASHC4           8’h68
‘define ASHC5           8’h69
‘define ASHC6            8’h6a
‘define SOB0            8’h6b
‘define SOB1            8’h6c
‘define SOB2            8’h6d
‘define SOB3            8’h6e
```

```
‘define SCC0            8’h6f

‘define JMP0            8’h70
‘define JMP1            8’h71
//‘define BPT0            8’h7e
//‘define IOT0            8’h7d
‘define TRAP0           8’h72
‘define TRAP1           8’h73
‘define TRAP2           8’h74
‘define TRAP3           8’h75
‘define TRAP4           8’h76
‘define TRAP5           8’h77
‘define TRAP6           8’h78
//‘define TRAP7           8’h79
//‘define TRAP8           8’h7a
//‘define TRAP9           8’h7b
//‘define EMT0            8’h7c

‘define JSR0            8’h79
‘define JSR1            8’h7a
‘define JSR2            8’h7b
‘define JSR3            8’h7c
‘define JSR4            8’h7d
‘define JSR5            8’h7e
‘define JSR6            8’h7f
‘define RTS0            8’h80
‘define RTS1            8’h81
‘define RTS2            8’h82
‘define RTS3            8’h83
‘define RTS4            8’h84
//‘define RTI0            8’h85
‘define RTT0            8’h87
‘define RTT1            8’h88
‘define RTT2            8’h89
‘define SPL0            8’h8b

‘define TRAP7           8’h8c
‘define TRAP8           8’h8d
‘define TRAP9           8’h8e
‘define TRAP10          8’h8f
‘define TRAP11          8’h90

‘define MFPT0           8’h91

‘define SWAB0            8’h92
‘define SWAB1            8’h93
‘define SWAB2            8’h94
‘define SWAB3            8’h95
‘define SWAB4            8’h96

‘define WRITE_BACK0     8’ha0
‘define WRITE_BACK1     8’ha1
‘define REG_WRITE_BACK0 8’ha2

‘define DIV0            8’hb0
‘define DIV1            8’hb1
```

```
‘define DIV2            8’hb2
‘define DIV3            8’hb3
‘define DIV4            8’hb4
‘define DIV5            8’hb5
‘define DIV6            8’hb6
‘define DIV7            8’hb7
‘define DIV8            8’hb8
‘define DIV9            8’hb9
‘define DIV10           8’hba
‘define DIV11           8’hbb
‘define DIV12           8’hbc
‘define DIV13           8’hbd
‘define DIV14           8’hbe
‘define DIV15           8’hbf
‘define DIV16           8’hc0

‘define ASH0            8’hc2
‘define ASH1            8’hc3
‘define ASH2            8’hc4
‘define ASH3            8’hc5

‘define ROTR0            8’hc7
‘define ROTL0            8’hc8
‘define ASR0            8’hc9
‘define ASL0            8’hca
‘define SINGLE_SFT0        8’hcb

‘define HALT0           8’hf0
‘define WAIT0           8’hf1
‘define BROKEN          8’hff
```

A.25 SuperController.v

```
/* Controller.v
 *
 * Created By Andrew Carter
 *
 * Its a Controller but better
 *
 */
‘timescale 1ns / 100ps
‘include "States.h"
‘include "pdp11.h"
‘include "Controller.h"

module SuperController(
    input   [15:0]  Instruction,
    input   [3:0]   Flags,
    input   [2:0]   Priority,
    input   [1:0]   CurPrivilege, PrevPrivilege,
    input           WDW_Zero, WDW_15, WDW_Lo_Zero, WDW_7,
                    SRC_Lo_15, SRC_Lo_7, SRC_Lo_0, SRC_Hi_15, SRC_Hi_0,
                    Acc_Hi_0, MemPSW,
                    L7Z, L15Z, DST_0, DST_7, SRC_8, DST_15,
                    CWALUOut, CBALUOut,
                    TraceTrap, sdi,
```

```
                clk, clkb,
                Interrupt,
                reset, scan,
 output reg [8:0]RegSelect,
 output  [8:0]   RegSelect_b,
                 HiEnable, HiEnable_b,
                 LowEnable, LowEnable_b,
 output  [7:0]   ALUASel, ALUASel_b,
                 ALUBSel, ALUBSel_b,
 output  [5:0]   DSTSel, DSTSel_b,
                 SRCLoSel, SRCLoSel_b,
 output  [4:0]   SRCHiSel, SRCHiSel_b,
                 WDWSel, WDWSel_b,
                 MANSel, MANSel_b,
 output reg[3:0] FlagsNext, ACCLoSel,
 output  [3:0]   ACCLoSel_b,
                 ACCHiSel, ACCHiSel_b,
 output reg[2:0] SFTConst, TrapVal,
 output  [2:0]   PriorityNext,
                 PSWSel, PSWSel_b,
                 ALUAConst,
 output reg      SFTAmt,
 output          MemReadPSW, MemReadPSW_b,
                 WDSel, WDSel_b,
                 Update, Update_b,
                 SFTAmt_b,
                 MemHigh, MANen, MANen_b,
                 sdo, scan_b, reset_b,
                 MemEn, MemWrite, Kernal,
                 ReadByte, ReadByte_b,
                 InterruptClear,
                 Lo_Lo_Bit,
                 Lo_Hi_Bit,
                 Lo_Mid_Bit,
                 // ALU Stuff
                 ALUinvertB, ALUinvertA, add_b,
                 ALUC_in, enALU, enALU_b, ALUenAND, ALUenAND_b,
                 ALUenOR, ALUenOR_b, ALUenCOMP, ALUenCOMP_b, clk_b, clkb_b
                    );
wire             CALUOut;
wire [7:0]       ALU_Ctrl;
wire             isByte, isByte_b;
assign CALUOut      = isByte ? CBALUOut : CWALUOut;
assign MemReadPSW   = MemPSW;
assign RegSelect_b  = ~RegSelect;
assign HiEnable_b   = ~HiEnable;
assign LowEnable_b  = ~LowEnable;
assign ALUASel_b    = ~ALUASel;
assign ALUBSel_b    = ~ALUBSel;
assign DSTSel_b     = ~DSTSel;
assign WDWSel_b     = ~WDWSel;
assign SRCLoSel_b   = ~SRCLoSel;
assign SRCHiSel_b   = ~SRCHiSel;
assign ACCLoSel_b   = ~ACCLoSel;
assign ACCHiSel_b   = ~ACCHiSel;
assign PSWSel_b     = ~PSWSel;
```

```
assign MANSel_b     = ~MANSel;
assign MemReadPSW_b = ~MemReadPSW;
assign isByte_b     = ~isByte;
assign WDSel_b      = ~WDSel;
assign Update_b     = ~Update;
assign SFTAmt_b     = ~SFTAmt;
assign MANen_b      = ~MANen;
assign scan_b       = ~scan;
assign reset_b      = ~reset;
assign clk_b        = ~clk;
assign clkb_b       = ~clkb;
wire [`STATE_LEN-1:0] State; // Mid_State;
wire [5:0]            State_Types;

wire [`STATE_LEN-1:0]   Next_State;
wire [`STATE_LEN-1+6:0] NS; // muxed Next_State
wire [5:0]              NST;


// We have made the buses larger to include instr_type, mult_div_sign,
// and remainder_sign in the 'state' register.
// These bits will be at the 'end' of 'state', in the order
// {instr_type,mult_div_sign,remainder_sign}
//
// resettable state register with initial value of FETCH0
mux2 #(`STATE_LEN + 6) resetmux({Next_State, NST}, {`FETCH0,6'b0}, reset, NS);
flop #(`STATE_LEN + 6) statereg(clk, clkb, NS, {State,State_Types});

 // **********************
 // *    controller      *
 // **********************
 wire    [`ALU_A_SEL_LEN-1:0]    ALU_A_Sel;
 wire    [`ACC_LO_SEL_LEN-1:0]   ACC_Lo_Sel;
 wire    [`RF_ADDR_SEL_LEN-1:0]  RF_Addr_Sel;
 wire    [`SHAMT_SEL_LEN-1:0]    Shamt_Sel;
 wire    [`C_SEL_LEN-1:0]        C_Sel;
 wire    [`V_SEL_LEN-1:0]        V_Sel;
 wire    [`Z_SEL_LEN-1:0]        Z_Sel;
 wire    [`N_SEL_LEN-1:0]        N_Sel;
 wire    [`TRAP_SEL_LEN-1:0]     Trap_Sel;
 wire                            RF_Enable, RF_Write_En, Mem_Space_Cur,
                                 Priority_Sel, PSW_En, DST_Zero, MemHigh_b,
                                 Mem_Enable, MOVB, Mem_Write_En, UseCin,
                                 Ashc, rotate, InvShamt, Interrupt_Clear,
                                 N_Neg, Z_Zero, V_Single_Pos, V_Single_Neg,
                                 V_Single, DST_Min, V_Diff_Src,
                                 V_Double_Add, V_Double_Cmp, V_Double_Sub,
                                 V_SFT, V_Neg, Cin;
 assign  MemHigh = ~MemHigh_b;
 assign  MemEn   = reset | Mem_Enable;
 assign  MemWrite= Mem_Write_En & ~MemPSW;
 Controller subController(
     .reset(reset), .State(State),
     .Next_State_Types(NST),
     .State_Types(State_Types),
     .Instruction(Instruction),
```

```
    .PSW({{11{1'b0}}, TraceTrap, Flags}),
    .WD_Zero(WDW_Zero), .WD_Lo_Zero(WDW_Lo_Zero),
    .WD_Neg(isByte ? WDW_7 : WDW_15), .WD_7(WDW_7),
    .DST_15(DST_15), .DST_7(DST_7), .SRC_8(SRC_8),
    .DST_0(DST_0), .DST_Zero(DST_Zero),
    .SRC_15(SRC_Lo_15), .SRC_16(SRC_Hi_0), .SRC_31(SRC_Hi_15), .SRC_0(SRC_Lo_0),
    .Acc_Hi_0(Acc_Hi_0), .Interrupt(Interrupt),
    .Next_State(Next_State), .ALU_Ctrl(ALU_Ctrl),
    .ALU_A_Sel(ALU_A_Sel), .ALU_B_Sel(ALUBSel),
    .SRC_Lo_Sel(SRCLoSel), .SRC_Hi_Sel(SRCHiSel),
    .DST_Sel(DSTSel), .WD_Sel(WDWSel), .Mem_Addr_Sel(MANSel),
    .ACC_Lo_Sel(ACC_Lo_Sel), .ACC_Hi_Sel(ACCHiSel),
    .RF_Addr_Sel(RF_Addr_Sel), .Shamt_Sel(Shamt_Sel),
    .PSW_Sel(PSW_En), .Lo_Lo_Bit(Lo_Lo_Bit),
    .Lo_Hi_Bit(Lo_Hi_Bit), .Lo_Mid_Bit(Lo_Mid_Bit),
    .C_Sel(C_Sel), .V_Sel(V_Sel), .Z_Sel(Z_Sel), .N_Sel(N_Sel),
    .Trap_Sel(Trap_Sel), .Use_Cin(UseCin), .Ashc(Ashc), .Rotate(rotate),
    .SFT_Inv_Shamt(InvShamt), .Shift_SRC_Hi(), .Shift_SRC_Lo(),
    .RF_Enable(RF_Enable), .RF_Write_En(RF_Write_En), .Mem_Enable(Mem_Enable),
    .Mem_Write_En(Mem_Write_En), .Mem_Space_Cur(Mem_Space_Cur),
    .Priority_Sel(Priority_Sel), .Interrupt_Clear(Interrupt_Clear),
    .Instr_Update_En(Update), .Byte(isByte), .Read_Byte(ReadByte),
    .WD_Byte_Sel(WDSel), .Mem_Addr_Out_Base(MemHigh_b), .MOVB(MOVB)
);
// **********************
// *    zipper logic    *
// **********************
assign  PSWSel[0]      = PSW_En;
assign  PSWSel[1]      = ~(PSW_En | (MemPSW & Mem_Write_En));
assign  PSWSel[2]      = ~PSW_En & MemPSW & Mem_Write_En;
wire    SRC_7, SRC_15;
reg     [2:0]   RF_Addr;
assign  SRC_7           = SRC_Lo_7;
assign  SRC_15          = SRC_Lo_15;
assign  N_Neg           = isByte ?  WDW_7           :  WDW_15           ;
assign  Z_Zero          = isByte ?  WDW_Lo_Zero     :  WDW_Zero         ;
assign  DST_Zero        = isByte ? ~DST_7 & L7Z     : ~DST_15 & L15Z    ;
assign  V_Single_Pos    = isByte ? ~DST_7 & WDW_7   : ~DST_15 & WDW_15  ;
assign  V_Single_Neg    = isByte ?  DST_7 &~WDW_7   :  DST_15 &~WDW_15  ;
assign  V_Single        = V_Single_Pos | V_Single_Neg;
assign  DST_Min         = isByte ?  DST_7 & L7Z     :  DST_15 & L15Z    ;
assign  V_Diff_Src      = isByte ?  DST_7 ^ SRC_7   :  DST_15 ^ SRC_15  ;
assign  V_Double_Add    = ~V_Diff_Src   & V_Single;
assign  V_Double_Cmp    = V_Diff_Src    &~V_Single;
assign  V_Double_Sub    = V_Diff_Src    & V_Single;
assign  V_SFT           = FlagsNext[3] ^ FlagsNext[0];
assign  V_Neg           = ~(V_Single | Z_Zero);
assign  Kernal          = ~|(Mem_Space_Cur ? PrevPrivilege : CurPrivilege);

// 3 : N, 2 : Z, 1 : V, 0 : C
always@(*)
begin
    case(C_Sel)
        `c_alu          :   FlagsNext[0] <=  CALUOut;
        `c_alu_not      :   FlagsNext[0] <= ~CALUOut;
        `c_set          :   FlagsNext[0] <=  1'b1;
```

```
        `c_clear        :   FlagsNext[0] <=  1'b0;
        `c_not_z        :   FlagsNext[0] <= ~Z_Zero;
        `c_same         :   FlagsNext[0] <=  Flags[0];
        default         :   FlagsNext[0] <= 1'bx;
    endcase
end
always@(*)
begin
    case(V_Sel)
        `v_single       :   FlagsNext[1] <=  V_Single;
        `v_double_add   :   FlagsNext[1] <=  V_Double_Add;
        `v_sft          :   FlagsNext[1] <=  V_SFT;
        `v_clear        :   FlagsNext[1] <=  1'b0;
        `v_set          :   FlagsNext[1] <=  1'b1;
        `v_neg          :   FlagsNext[1] <=  V_Neg;
        `v_double_sub   :   FlagsNext[1] <=  V_Double_Sub;
        `v_double_cmp   :   FlagsNext[1] <=  V_Double_Cmp;
        `v_single_pos   :   FlagsNext[1] <=  V_Single_Pos;
        `v_single_neg   :   FlagsNext[1] <=  DST_Min;
        `v_same         :   FlagsNext[1] <=  Flags[1];
        default         :   FlagsNext[1] <= 1'bx;
    endcase
    case(Z_Sel)
        `z_zero         :   FlagsNext[2] <=  Z_Zero;
        `z_set          :   FlagsNext[2] <=  1'b1;
        `z_clear        :   FlagsNext[2] <=  1'b0;
        `z_same         :   FlagsNext[2] <=  Flags[2];
    endcase
    case(N_Sel)
        `n_neg          :   FlagsNext[3] <= N_Neg;
        `n_set          :   FlagsNext[3] <= 1'b1;
        `n_clear        :   FlagsNext[3] <= 1'b0;
        `n_same         :   FlagsNext[3] <= Flags[3];
    endcase
end
always @(*)
begin
    case(RF_Addr_Sel)
        `rf_addr_sp     :   RF_Addr      <= 3'b110;
        `rf_addr_r5     :   RF_Addr      <= 3'b101;
        `rf_addr_dst_reg:   RF_Addr      <= Instruction[`DST_REG];
        `rf_addr_src_reg:   RF_Addr      <= Instruction[`SRC_REG];
        `rf_addr_rv1    :   RF_Addr      <= Instruction[`SRC_REG] | 3'b001;
        `rf_addr_pc     :   RF_Addr      <= 3'b111;
    endcase
end
always @(*)
begin
    if(reset) begin
        RegSelect <= 9'b1_0000_0000;
    end else begin
        case(RF_Addr)
            3'b000 :            RegSelect <= 9'b0_0000_0001;
            3'b001 :            RegSelect <= 9'b0_0000_0010;
            3'b010 :            RegSelect <= 9'b0_0000_0100;
            3'b011 :            RegSelect <= 9'b0_0000_1000;
```

```
            3'b100 :            RegSelect <= 9'b0_0001_0000;
            3'b101 :            RegSelect <= 9'b0_0010_0000;
            3'b110 : if(Kernal) RegSelect <= 9'b0_1000_0000;
                     else       RegSelect <= 9'b0_0100_0000;
            3'b111 :            RegSelect <= 9'b1_0000_0000;
        endcase
    end
end
always @(*)
begin
    case(Shamt_Sel)
        `shamt_src :
            begin
                SFTAmt      <=  1'b1;
                SFTConst    <=  3'bxxx;
            end
        `shamt_L1:
            begin
                SFTAmt      <=  1'b0;
                SFTConst    <=  3'b100;
            end
        `shamt_R1:
            begin
                SFTAmt      <=  1'b0;
                SFTConst    <=  3'b111;
            end
        `shamt_8:
            begin
                SFTAmt      <=  1'b0;
                SFTConst    <=  3'b010;
            end
        `shamt_zero:
            begin
                SFTAmt      <=  1'b0;
                SFTConst    <=  3'b000;
            end
        default:
            begin
                SFTAmt      <=  1'bx;
                SFTConst    <=  3'bxxx;
            end
    endcase
    if(ACC_Lo_Sel == `acc_lo_trap) begin
        ACCLoSel <= ACC_Lo_Sel == `trap_mem_data ? `acc_lo_mem : `acc_lo_trap;
        case(Trap_Sel)
            `trap_io        :   TrapVal <= 3'b100;
            `trap_trap      :   TrapVal <= 3'b111;
            `trap_emt       :   TrapVal <= 3'b110;
            `trap_bpt       :   TrapVal <= 3'b011;
            `trap_broken    :   TrapVal <= 3'b001;
            default         :   TrapVal <= 3'bxxx;
        endcase
    end else begin
        ACCLoSel <= ACC_Lo_Sel;
        TrapVal  <= 3'bxxx;
    end
```

```
    end
    assign  ALUASel = ALU_A_Sel[4:0];
    assign  ALUAConst = ALU_A_Sel[7:5];
    assign  LowEnable = RegSelect & {9{(RF_Write_En | reset)& clkb}};
    assign  HiEnable= LowEnable & {9{isByte_b | MOVB | reset}};
    assign  PriorityNext = Priority_Sel ? Instruction[`SRC_REG] : Priority;
    assign  Cin = Flags[0];
    assign  ReadByte_b = ~ReadByte;
//  ALUStuff
    assign  ALUinvertB  = ALU_Ctrl[3] | ALU_Ctrl[5];
    assign  ALUinvertA  = ALU_Ctrl[4];
    assign  add_b       = ALU_Ctrl[1];
    assign  ALUC_in     = (ALU_Ctrl[4] | ALU_Ctrl[5]) | (UseCin & Cin);
    assign  enALU       = ALU_Ctrl[1] | ALU_Ctrl[4] | ALU_Ctrl[5] | ALU_Ctrl[6];
    assign  ALUenAND    = ALU_Ctrl[7] | ALU_Ctrl[3];
    assign  ALUenOR     = ALU_Ctrl[2];
    assign  ALUenCOMP   = ALU_Ctrl[0];
    assign  enALU_b     = ~enALU;
    assign  ALUenAND_b  = ~ALUenAND;
    assign  ALUenOR_b   = ~ALUenOR;
    assign  ALUenCOMP_b = ~ALUenCOMP;
endmodule

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

  wire [WIDTH-1:0] mid;

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


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

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

module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

A.26 test_cpu.v

```
`timescale 1ns / 100ps
`ifndef TEST_FILE
```

```
‘define TEST_FILE   "../basic/xor"
‘endif
‘define PC_RESET    16’h0000

‘include "pdp11.h"
‘include "States.h"
module test_cpu;

reg clk, clk_b, reset;

wire [15:0] Mem_Read_Data, Mem_Addr;
wire        Mem_Write_En, Mem_En, Byte, Interrupt, Kernal;
wire [7:0]  Mem_Write_Data;

wire [7:0] State;

reg  [15:0] Load_PC;

initial begin
    clk = 0;
    clk_b = 1;
    $monitor("Time: %d, State: %h", $time, State);
    $dumpfile("pdp11_init_test.vcd");
    $dumpvars;

    reset = 1;

    #600 reset = 0;
    #10000000 $finish();
end

always
begin
    #100 clk_b <= 0;
    #100 clk <= 1;
    #100 clk <= 0;
    #100 clk_b <= 1;
end

always @(posedge clk_b) begin
    if (Mem_Write_En & Mem_Addr == 0 || State == 16’hf0)
        ‘TICK $finish();
end
wire sdo, Interrupt_Clear;
core PDP11(.clk(clk), .clkb(clk_b), .sdi(1’bz), .scan(1’b0), .reset(reset),
          .MDI(Mem_Read_Data),
          .Interrupt(Interrupt),
          .MAO(Mem_Addr), .WD(Mem_Write_Data),
          .MemWrite(Mem_Write_En), .MemEn(Mem_En),
          .ReadByte(Byte), .InterruptClear(Interrupt_Clear),
          .Kernal(Kernal),
          .sdo(sdo));

Memory mem(.clk(clk), .reset(reset),
           .Mem_Addr(Mem_Addr), .Mem_Write_Data(Mem_Write_Data),
           .Mem_Enable(Mem_En), .Mem_Write_Enable(Mem_Write_En),
```

```
            .Mem_Read_Data(Mem_Read_Data), .Read_Valid(Valid),
            .Byte(Byte), .Interrupt(Interrupt), .Kernal(Kernal),
            .Interrupt_Clear(Interrupt_Clear));
assign State = PDP11.sc.State;
`ifdef v_filename
    reg[31:0] v_file;
    initial begin
        v_file = $fopen(`v_filename, "w");
        $fdisplay(v_file, "_C_V_Z_N_       R0       _       R1       _       R2       _       R3       _       R4
    end
    always @(negedge clk_b)
        if(State == `FETCH0)
        begin
            if(~reset)
                $fdisplay(v_file, "_%b_%b_%b_%b_%b_%b_%b_%b_%b_%b_%b_%b",
                PDP11.Flags[0], PDP11.Flags[1], PDP11.Flags[2], PDP11.Flags[3],
                {PDP11.dp.RB.RFB.RF.r0.reg_hi[15].net40, PDP11.dp.RB.RFB.RF.r0.reg_hi[14].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r0.reg_hi[11].net40, PDP11.dp.RB.RFB.RF.r0.reg_hi[10].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r0.reg_low[7].net40, PDP11.dp.RB.RFB.RF.r0.reg_low[6].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r0.reg_low[3].net40, PDP11.dp.RB.RFB.RF.r0.reg_low[2].net40, PDP11.dp.RB.RFB.RF
                {PDP11.dp.RB.RFB.RF.r1.reg_hi[15].net40, PDP11.dp.RB.RFB.RF.r1.reg_hi[14].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r1.reg_hi[11].net40, PDP11.dp.RB.RFB.RF.r1.reg_hi[10].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r1.reg_low[7].net40, PDP11.dp.RB.RFB.RF.r1.reg_low[6].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r1.reg_low[3].net40, PDP11.dp.RB.RFB.RF.r1.reg_low[2].net40, PDP11.dp.RB.RFB.RF
                {PDP11.dp.RB.RFB.RF.r2.reg_hi[15].net40, PDP11.dp.RB.RFB.RF.r2.reg_hi[14].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r2.reg_hi[11].net40, PDP11.dp.RB.RFB.RF.r2.reg_hi[10].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r2.reg_low[7].net40, PDP11.dp.RB.RFB.RF.r2.reg_low[6].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r2.reg_low[3].net40, PDP11.dp.RB.RFB.RF.r2.reg_low[2].net40, PDP11.dp.RB.RFB.RF
                {PDP11.dp.RB.RFB.RF.r3.reg_hi[15].net40, PDP11.dp.RB.RFB.RF.r3.reg_hi[14].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r3.reg_hi[11].net40, PDP11.dp.RB.RFB.RF.r3.reg_hi[10].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r3.reg_low[7].net40, PDP11.dp.RB.RFB.RF.r3.reg_low[6].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r3.reg_low[3].net40, PDP11.dp.RB.RFB.RF.r3.reg_low[2].net40, PDP11.dp.RB.RFB.RF
                {PDP11.dp.RB.RFB.RF.r4.reg_hi[15].net40, PDP11.dp.RB.RFB.RF.r4.reg_hi[14].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r4.reg_hi[11].net40, PDP11.dp.RB.RFB.RF.r4.reg_hi[10].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r4.reg_low[7].net40, PDP11.dp.RB.RFB.RF.r4.reg_low[6].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r4.reg_low[3].net40, PDP11.dp.RB.RFB.RF.r4.reg_low[2].net40, PDP11.dp.RB.RFB.RF
                {PDP11.dp.RB.RFB.RF.r5.reg_hi[15].net40, PDP11.dp.RB.RFB.RF.r5.reg_hi[14].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r5.reg_hi[11].net40, PDP11.dp.RB.RFB.RF.r5.reg_hi[10].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r5.reg_low[7].net40, PDP11.dp.RB.RFB.RF.r5.reg_low[6].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r5.reg_low[3].net40, PDP11.dp.RB.RFB.RF.r5.reg_low[2].net40, PDP11.dp.RB.RFB.RF
                {PDP11.dp.RB.RFB.RF.r6_k.reg_hi[15].net40, PDP11.dp.RB.RFB.RF.r6_k.reg_hi[14].net40, PDP11.dp.RB.RF
                 PDP11.dp.RB.RFB.RF.r6_k.reg_hi[11].net40, PDP11.dp.RB.RFB.RF.r6_k.reg_hi[10].net40, PDP11.dp.RB.RF
                 PDP11.dp.RB.RFB.RF.r6_k.reg_low[7].net40, PDP11.dp.RB.RFB.RF.r6_k.reg_low[6].net40, PDP11.dp.RB.RF
                 PDP11.dp.RB.RFB.RF.r6_k.reg_low[3].net40, PDP11.dp.RB.RFB.RF.r6_k.reg_low[2].net40, PDP11.dp.RB.RF
                {PDP11.dp.RB.RFB.RF.r7.reg_hi[15].net40, PDP11.dp.RB.RFB.RF.r7.reg_hi[14].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r7.reg_hi[11].net40, PDP11.dp.RB.RFB.RF.r7.reg_hi[10].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r7.reg_low[7].net40, PDP11.dp.RB.RFB.RF.r7.reg_low[6].net40, PDP11.dp.RB.RFB.RF
                 PDP11.dp.RB.RFB.RF.r7.reg_low[3].net40, PDP11.dp.RB.RFB.RF.r7.reg_low[2].net40, PDP11.dp.RB.RFB.RF
        end
`endif

`ifdef m_filename
    integer m_file;
    initial m_file = $fopen(`m_filename, "w");
    always @(posedge clk_b)
        if (Mem_Write_En)
```

```
            $fdisplay(m_file, "_%b_%b", Mem_Addr, Mem_Write_Data);
`endif


    `ifdef t_filename
        integer t_file;
        initial t_file = $fopen(`t_filename, "w");
    `endif

    wire [15:0] STDOUT;
    assign STDOUT = -138;
    always @(posedge clk_b)
        if  (Mem_Write_En && (Mem_Addr == STDOUT))
        begin
`ifdef t_filename
            $fwrite(t_file, "%c", Mem_Write_Data);
`endif
            $display("@TTY_%c", Mem_Write_Data);
        end


endmodule
module Memory #(parameter LENGTH=262144) (input  clk, reset,
                                          input  [15:0] Mem_Addr,
                                          input  [7:0] Mem_Write_Data,
                                          input  Mem_Enable,
                                          input  Mem_Write_Enable,
                                          input  Byte,
                                          input  Interrupt_Clear,
                                          input  Kernal,
                                          output [15:0] Mem_Read_Data,
                                          output Read_Valid,
                                          output Interrupt);

    reg [7:0] mem[LENGTH-1:0];

    reg [15:0] MMU_SR0;
    reg [15:0] MMU_SR2;
    reg [7:0] User_PDR[3:0];
    reg [7:0] User_PAR[3:0];
    reg [7:0] Kernal_PDR[3:0];
    reg [7:0] Kernal_PAR[3:0];

    wire [7:0]  foo;
    wire [7:0]  foo0;
    wire [15:0] PC_reset;

    wire        Extend_Bit;
    wire [3:0]  PAR_Sel;
    wire [15:0] K_PAR_Full;
    wire [15:0] U_PAR_Full;
    wire [11:0] K_PAD;
    wire [11:0] U_PAD;
    wire [17:0] Kernal_PA;
    wire [17:0] User_PA;
    wire [17:0] PA;
```

```
    wire [17:0] addr_plus_one;
    assign PC_reset = `PC_RESET;

    assign Interrupt = 1'b0;

    // Comment until next comment unless testing interrupt
/*    wire Interrupt_Start  =~|(count);
    reg Interrupt;
    reg done;

    always @(posedge clk) begin
        if (reset) begin
            Interrupt = 1'b0;
            done = 1'b0;
        end
        else begin
            Interrupt <= Interrupt ? ~Interrupt_Clear : ~done & Interrupt_Start;
            done <= done | Interrupt_Start;
        end
    end

    reg [4:0] count;
    initial begin
        count = 5'b0_0001;
        done = 1'b0;
    end

    always @(posedge clk) begin
        count <= count+1'b1;
    end

    assign Mem_Read_Data[7:0] = Interrupt ? 8'h40 : (reset ? PC_reset[7:0] : (Mem_Enable ? mem[Mem_Addr] : {8{1'bx}
    assign Mem_Read_Data[15:8] = Interrupt ? 8'h00 : (reset ? PC_reset[15:8] : (Mem_Enable ? (Byte ? 8'b0 : mem[add
*/
    // End test interrupt stuff

    assign foo0 = mem[Mem_Addr];
    assign foo = mem[addr_plus_one];
    integer i;



    initial begin
        $readmemb(`TEST_FILE, mem);
        for (i=0; i<16; i=i+1) begin
            User_PDR[i] = 0;
            User_PAR[i] = 0;
            Kernal_PDR[i] = 0;
            Kernal_PAR[i] = 0;
        end
    end

    assign Extend_Bit = &Mem_Addr[15:12];
    assign PAR_Sel = {Mem_Addr[15:13],1'b0};
    assign K_PAR_Full = {Kernal_PAR[PAR_Sel+1],Kernal_PAR[PAR_Sel]};
    assign K_PAD = K_PAR_Full[11:0];
```

```
    assign Kernal_PA = {{K_PAD+Mem_Addr[12:6]},Mem_Addr[5:0]};
    assign U_PAR_Full = {User_PAR[PAR_Sel+1],User_PAR[PAR_Sel]};
    assign U_PAD = U_PAR_Full[11:0];
    assign User_PA = {{U_PAD+Mem_Addr[12:6]},Mem_Addr[5:0]};
    assign PA = MMU_SR0[0] ? (Kernal ? Kernal_PA : User_PA) : {{2{Extend_Bit}},Mem_Addr};

    assign addr_plus_one = PA + 1'b1;

    always @(posedge clk) begin
        if (reset) begin
            MMU_SR0 <= 16'h0;
            MMU_SR2 <= 16'h0;
        end

        if (Mem_Enable & Mem_Write_Enable) begin
            if (PA[17:1] == 17'h1FFBD) begin
                if (PA[0] == 0)
                    MMU_SR0 <= {MMU_SR0[15:8],Mem_Write_Data};
                else
                    MMU_SR0 <= {Mem_Write_Data,MMU_SR0[7:0]};
            end
            if (PA[17:4] == 14'h3F4C) begin
                Kernal_PDR[PA[3:0]] <= Mem_Write_Data;
            end
            if (PA[17:4] == 14'h3F4E) begin
                Kernal_PAR[PA[3:0]] <= Mem_Write_Data;
            end
            if (PA[17:4] == 14'h3FF8) begin
                User_PDR[PA[3:0]] <= Mem_Write_Data;
            end
            if (PA[17:4] == 14'h3FFA) begin
                User_PDR[PA[3:0]] <= Mem_Write_Data;
            end

            mem[PA] <= Mem_Write_Data;
        end
    end

    // Need to add read by byte capability

    assign Mem_Read_Data[7:0] = reset ? PC_reset[7:0] : (Mem_Enable ? mem[PA] : {8{1'bx}});
    assign Mem_Read_Data[15:8] = reset ? PC_reset[15:8] : (Mem_Enable ? (Byte ? 8'b0 : mem[addr_plus_one]) : {8{1'b
    assign Read_Valid = Mem_Enable;

endmodule
```

A.27 test_scan.v

```
`timescale 1ns / 100ps
`ifndef TEST_FILE
`define TEST_FILE   "../basic/xor"
`endif
`define PC_RESET 16'h0000
`include "pdp11.h"
```

```
module test_cpu;
            reg clk, clk_b, reset, scan, sdi;
            wire [15:0] Mem_Read_Data, Mem_Addr;
            wire        Mem_Write_En, Mem_En, Byte, Interrupt, Kernal;
            wire [7:0]  Mem_Write_Data;

            wire sdo, Interrupt_Clear;

            reg  [15:0] Load_PC;

        initial begin
            clk = 0;
            clk_b = 1;
            reset = 1;
            scan = 1;
            sdi = 0;
            #600 reset = 0;
            #99400 sdi = 1;
            #10000000 $finish();
        end
    always@(negedge clk) $display("Update");
        always@(*) $display("PSW %b, RF %b", PDP11.dp.RB.PSWB.PSW_SR.q, PDP11.dp.RB.RFB.LatchOut);
//        always@(*) $display("PSW %b, RF %b, ACCHHi %b, MAB %b", PDP11.dp.RB.PSWB.PSW, PDP11.dp.RF, PDP11.dp.AB.AH
        always@(*) begin
            if(sdi & sdo)
            begin
                $display("Time %d", $time);
                $finish;
            end
        end
        always
        begin
            #100 clk_b <= 0;
            #100 clk <= 1;
            #100 clk <= 0;
            #100 clk_b <= 1;
        end

            always @(posedge clk_b) begin
                if (Mem_Write_En & Mem_Addr & ~scan == 0)
                    `TICK $finish();
            end

            core PDP11(.clk(clk), .clkb(clk_b), .sdi(sdi), .scan(scan), .reset(reset),
            .MDI(Mem_Read_Data),
            .Interrupt(Interrupt),
            .MAO(Mem_Addr), .WD(Mem_Write_Data),
            .MemWrite(Mem_Write_En), .MemEn(Mem_En),
            .ReadByte(Byte), .InterruptClear(Interrupt_Clear),
            .Kernal(Kernal),
            .sdo(sdo));

            Memory mem(.clk(clk), .reset(reset),
            .Mem_Addr(Mem_Addr), .Mem_Write_Data(Mem_Write_Data),
            .Mem_Enable(Mem_En), .Mem_Write_Enable(Mem_Write_En),
            .Mem_Read_Data(Mem_Read_Data), .Read_Valid(Valid),
```

```
            .Byte(Byte), .Interrupt(Interrupt), .Kernal(Kernal),
            .Interrupt_Clear(Interrupt_Clear));
endmodule
module Memory #(parameter LENGTH=262144) (input  clk, reset,
                                          input  [15:0] Mem_Addr,
                                          input  [7:0] Mem_Write_Data,
                                          input  Mem_Enable,
                                          input  Mem_Write_Enable,
                                          input  Byte,
                                          input  Interrupt_Clear,
                                          input  Kernal,
                                          output [15:0] Mem_Read_Data,
                                          output Read_Valid,
                                          output Interrupt);

    reg [7:0] mem[LENGTH-1:0];

    reg [15:0] MMU_SR0;
    reg [15:0] MMU_SR2;
    reg [7:0] User_PDR[3:0];
    reg [7:0] User_PAR[3:0];
    reg [7:0] Kernal_PDR[3:0];
    reg [7:0] Kernal_PAR[3:0];

    wire [7:0]  foo;
    wire [7:0]  foo0;
    wire [15:0] PC_reset;

    wire        Extend_Bit;
    wire [3:0]  PAR_Sel;
    wire [15:0] K_PAR_Full;
    wire [15:0] U_PAR_Full;
    wire [11:0] K_PAD;
    wire [11:0] U_PAD;
    wire [17:0] Kernal_PA;
    wire [17:0] User_PA;
    wire [17:0] PA;
    wire [17:0] addr_plus_one;
    assign PC_reset = `PC_RESET;

    assign Interrupt = 1'b0;

    // Comment until next comment unless testing interrupt
/*    wire Interrupt_Start  =~|(count);
    reg Interrupt;
    reg done;

    always @(posedge clk) begin
        if (reset) begin
            Interrupt = 1'b0;
            done = 1'b0;
        end
        else begin
            Interrupt <= Interrupt ? ~Interrupt_Clear : ~done & Interrupt_Start;
            done <= done | Interrupt_Start;
        end
```

```
    end

    reg [4:0] count;
    initial begin
        count = 5'b0_0001;
        done = 1'b0;
    end

    always @(posedge clk) begin
        count <= count+1'b1;
    end

    assign Mem_Read_Data[7:0] = Interrupt ? 8'h40 : (reset ? PC_reset[7:0] : (Mem_Enable ? mem[Mem_Addr] : {8{1'bx}
    assign Mem_Read_Data[15:8] = Interrupt ? 8'h00 : (reset ? PC_reset[15:8] : (Mem_Enable ? (Byte ? 8'b0 : mem[add
*/
    // End test interrupt stuff

    assign foo0 = mem[Mem_Addr];
    assign foo = mem[addr_plus_one];
    integer i;



    initial begin
        $readmemb(`TEST_FILE, mem);
        for (i=0; i<16; i=i+1) begin
            User_PDR[i] = 0;
            User_PAR[i] = 0;
            Kernal_PDR[i] = 0;
            Kernal_PAR[i] = 0;
        end
    end

    assign Extend_Bit = &Mem_Addr[15:12];
    assign PAR_Sel = {Mem_Addr[15:13],1'b0};
    assign K_PAR_Full = {Kernal_PAR[PAR_Sel+1],Kernal_PAR[PAR_Sel]};
    assign K_PAD = K_PAR_Full[11:0];
    assign Kernal_PA = {{K_PAD+Mem_Addr[12:6]},Mem_Addr[5:0]};
    assign U_PAR_Full = {User_PAR[PAR_Sel+1],User_PAR[PAR_Sel]};
    assign U_PAD = U_PAR_Full[11:0];
    assign User_PA = {{U_PAD+Mem_Addr[12:6]},Mem_Addr[5:0]};
    assign PA = MMU_SR0[0] ? (Kernal ? Kernal_PA : User_PA) : {{2{Extend_Bit}},Mem_Addr};

    assign addr_plus_one = PA + 1'b1;

    always @(posedge clk) begin
        if (reset) begin
            MMU_SR0 <= 16'h0;
            MMU_SR2 <= 16'h0;
        end

        if (Mem_Enable & Mem_Write_Enable) begin
            if (PA[17:1] == 17'h1FFBD) begin
                if (PA[0] == 0)
                    MMU_SR0 <= {MMU_SR0[15:8],Mem_Write_Data};
                else
```

```
                MMU_SR0 <= {Mem_Write_Data,MMU_SR0[7:0]};
            end
            if (PA[17:4] == 14'h3F4C) begin
                Kernal_PDR[PA[3:0]] <= Mem_Write_Data;
            end
            if (PA[17:4] == 14'h3F4E) begin
                Kernal_PAR[PA[3:0]] <= Mem_Write_Data;
            end
            if (PA[17:4] == 14'h3FF8) begin
                User_PDR[PA[3:0]] <= Mem_Write_Data;
            end
            if (PA[17:4] == 14'h3FFA) begin
                User_PDR[PA[3:0]] <= Mem_Write_Data;
            end

            mem[PA] <= Mem_Write_Data;
        end
    end

    // Need to add read by byte capability

    assign Mem_Read_Data[7:0] = reset ? PC_reset[7:0] : (Mem_Enable ? mem[PA] : {8{1'bx}});
    assign Mem_Read_Data[15:8] = reset ? PC_reset[15:8] : (Mem_Enable ? (Byte ? 8'b0 : mem[addr_plus_one]) : {8{1'b
    assign Read_Valid = Mem_Enable;

endmodule
```

A.28 WDW_Block.v

```
`timescale 1ns / 100ps
module WDW_Block(
    output          WDW_7, WDW_15, WDW_Lo_Zero, WDW_Zero,
    input   [15:0]  WDW);
    assign WDW_Zero     = ~|(WDW[15:0]);
    assign WDW_15       = WDW[15];
    assign WDW_Lo_Zero  = ~|(WDW[7:0]);
    assign WDW_7        = WDW[7];
endmodule
```

A.29 Zipper.v

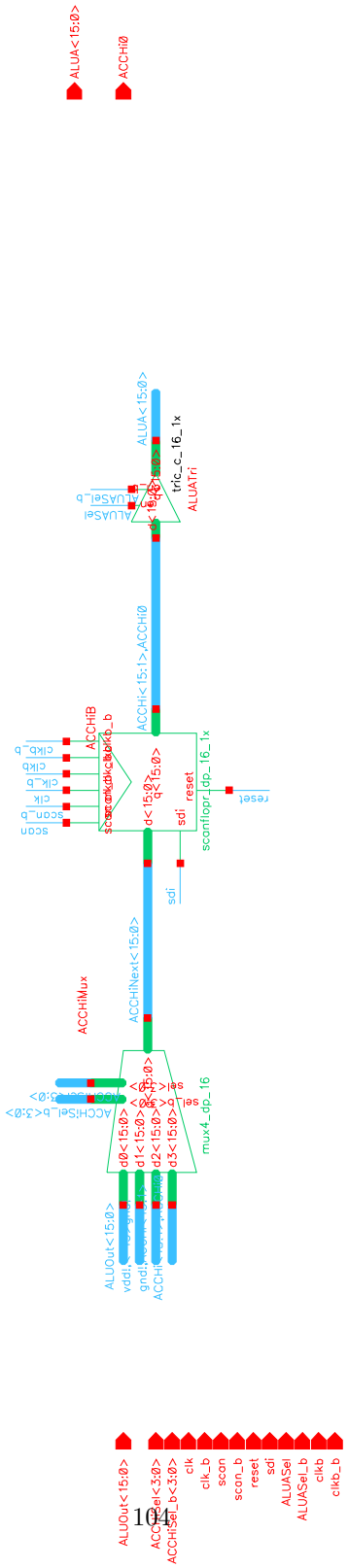
```
`timescale 1ns / 100ps
`include "pdp11.h"
module Zipper(
        input        Mem_Write_En, Mem_Addr_Is_PSW, Mem_En, PSW_Sel_MEM,
                      Byte, WDW_N, WDB_N, WDHB_Z, WDLB_Z, DSTW_N, DSTB_N, SRCW_N, SRCB_N,
                        DST_XSZW, DST_XSZB,
                        C_ALU, C_SFT, C_Cur, N_Cur, V_Cur, Z_Cur, Mem_Space_Cur, Priority_Sel,
        input    [1:0]    Prev_Privilege, Cur_Privilege,
        input    [2:0] DST_REG, SRC_REG, Priority_Cur,
        input    [`C_SEL_LEN-1:0] C_Sel,
        input     [`N_SEL_LEN-1:0] N_Sel,
        input     [`V_SEL_LEN-1:0] V_Sel,
        input     [`Z_SEL_LEN-1:0] Z_Sel,
        input    [`RF_ADDR_SEL_LEN-1:0] RF_Sel,
```

```
        output     PSW_WD, Mem_Read_PSW, PSW_Mem, PSW_Same,
                        C_Next, N_Next, N_Neg, Z_Next, Z_Zero, V_Next,
        output  Kernal,
        output  [2:0] Priority_Next, RF_Addr);

    wire [1:0]  Mem_Privilege;
    wire PSW_Zero;
    // PSW and Condition Code stuff
        assign PSW_Zero = ~PSW_Sel_MEM;
    assign PSW_WD = Mem_Write_En & Mem_Addr_Is_PSW;
    assign Mem_Read_PSW = Mem_En & Mem_Addr_Is_PSW;
    assign PSW_Mem = PSW_Sel_MEM & ~PSW_WD;
    assign PSW_Same = PSW_Zero & ~PSW_WD;
        assign N_Neg = Byte ? WDB_N : WDW_N;
        assign Z_Zero = WDLB_Z & (Byte | WDHB_Z);
    // Change in sign from pos to neg in a single op instr
        assign V_Single_Pos = Byte ? ~DSTB_N & WDB_N : ~DSTW_N & WDW_N;
    // Change in sign from pos to neg in a single op instr
        assign V_Single_Neg = Byte ? DSTB_N & ~WDB_N : DSTW_N & ~WDW_N;
    // If Single Op and Change in sign from DST to result
        assign V_Single = V_Single_Pos | V_Single_Neg;
    // If destination is the minimum integer
        assign DST_MIN = Byte ? DSTB_N & DST_XSZB : DSTW_N & DST_XSZW;
        assign V_Diff_Src = Byte ? DSTB_N ^ SRCB_N: DSTW_N ^ SRCW_N;
    // If Double Op and both SRC and DST have same sign and result has
    // a different sign
        assign V_Double_Add = ~V_Diff_Src & V_Single;
    // If Double Op and SRC and DST have opposite sign and the result has the
    // same sign as the dst
        assign V_Double_Cmp = V_Diff_Src & ~V_Single;
    // If Double Op and SRC and DST have opposite sign and the result has the
    // same sign as the src
        assign V_Double_Sub = V_Diff_Src & V_Single;
    // If Shift or Rotate, then check N_Next != C_Next
        assign V_SFT = N_Next ^ C_Next;
        assign V_Neg = ~(V_Single | Z_Zero);
    // Kernal is Mem_Privilege 00 and User is 11
        assign Kernal = ~(|Mem_Privilege);

// Register write select
        Tri #(.WIDTH(3)) rf0(.d(3'b110), .en(RF_Sel[`RF_ADDR_SP]), .q(RF_Addr));
        Tri #(.WIDTH(3)) rf1(.d(3'b101), .en(RF_Sel[`RF_ADDR_R5]), .q(RF_Addr));
        Tri #(.WIDTH(3)) rf2(.d(DST_REG), .en(RF_Sel[`RF_ADDR_DST_REG]), .q(RF_Addr));
        Tri #(.WIDTH(3)) rf3(.d(SRC_REG), .en(RF_Sel[`RF_ADDR_SRC_REG]), .q(RF_Addr));
        Tri #(.WIDTH(3)) rf5(.d({SRC_REG[2:1],1'b1}), .en(RF_Sel[`RF_ADDR_RV1]), .q(RF_Addr));
        Tri #(.WIDTH(3)) rf4(.d(3'b111), .en(RF_Sel[`RF_ADDR_PC]), .q(RF_Addr));
// C select
        Tri #(.WIDTH(1)) cn0(.d(C_ALU), .en(C_Sel[`C_ALU]), .q(C_Next));
        Tri #(.WIDTH(1)) cn1(.d(~C_ALU), .en(C_Sel[`C_ALU_NOT]), .q(C_Next));
        Tri #(.WIDTH(1)) cn2(.d(C_SFT), .en(C_Sel[`C_SFT]), .q(C_Next));
        Tri #(.WIDTH(1)) cn3(.d(C_Cur), .en(C_Sel[`C_SAME]), .q(C_Next));
        Tri #(.WIDTH(1)) cn4(.d(~Z_Zero), .en(C_Sel[`C_NOT_Z]), .q(C_Next));
        Tri #(.WIDTH(1)) cn5(.d(1'b1), .en(C_Sel[`C_SET]), .q(C_Next));
        Tri #(.WIDTH(1)) cn6(.d(1'b0), .en(C_Sel[`C_CLEAR]), .q(C_Next));
// V select
        Tri #(.WIDTH(1)) vn0 (.d(V_Single),         .en(V_Sel[`V_SINGLE]),             .q(V_Next));
```

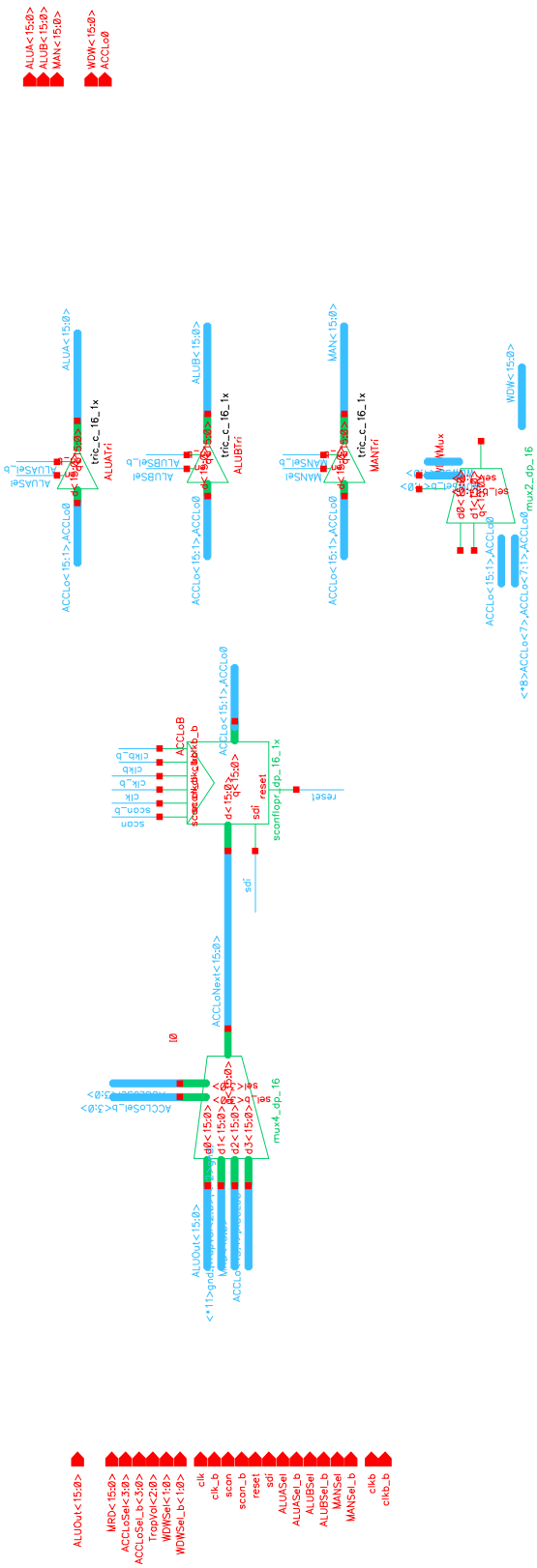
```
        Tri #(.WIDTH(1)) vn1 (.d(V_Double_Add), .en(V_Sel[`V_DOUBLE_ADD]),     .q(V_Next));
        Tri #(.WIDTH(1)) vn2 (.d(V_SFT),                .en(V_Sel[`V_SFT]),                .q(V_Next));
        Tri #(.WIDTH(1)) vn3 (.d(V_Cur),                .en(V_Sel[`V_SAME]),                                  .q(V_
        Tri #(.WIDTH(1)) vn4 (.d(V_Double_Sub), .en(V_Sel[`V_DOUBLE_SUB]),     .q(V_Next));
        Tri #(.WIDTH(1)) vn5 (.d(V_Double_Cmp), .en(V_Sel[`V_DOUBLE_CMP]),     .q(V_Next));
        Tri #(.WIDTH(1)) vn6 (.d(V_Neg),                .en(V_Sel[`V_NEG]),                .q(V_Next));
        Tri #(.WIDTH(1)) vn7 (.d(1'b1),                .en(V_Sel[`V_SET]),                .q(V_Next));
        Tri #(.WIDTH(1)) vn8 (.d(1'b0),                .en(V_Sel[`V_CLEAR]),             .q(V_Next));
        Tri #(.WIDTH(1)) vn9 (.d(V_Single_Pos), .en(V_Sel[`V_SINGLE_POS]),     .q(V_Next));
        Tri #(.WIDTH(1)) vn10(.d(DST_MIN),             .en(V_Sel[`V_DST_MIN]),         .q(V_Next));
// N Select
        Tri #(.WIDTH(1)) nn0(.d(N_Neg), .en(N_Sel[`N_NEG]), .q(N_Next));
        Tri #(.WIDTH(1)) nn1(.d(N_Cur), .en(N_Sel[`N_SAME]), .q(N_Next));
        Tri #(.WIDTH(1)) nn2(.d(1'b1), .en(N_Sel[`N_SET]), .q(N_Next));
        Tri #(.WIDTH(1)) nn3(.d(1'b0), .en(N_Sel[`N_CLEAR]), .q(N_Next));
// Z Select
        Tri #(.WIDTH(1)) zn0(.d(Z_Zero), .en(Z_Sel[`Z_ZERO]), .q(Z_Next));
        Tri #(.WIDTH(1)) zn1(.d(Z_Cur), .en(Z_Sel[`Z_SAME]), .q(Z_Next));
        Tri #(.WIDTH(1)) zn2(.d(1'b1), .en(Z_Sel[`Z_SET]), .q(Z_Next));
        Tri #(.WIDTH(1)) zn3(.d(1'b0), .en(Z_Sel[`Z_CLEAR]), .q(Z_Next));
// Privilege Select
        Mux2 #(.WIDTH(2)) mem_priv(.d0(Prev_Privilege), .d1(Cur_Privilege), .sel(Mem_Space_Cur), .q(Mem_Privilege))
// Priority Select
        Mux2 #(.WIDTH(3)) priority(.d0(Priority_Cur), .d1(SRC_REG), .sel(Priority_Sel), .q(Priority_Next));
endmodule
```

B Schematics

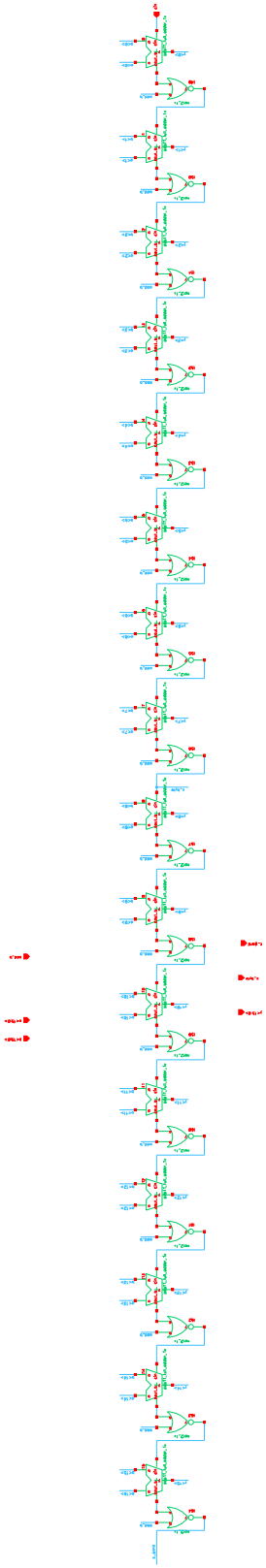
B.1 ACC_Hi_Block_sch.pdf



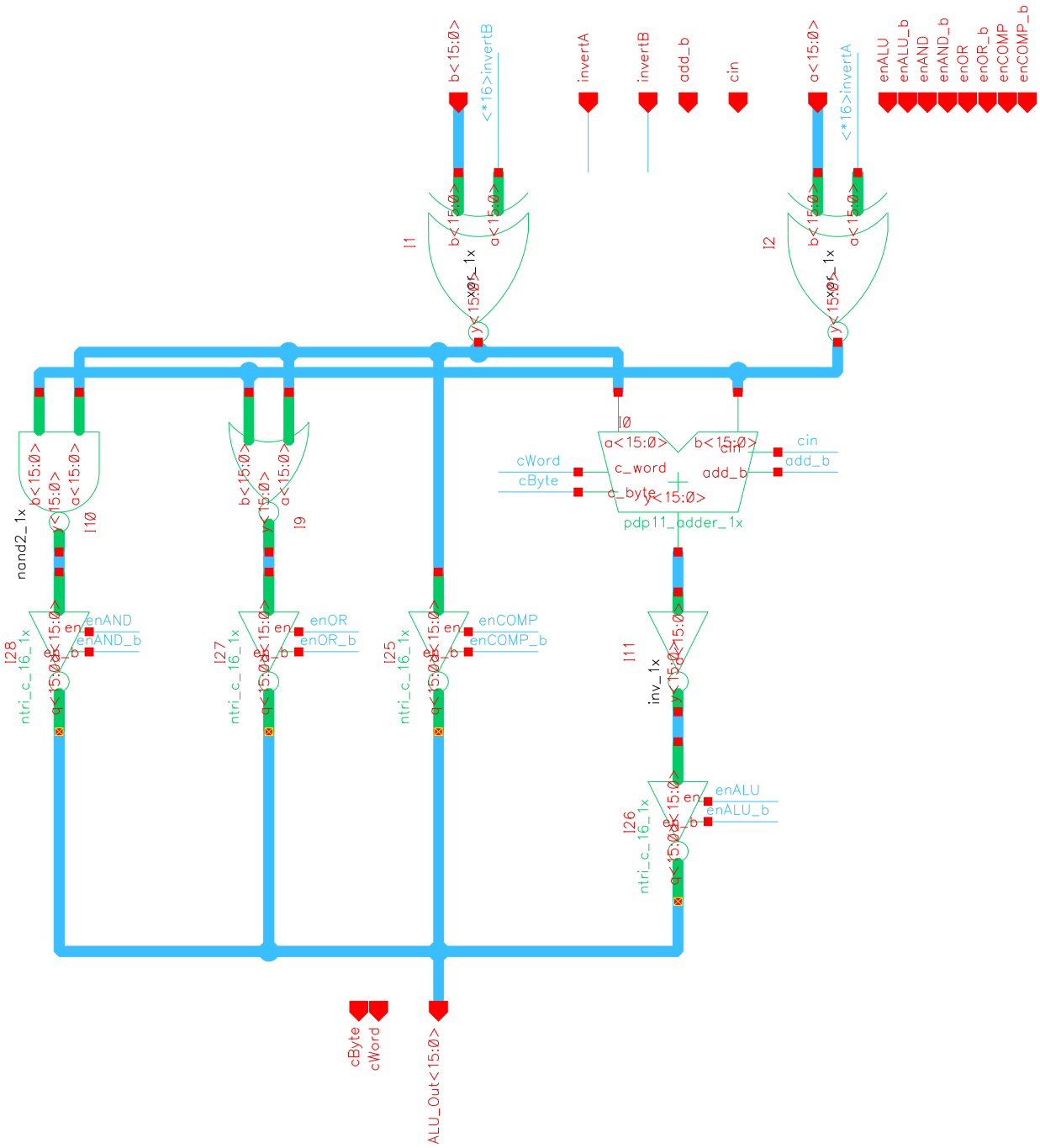
B.2 ACC_Lo_Block_sch.pdf



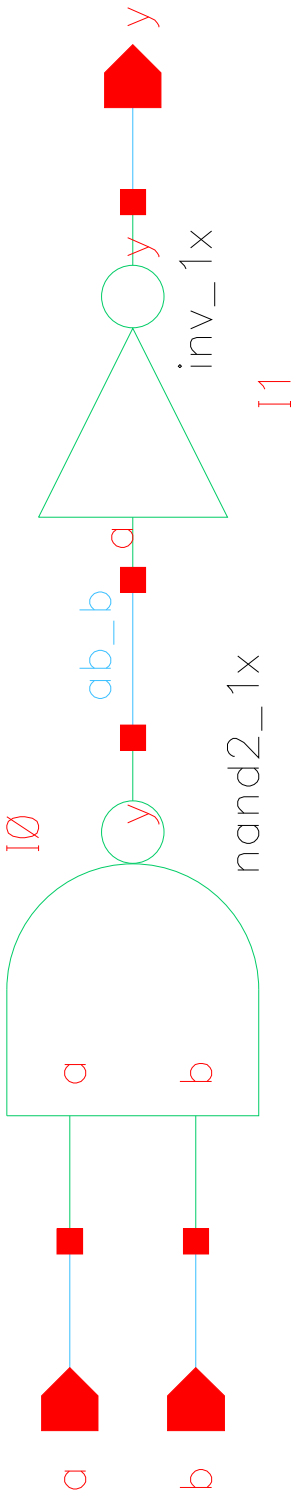
B.3 adder_1x_sch



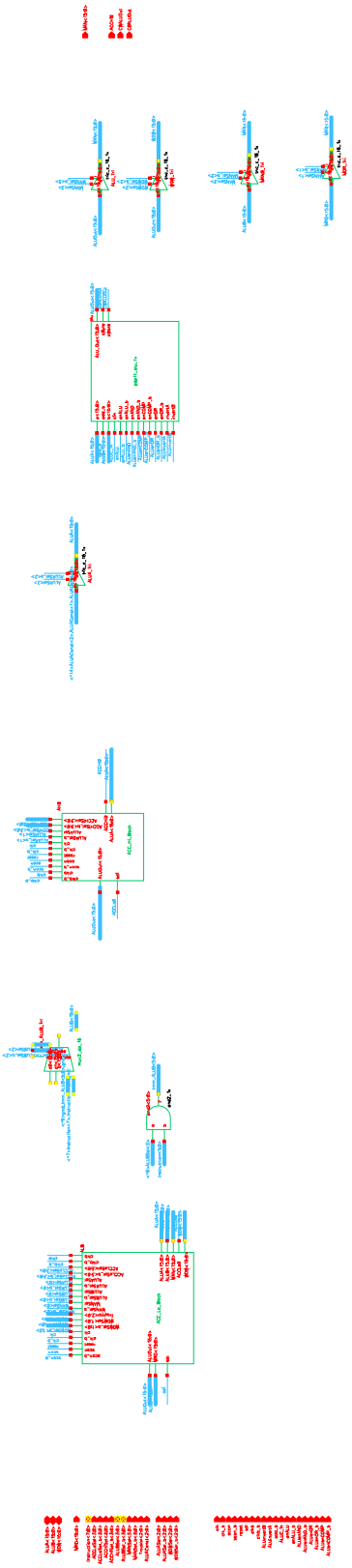
B.4 alu_1x_sch


b<15:0>
<*16>invertB
invertA
invertB
add_b
cin
a<15:0>
<*16>invertA
enALU
enALU_b
enAND
enAND_b
enOR
enOR_b
enCOMP
enCOMP_b
I1
I2
I0
I9
I10
I11
I25
I26
I27
I28
nand2_1x
inv_1x
ntri_c_16_1x
pdp11_adder_1x
cWord
cByte
c_word
c_byte
add_b
cin
enAND
enAND_b
enOR
enOR_b
enCOMP
enCOMP_b
enALU
enALU_b
ALU_Out<15:0>

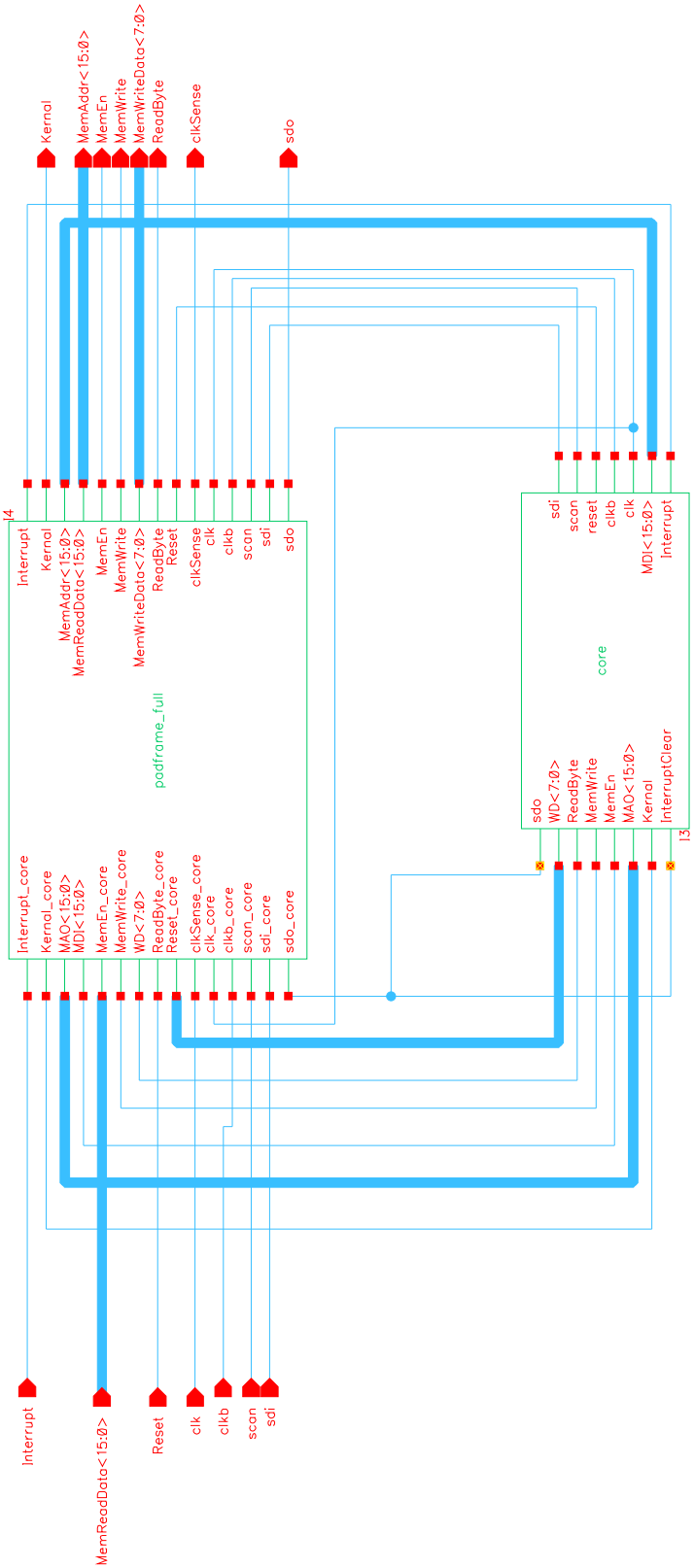
B.5 and2_1x.pdf



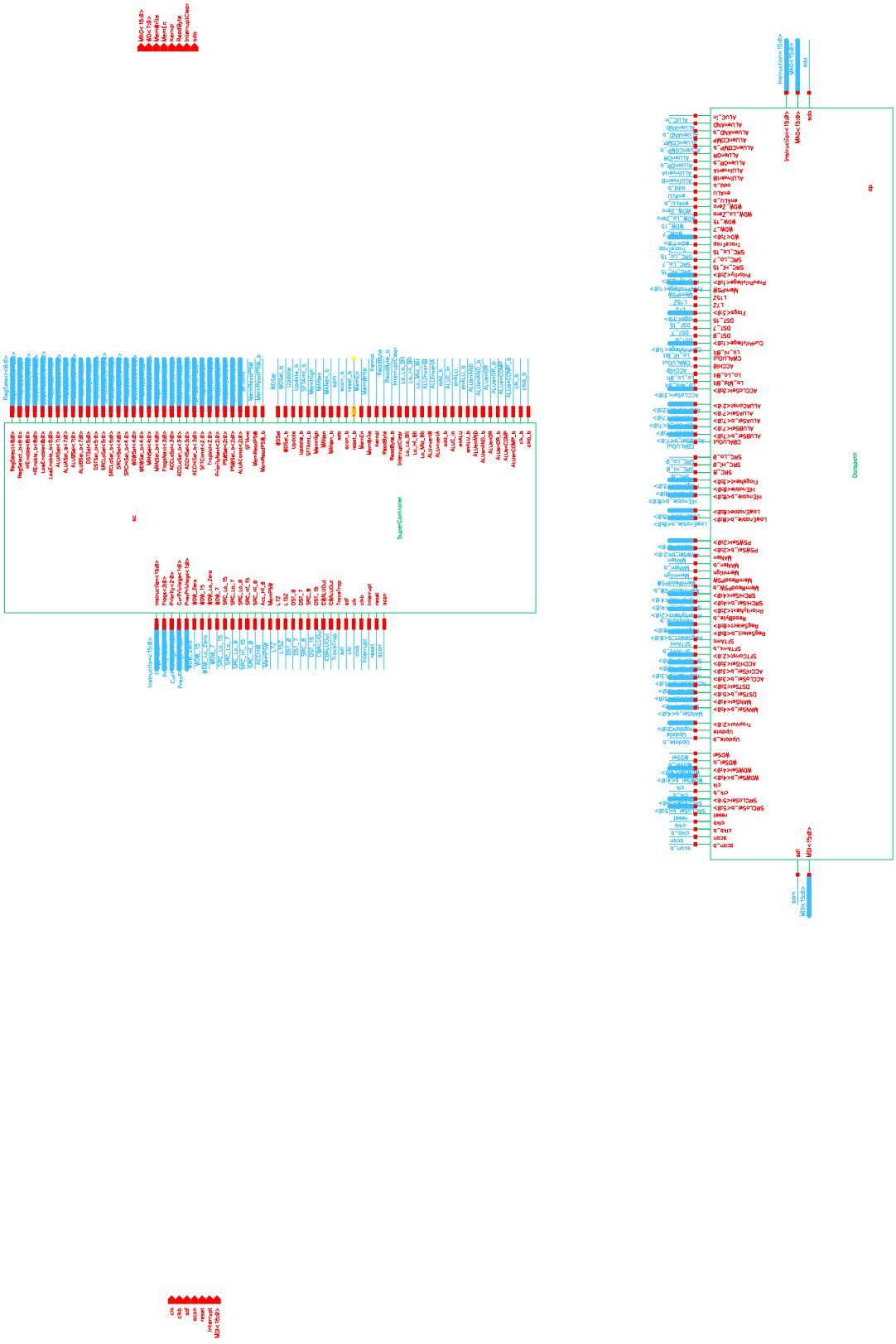
B.6 Ari_Block_sch



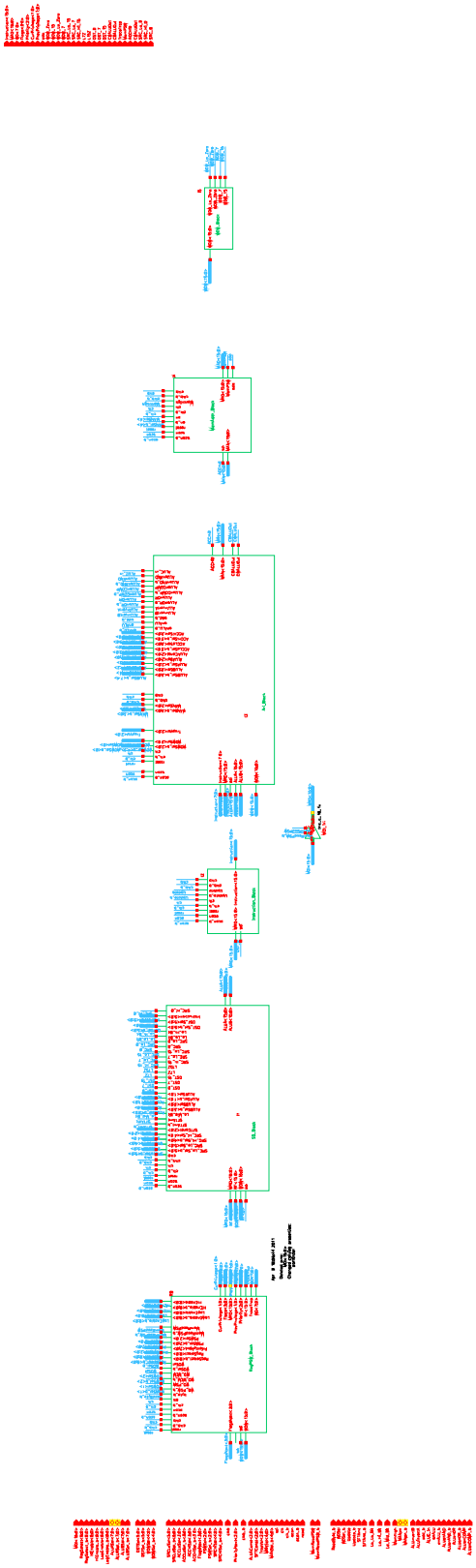
B.7 chip


Kernal
MemAddr<15:0>
MemEn
MemWrite
MemWriteData<7:0>
ReadByte
clkSense
sdo
I4
Interrupt
Kernal
MemAddr<15:0>
MemReadData<15:0>
MemEn
MemWrite
MemWriteData<7:0>
ReadByte
Reset
clkSense
clk
clkb
scan
sdi
sdo
podframe_full
Interrupt_core
Kernal_core
MAO<15:0>
MDI<15:0>
MemEn_core
MemWrite_core
WD<7:0>
ReadByte_core
Reset_core
clkSense_core
clk_core
clkb_core
scan_core
sdi_core
sdo_core
sdi
scan
reset
clkb
clk
MDI<15:0>
Interrupt
core
sdo
WD<7:0>
ReadByte
MemWrite
MemEn
MAO<15:0>
Kernal
InterruptClear
I3
Interrupt
MemReadData<15:0>
Reset
clk
clkb
scan
sdi

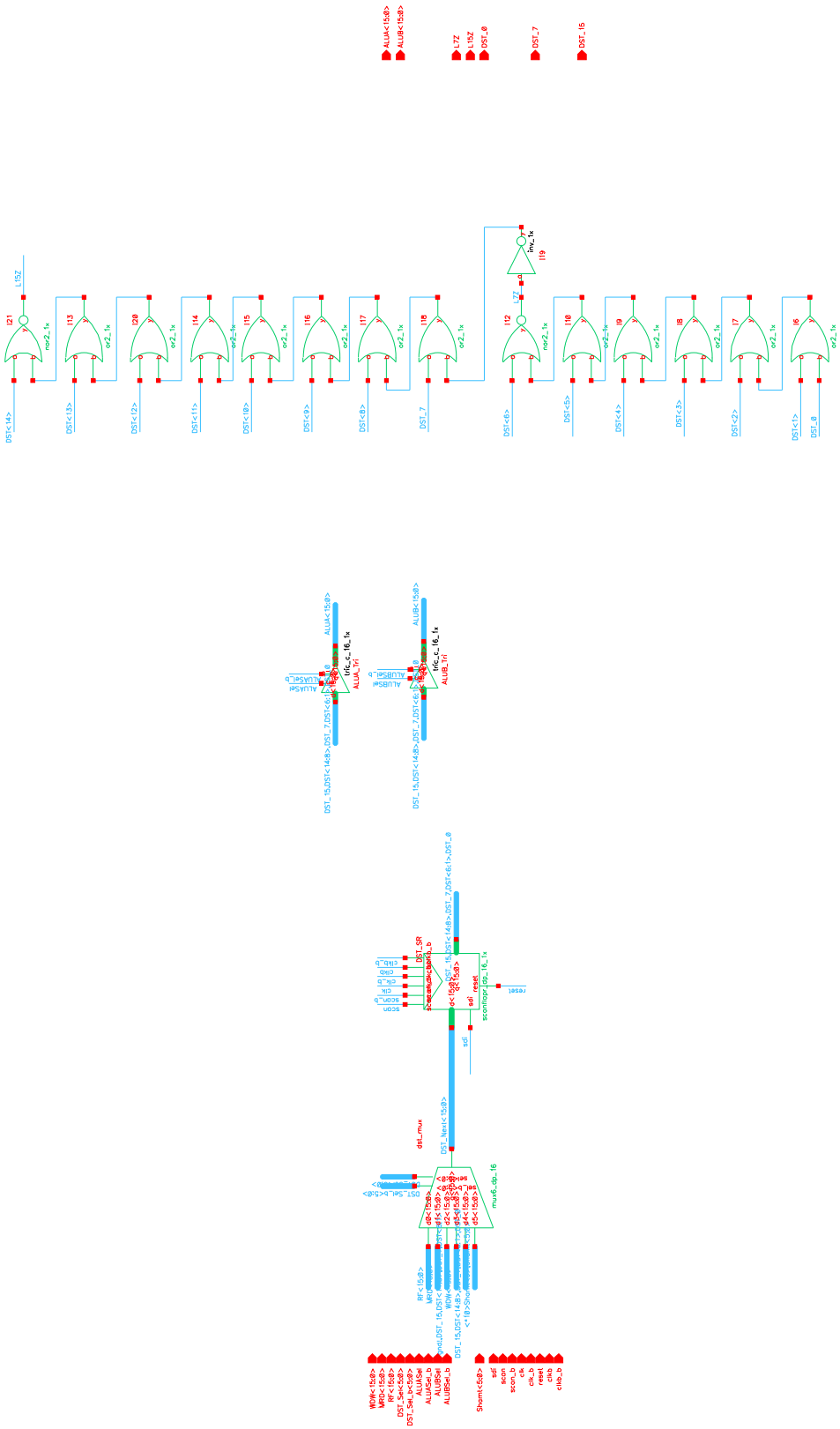
B.8 core



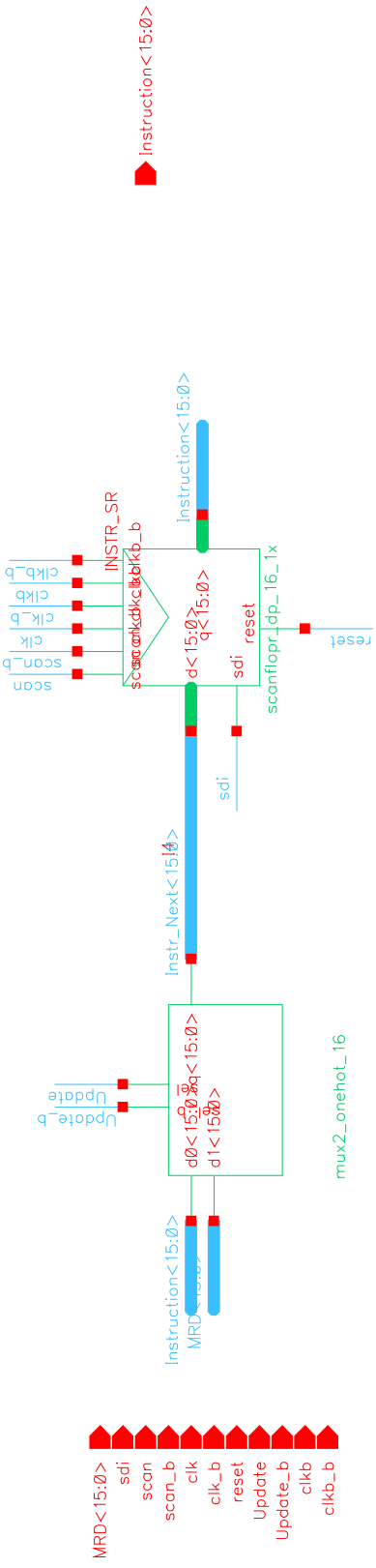
B.9 datapath_sch



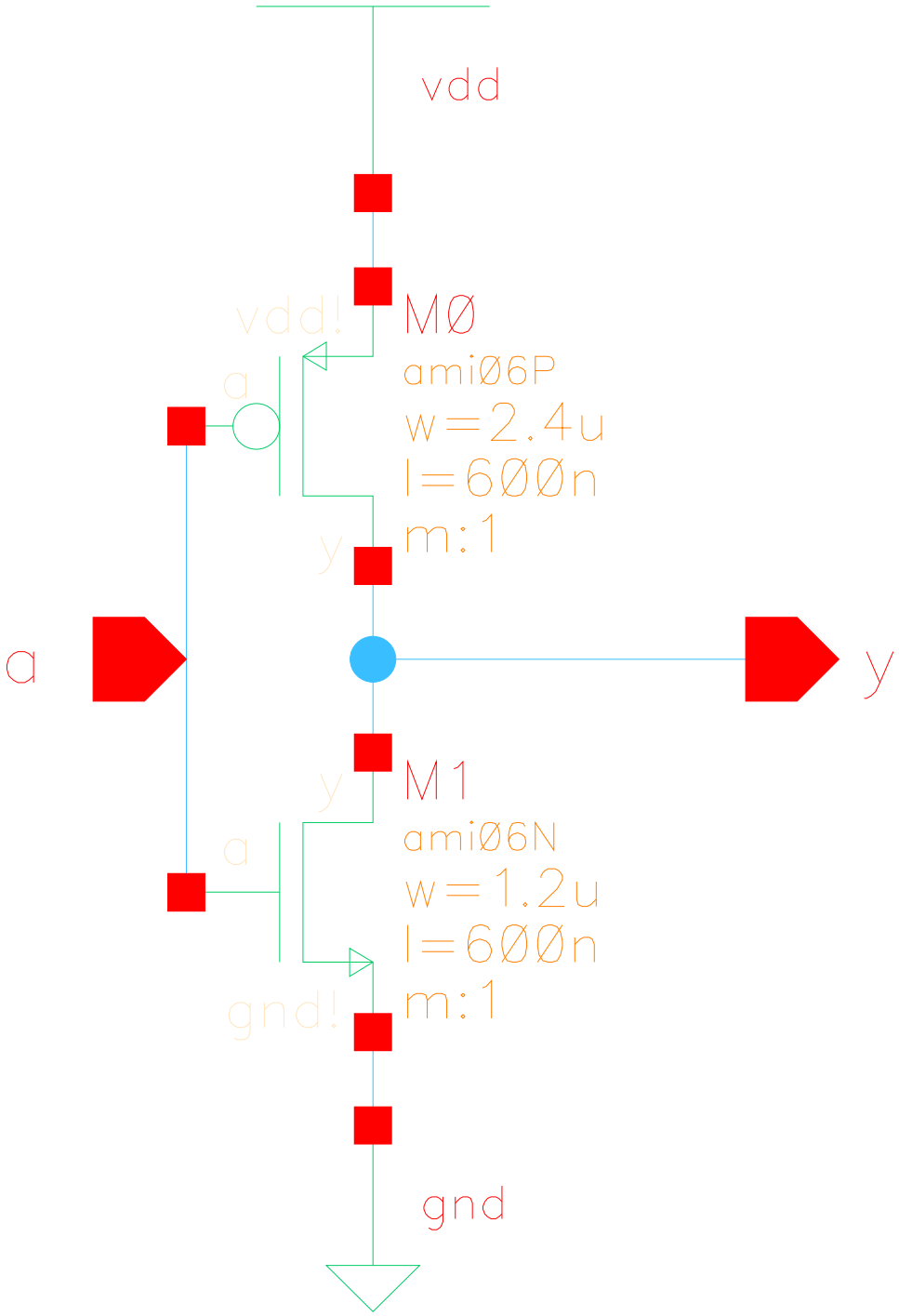
B.10 DST_Block_sch



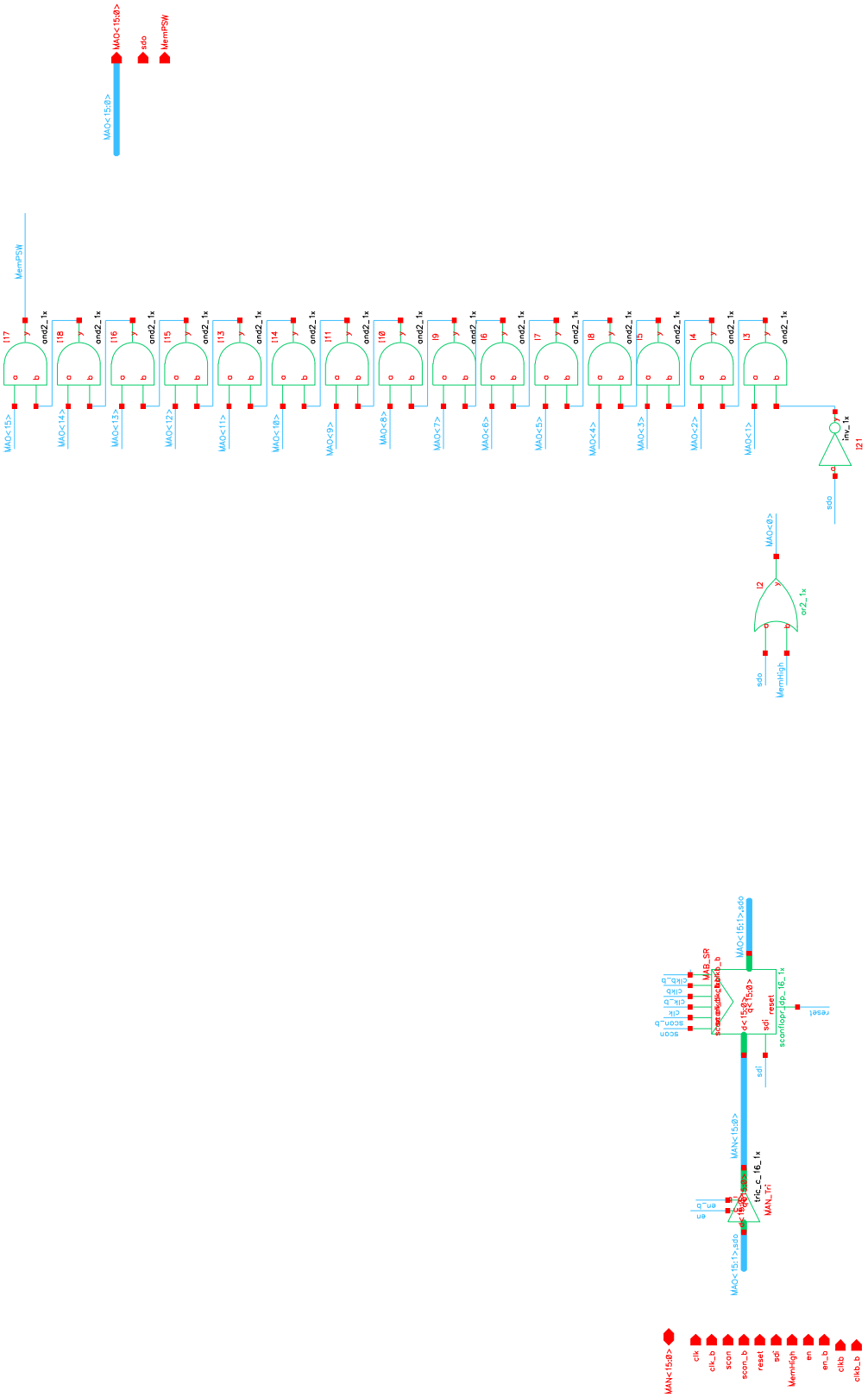
B.11 Instruction_Block_sch



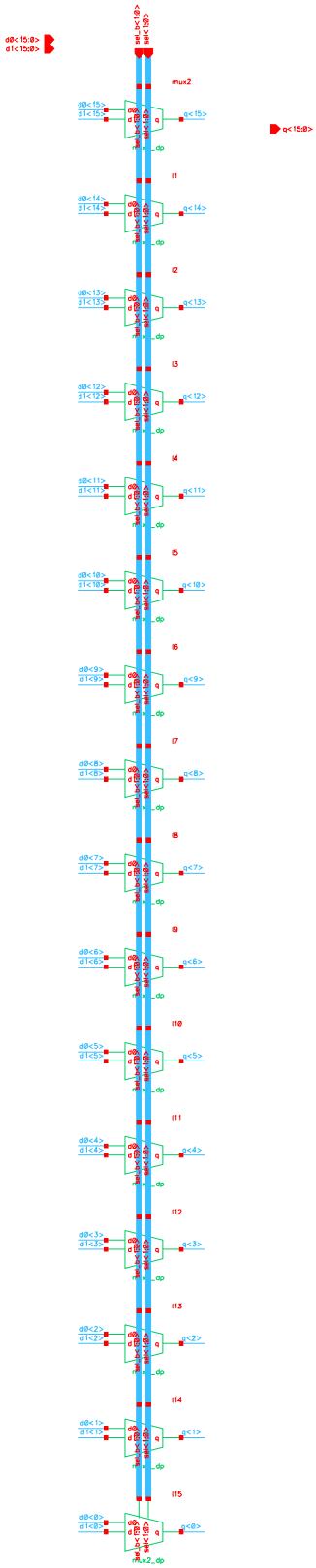
B.12 inv_1x


vdd
vdd!
MØ
amiØ6P
w=2.4u
l=6ØØn
m:1
a
y
a
y
M1
amiØ6N
w=1.2u
l=6ØØn
m:1
a
y
gnd!
gnd

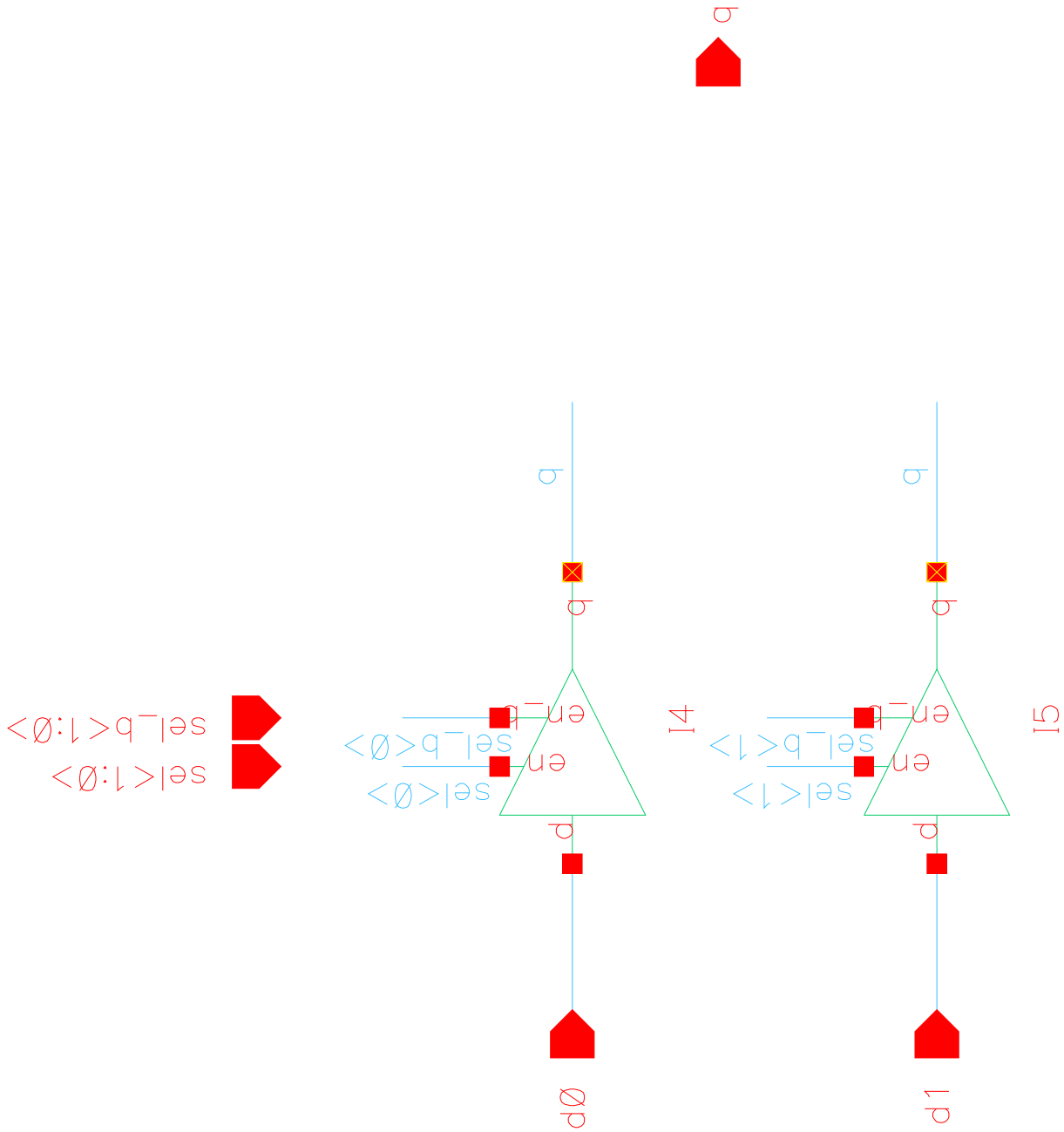
B.13 MemAddr_Block_sch



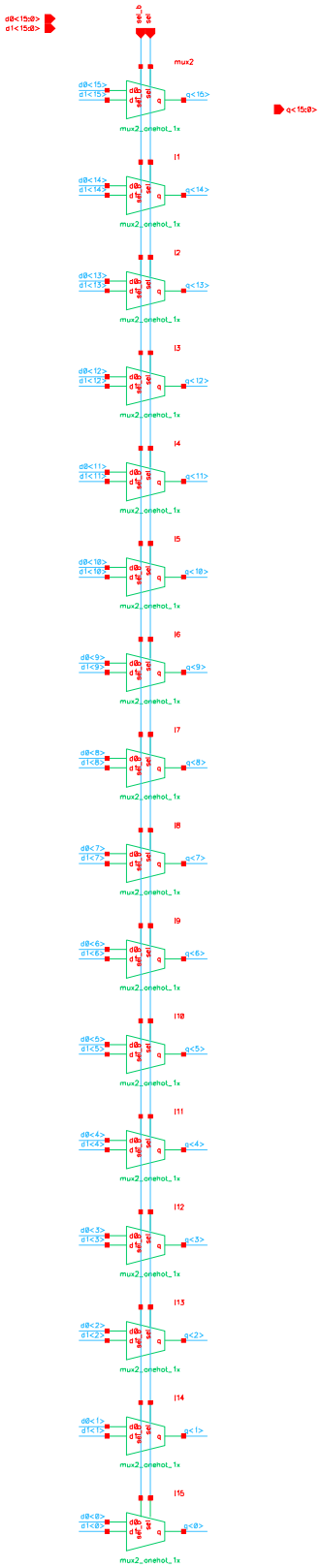
B.14 mux2_16_sch



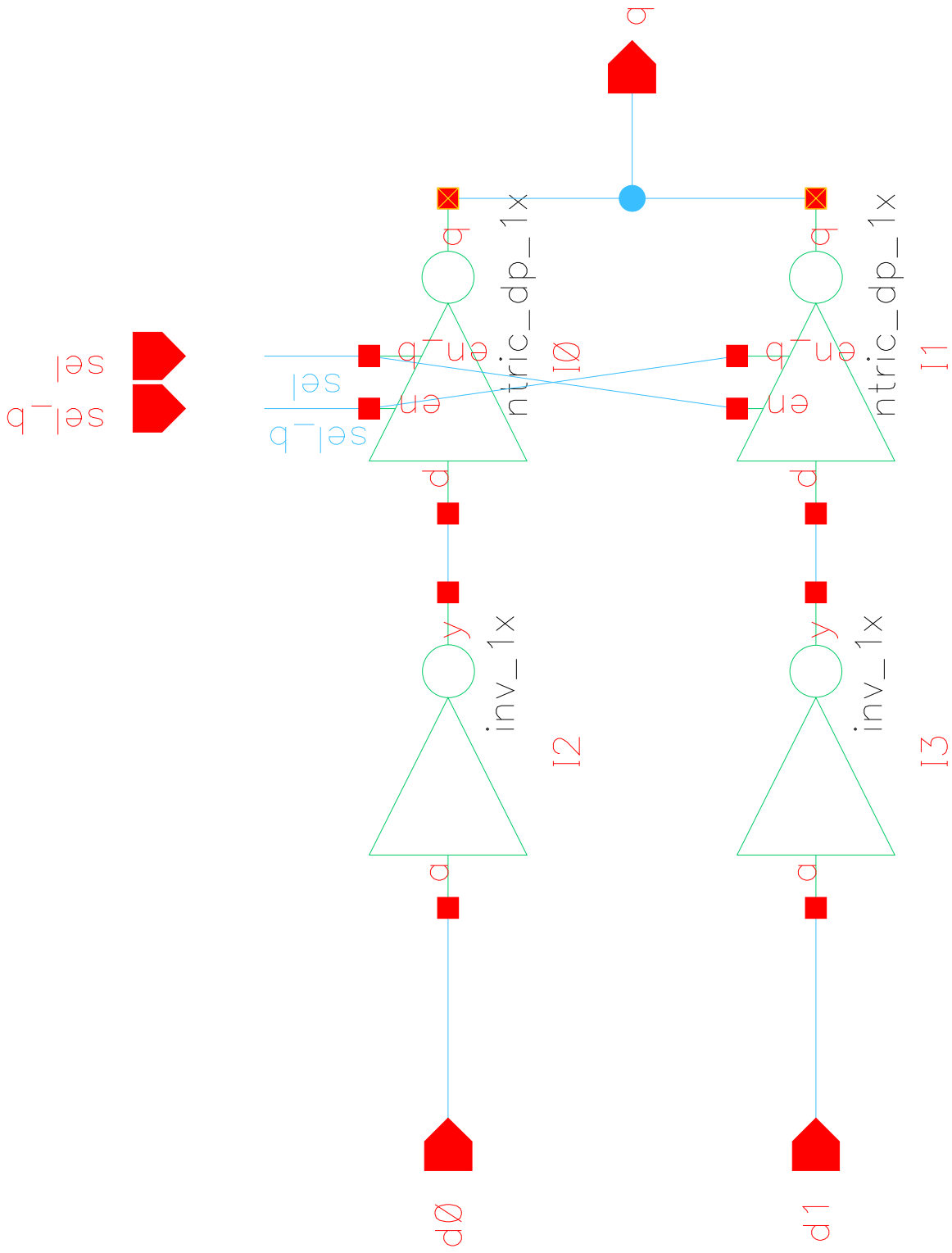
B.15 mux2_1x_sch



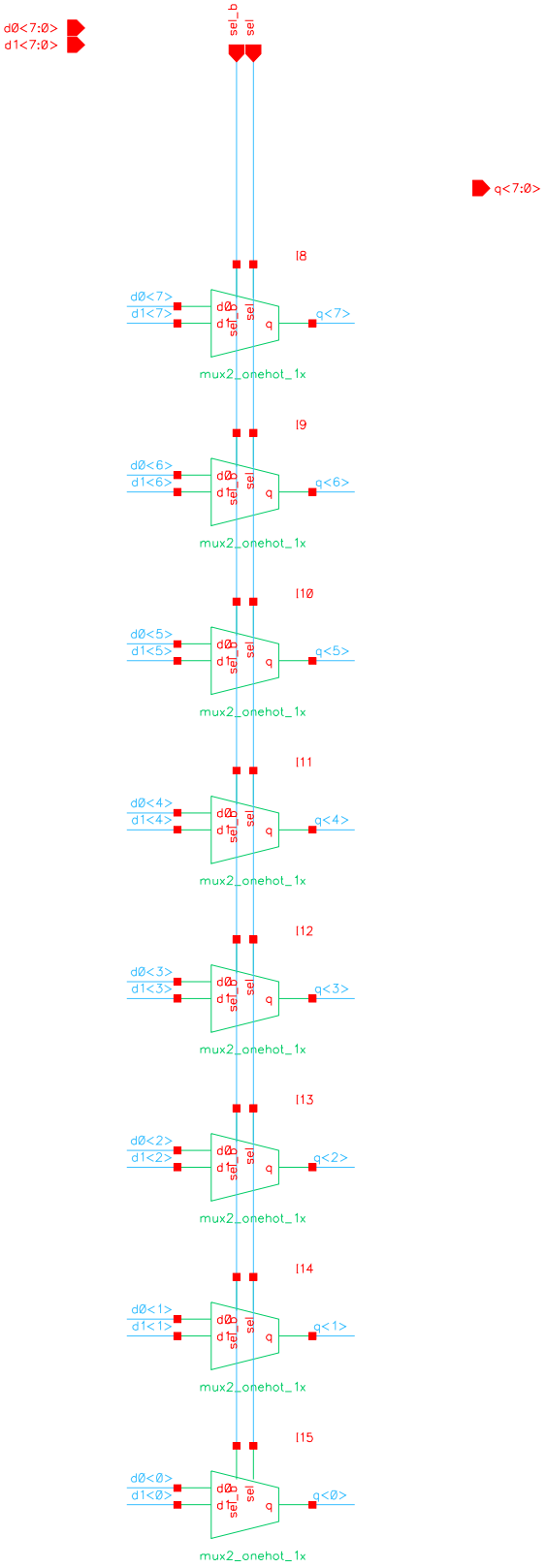
B.16 mux2_onehot_16_sch



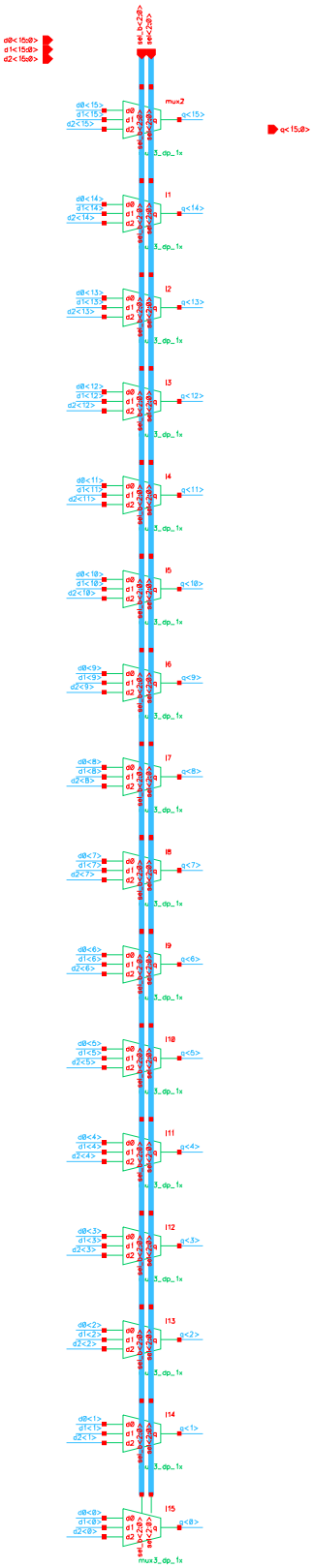
B.17 mux2_onehot_1x_sch



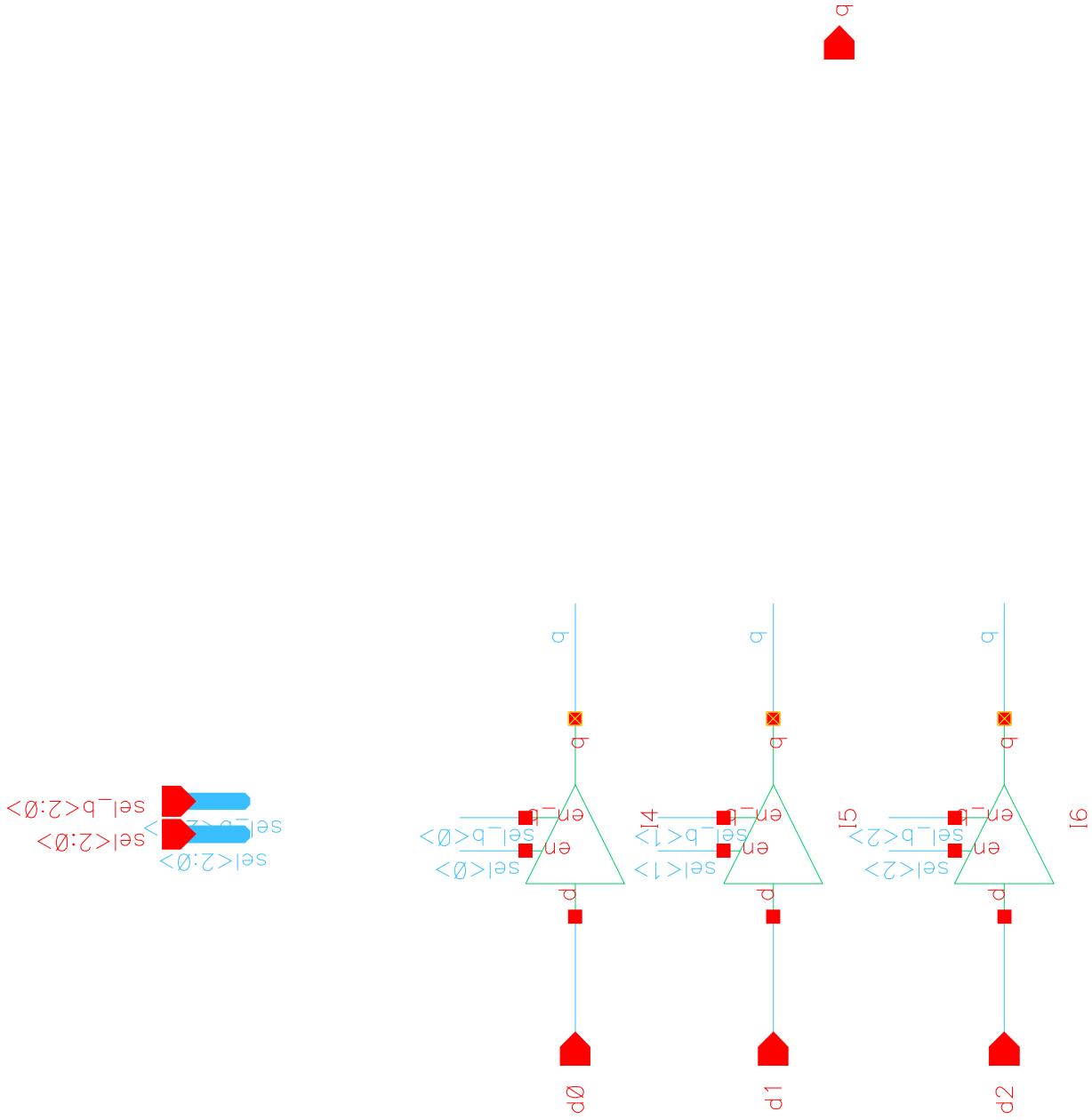
B.18 mux2_onehot_8_sch



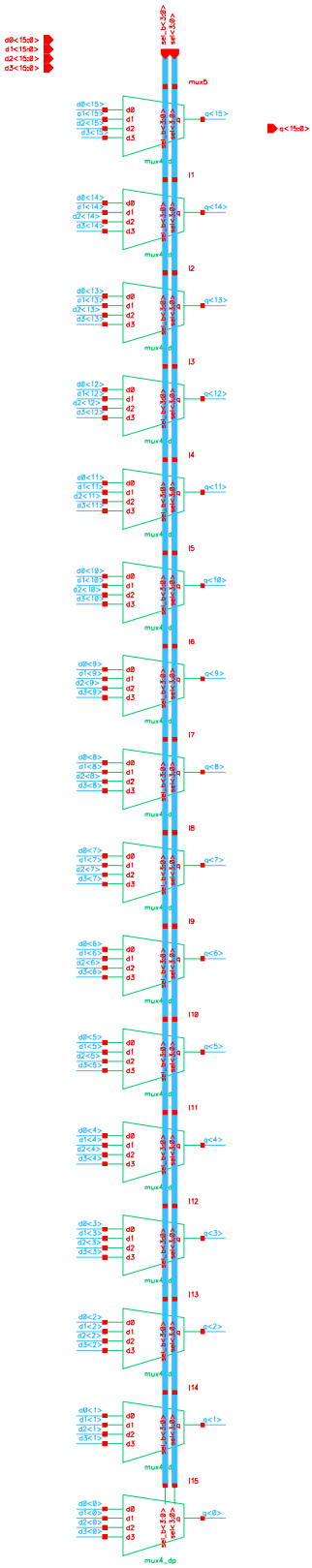
B.19 mux3_16_sch



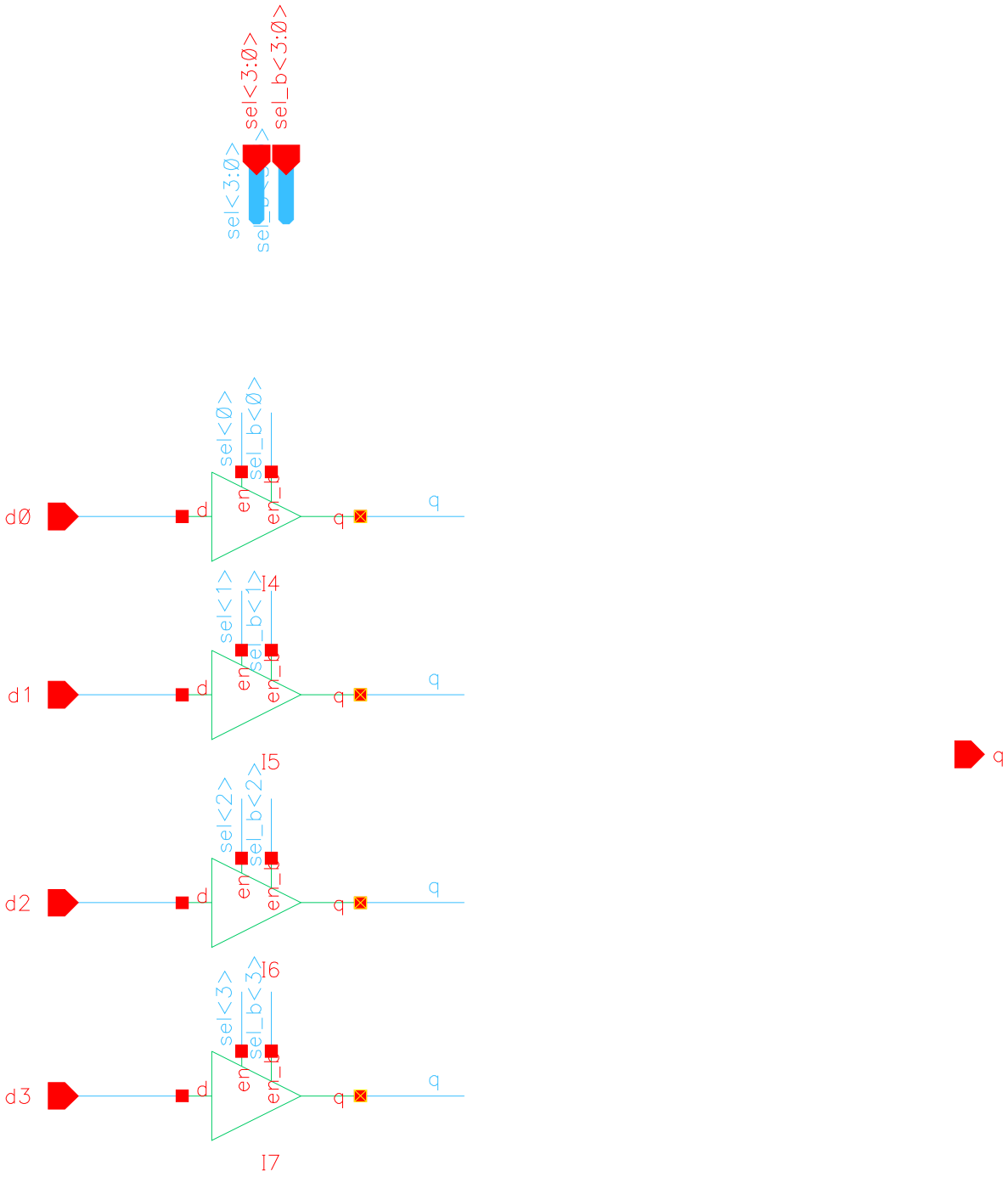
B.20 mux3_1x_sch



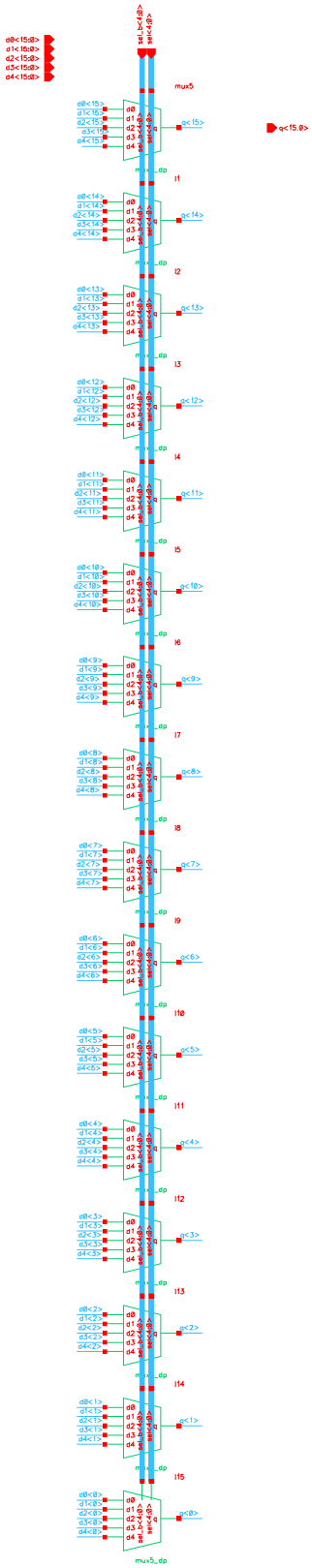
B.21 mux4_16_sch



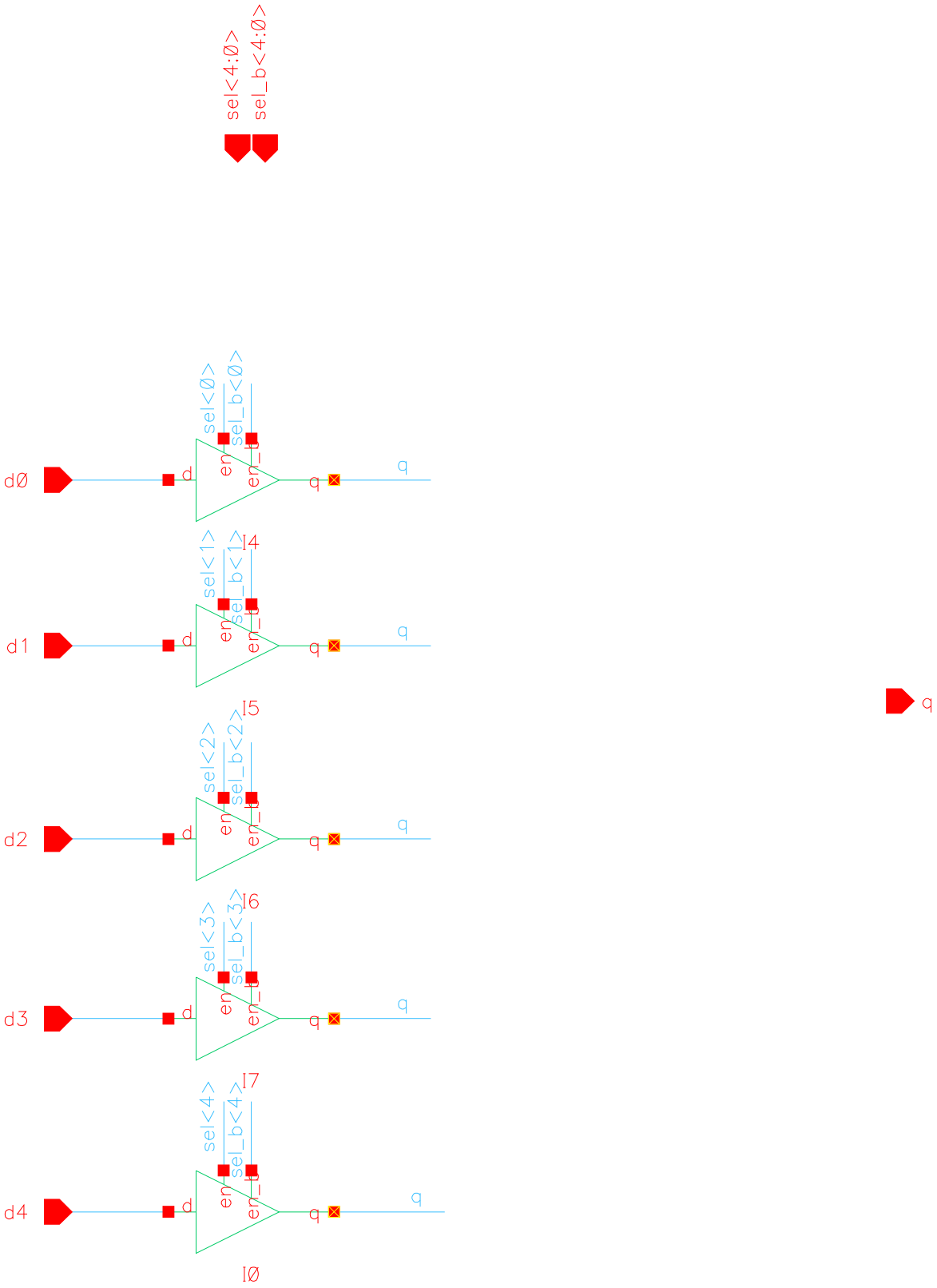
B.22 mux4_1x_sch



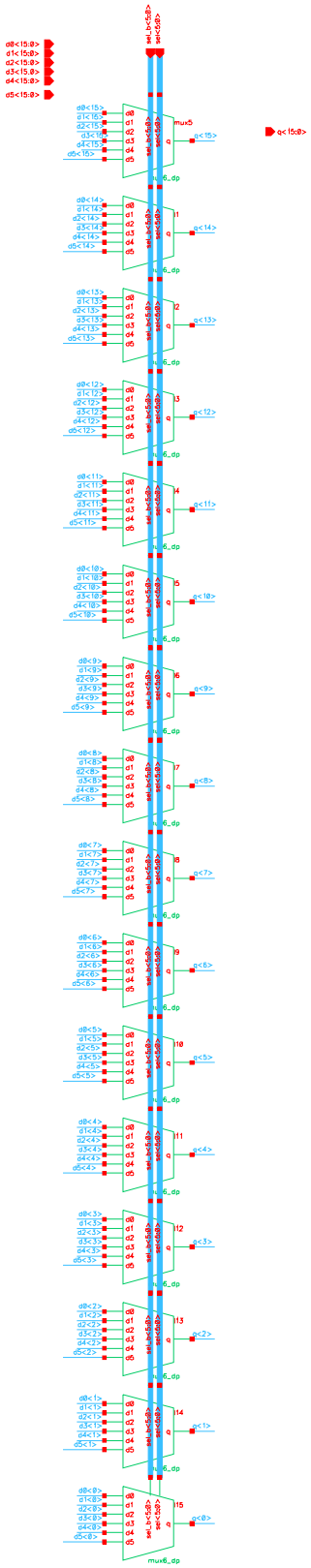
B.23 mux5_16_sch



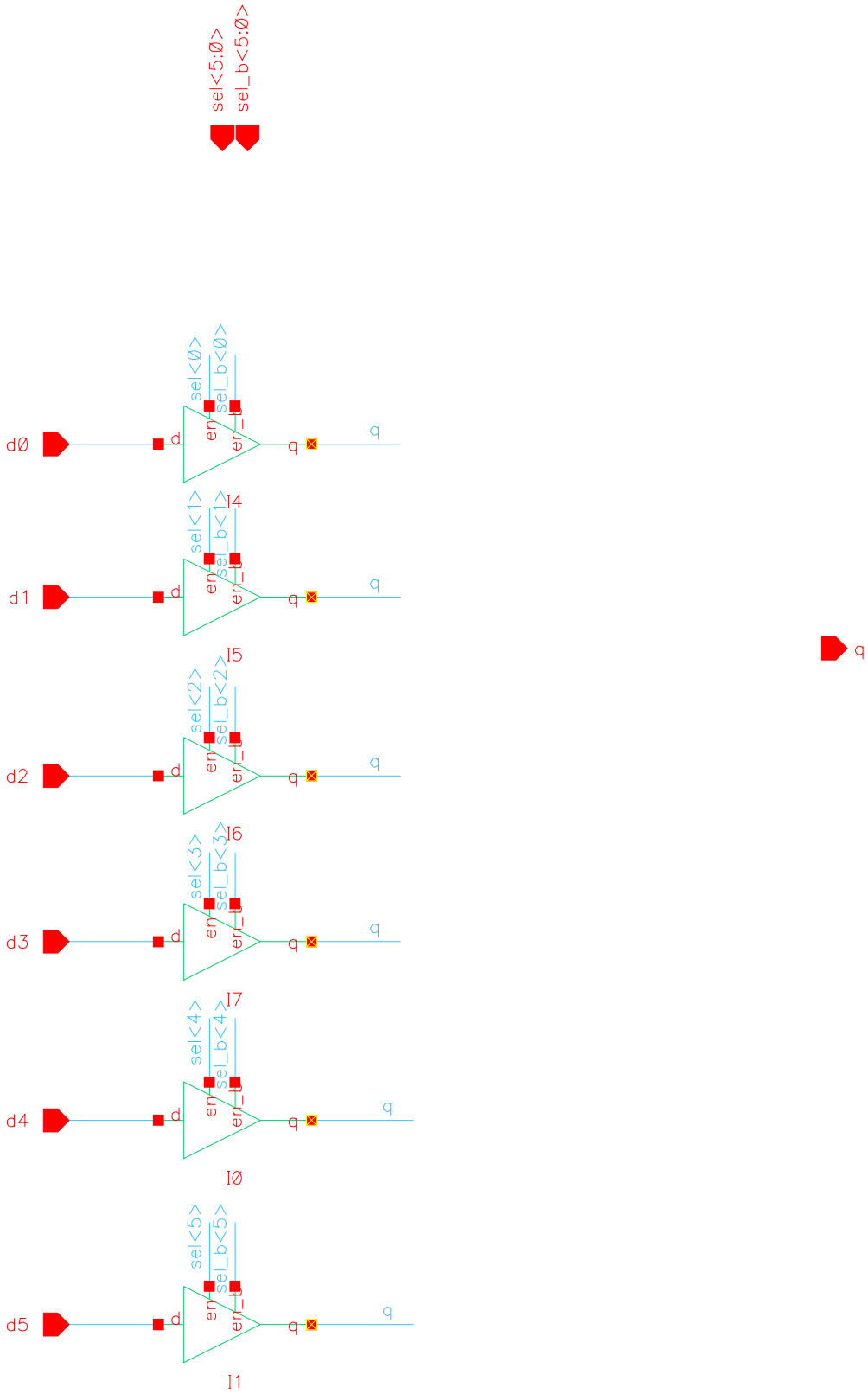
B.24 mux5_1x_sch



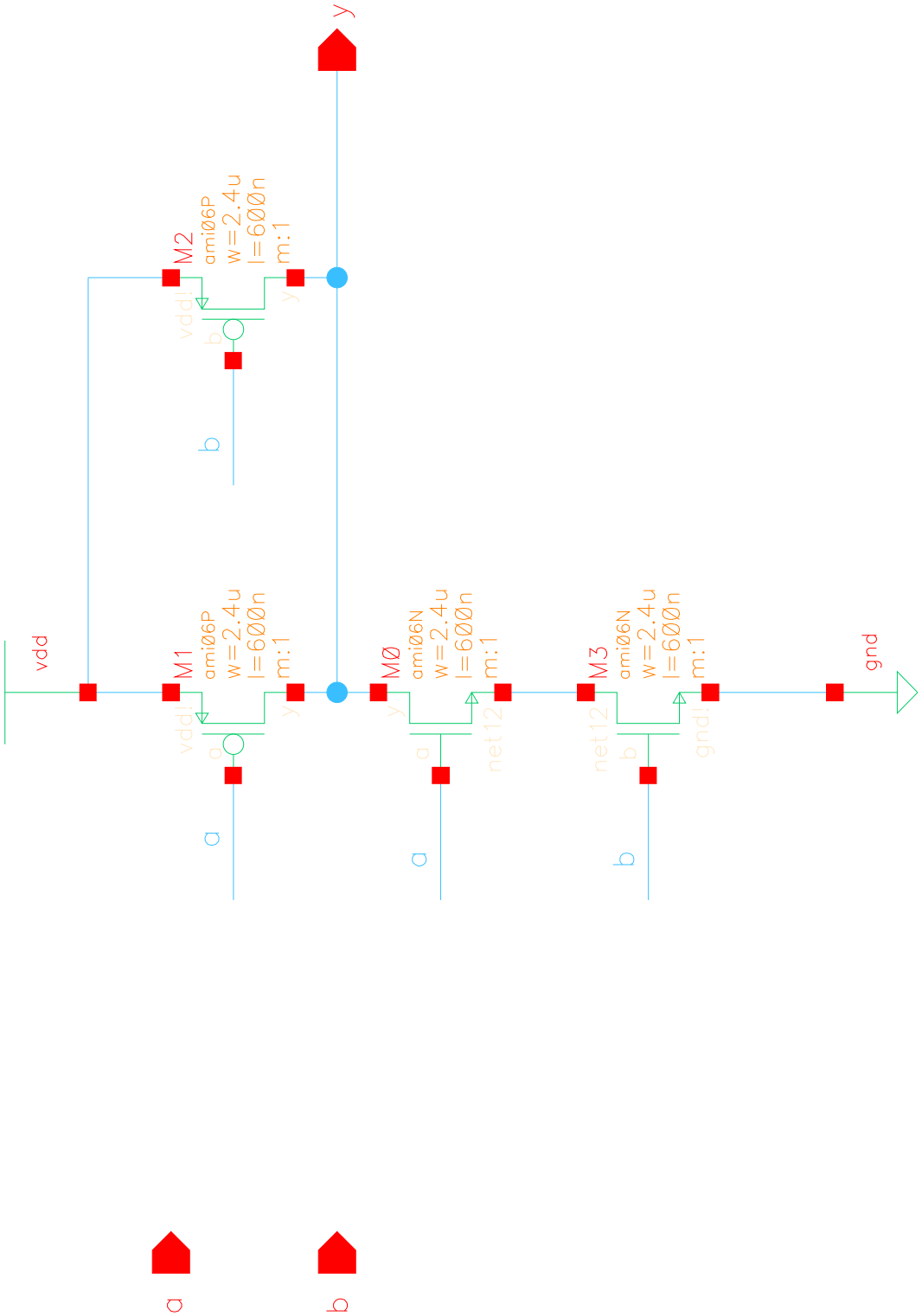
B.25 mux6_16_sch



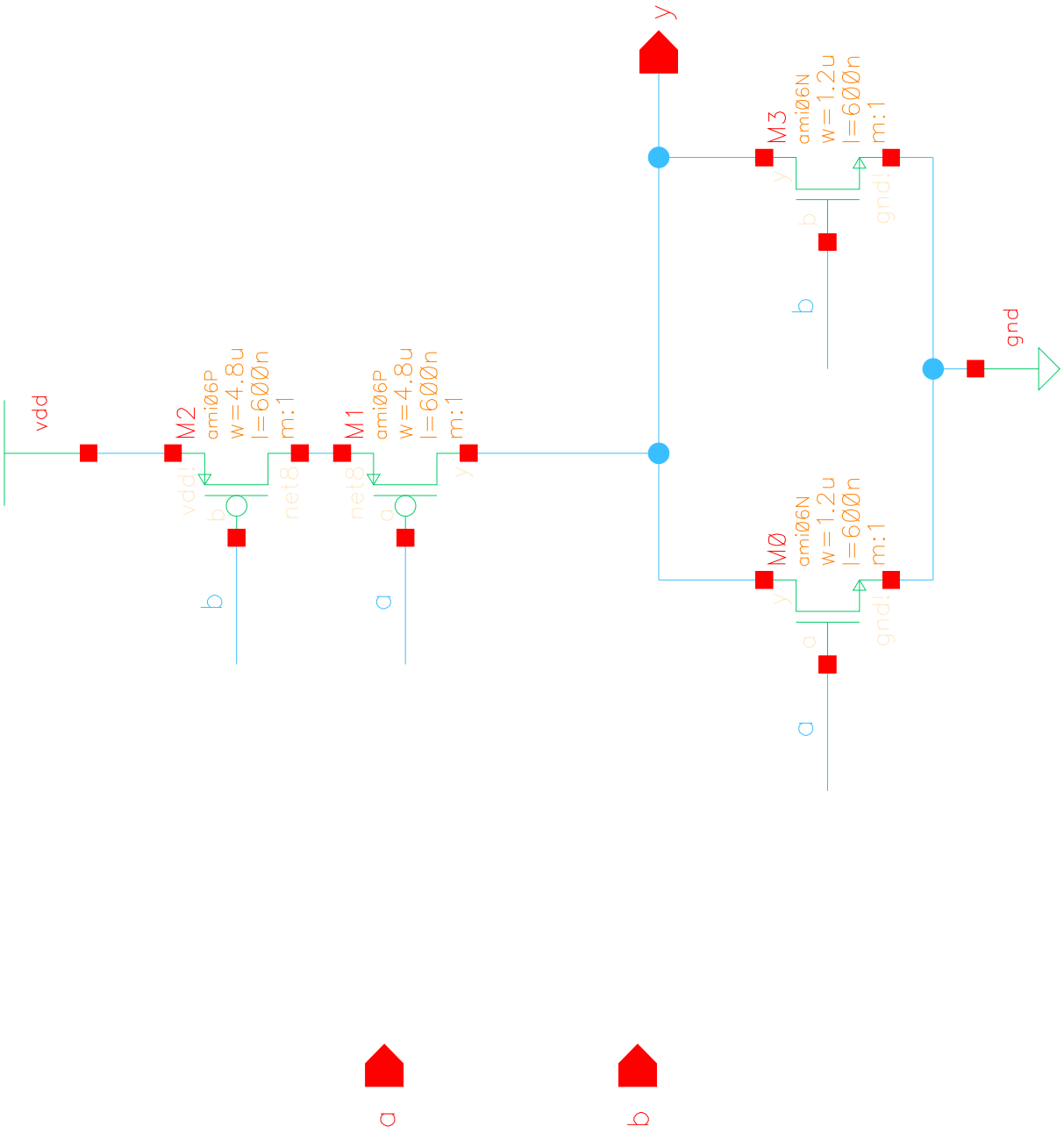
B.26 mux6_1x_sch



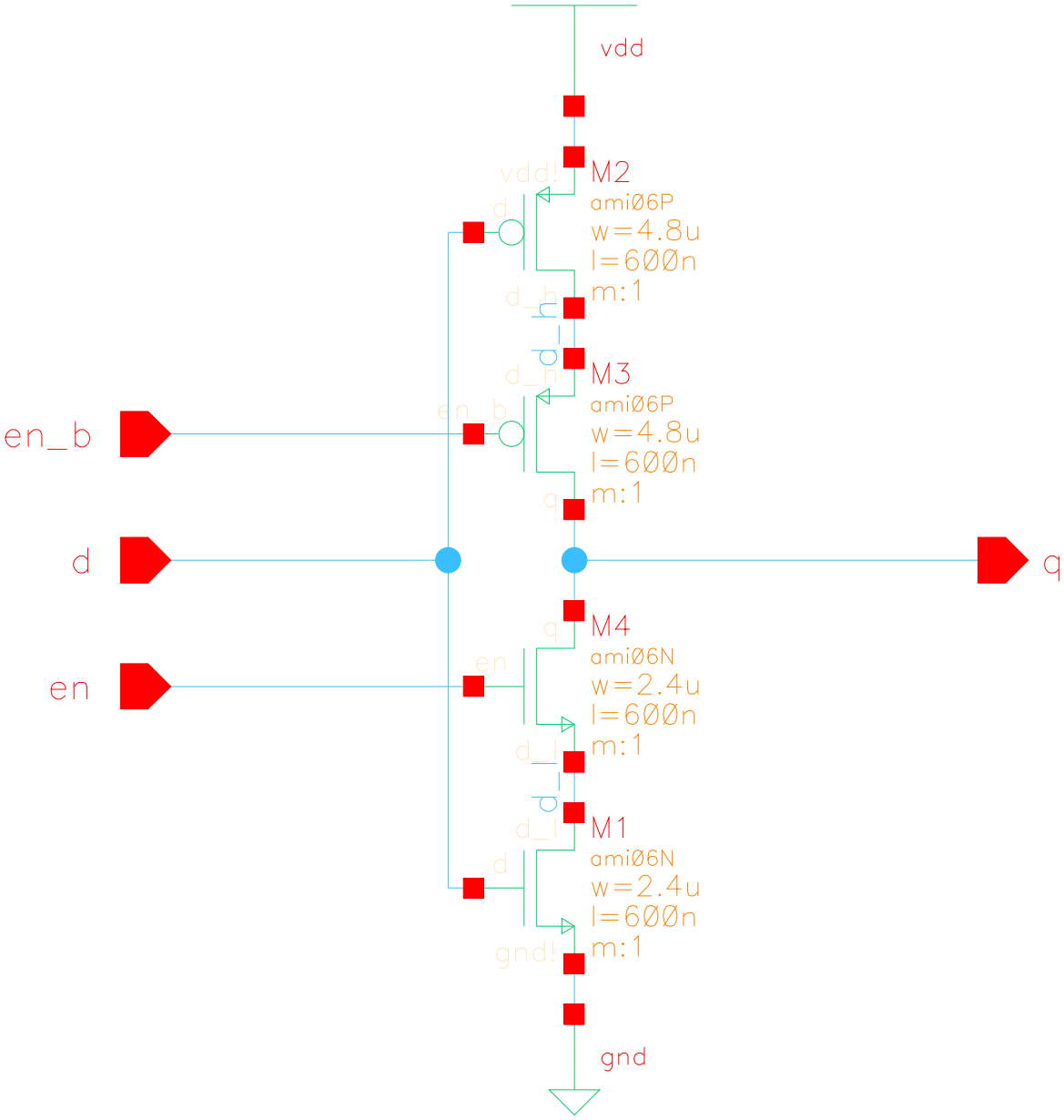
B.27 nand2_1x


y
M2
ami06P
w=2.4u
l=600n
m:1
vdd!
b
y
b
vdd
M1
ami06P
w=2.4u
l=600n
m:1
vdd!
a
y
a
M0
ami06N
w=2.4u
l=600n
m:1
y
a
net12
net12
M3
ami06N
w=2.4u
l=600n
m:1
b
gnd!
b
gnd
a
b

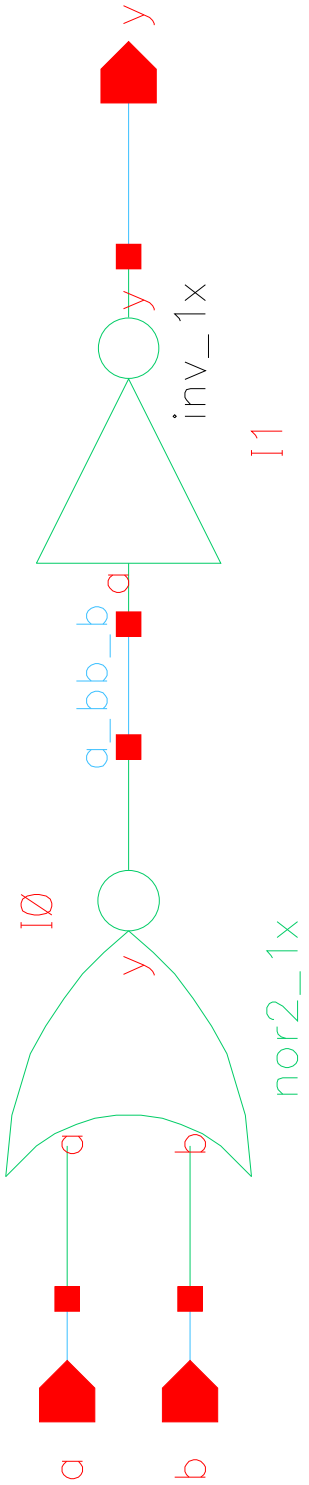
B.28 nor2_1x


y
M3
ami06N
w=1.2u
l=600n
m:1
gnd!
b
b
gnd
vdd
M2
ami06P
w=4.8u
l=600n
m:1
vdd!
b
net8
M1
ami06P
w=4.8u
l=600n
m:1
net8
a
y
b
a
M0
ami06N
w=1.2u
l=600n
m:1
gnd!
a
a
b

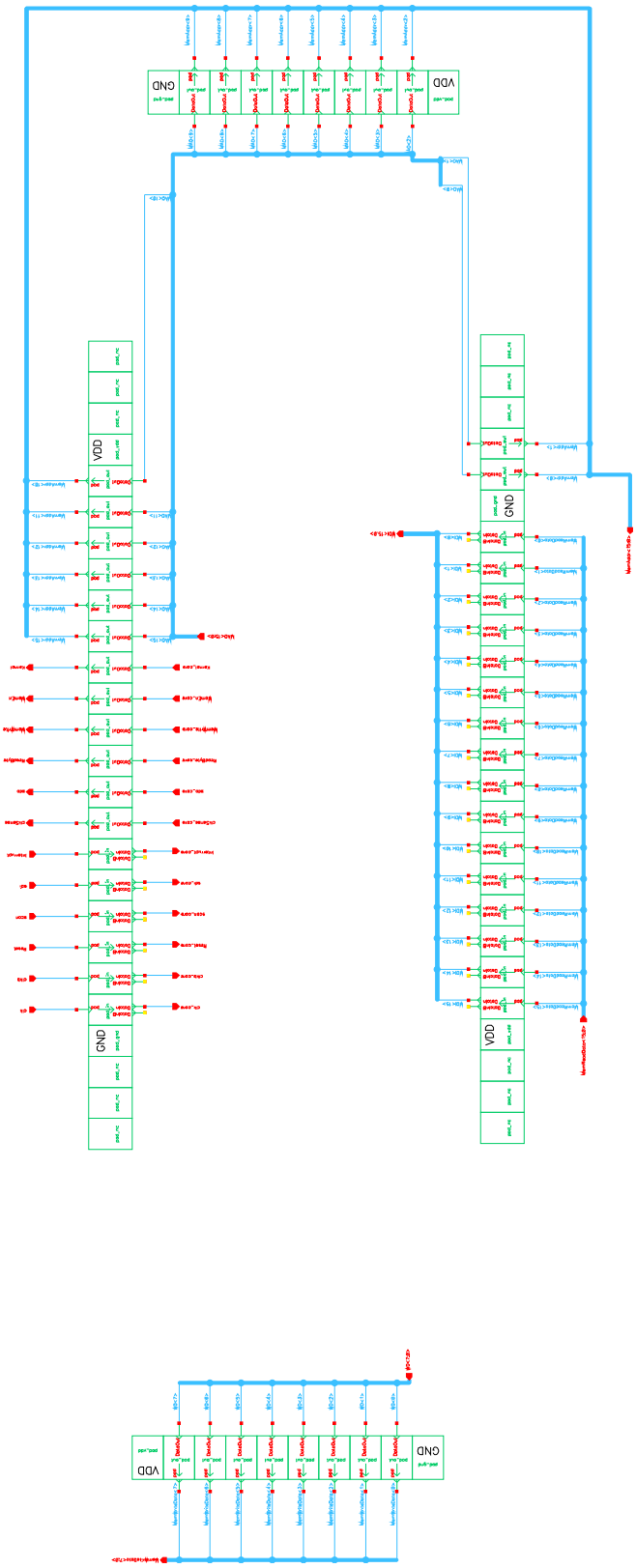
B.29 ntri_dp_1x



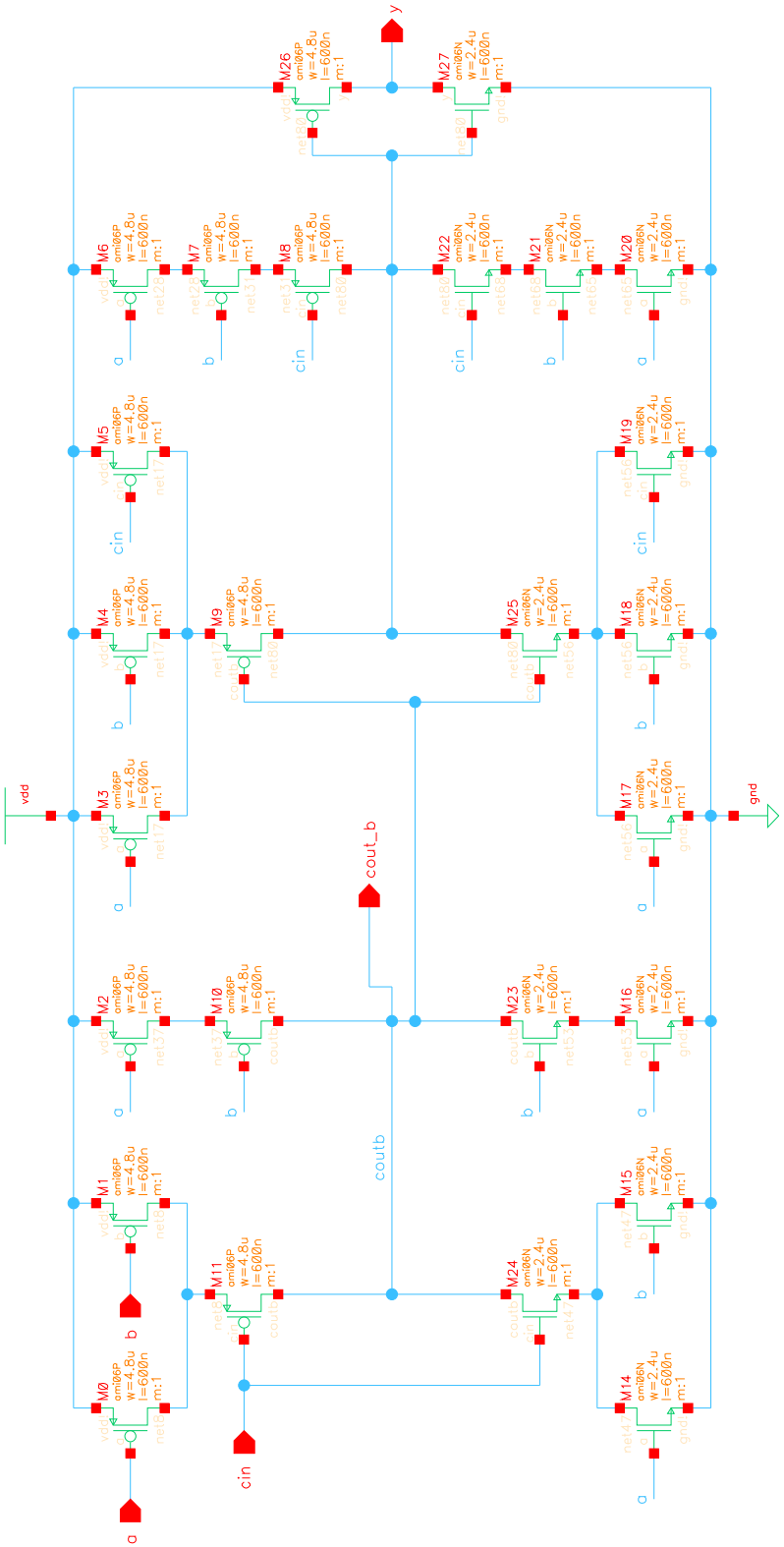
B.30 or2_1x



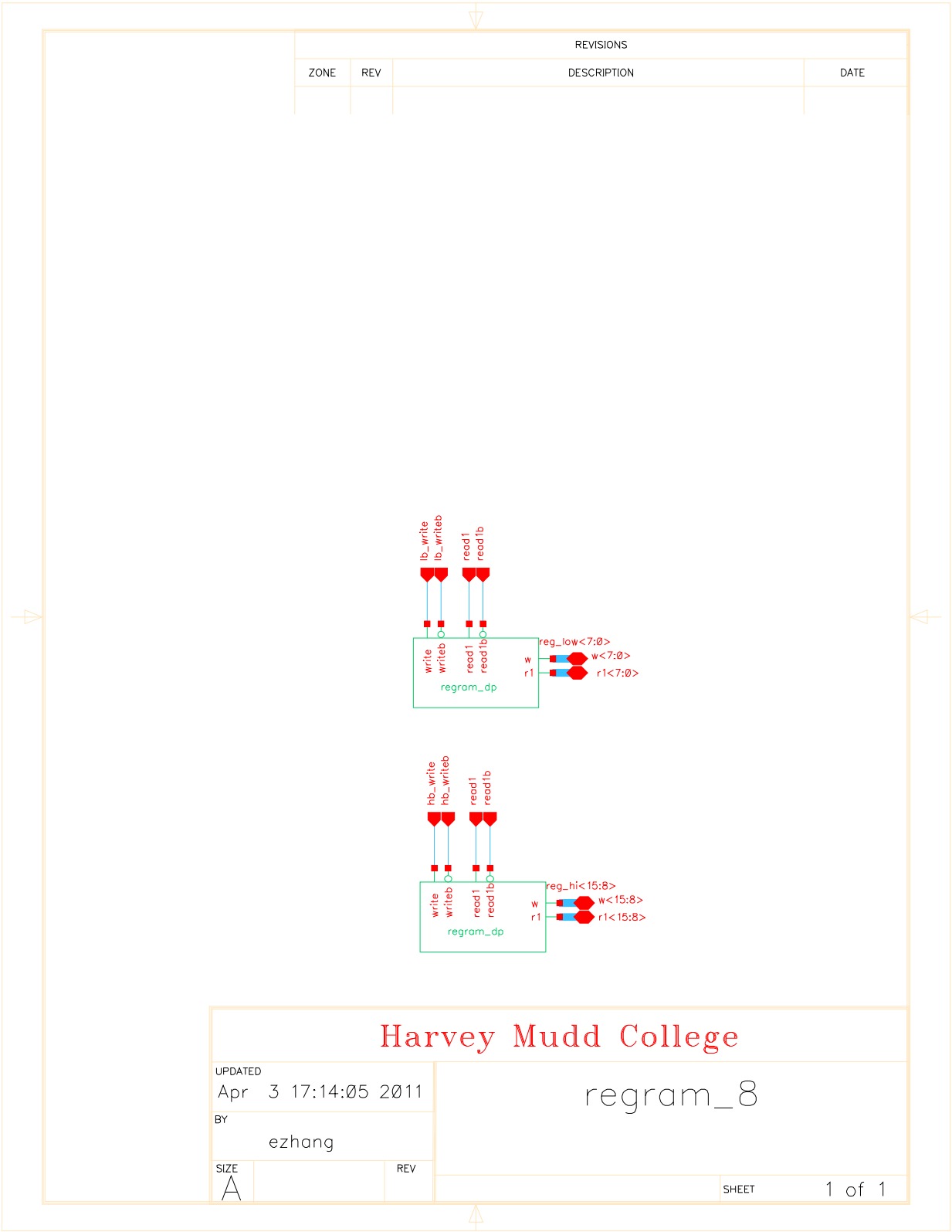
B.31 padframes_full



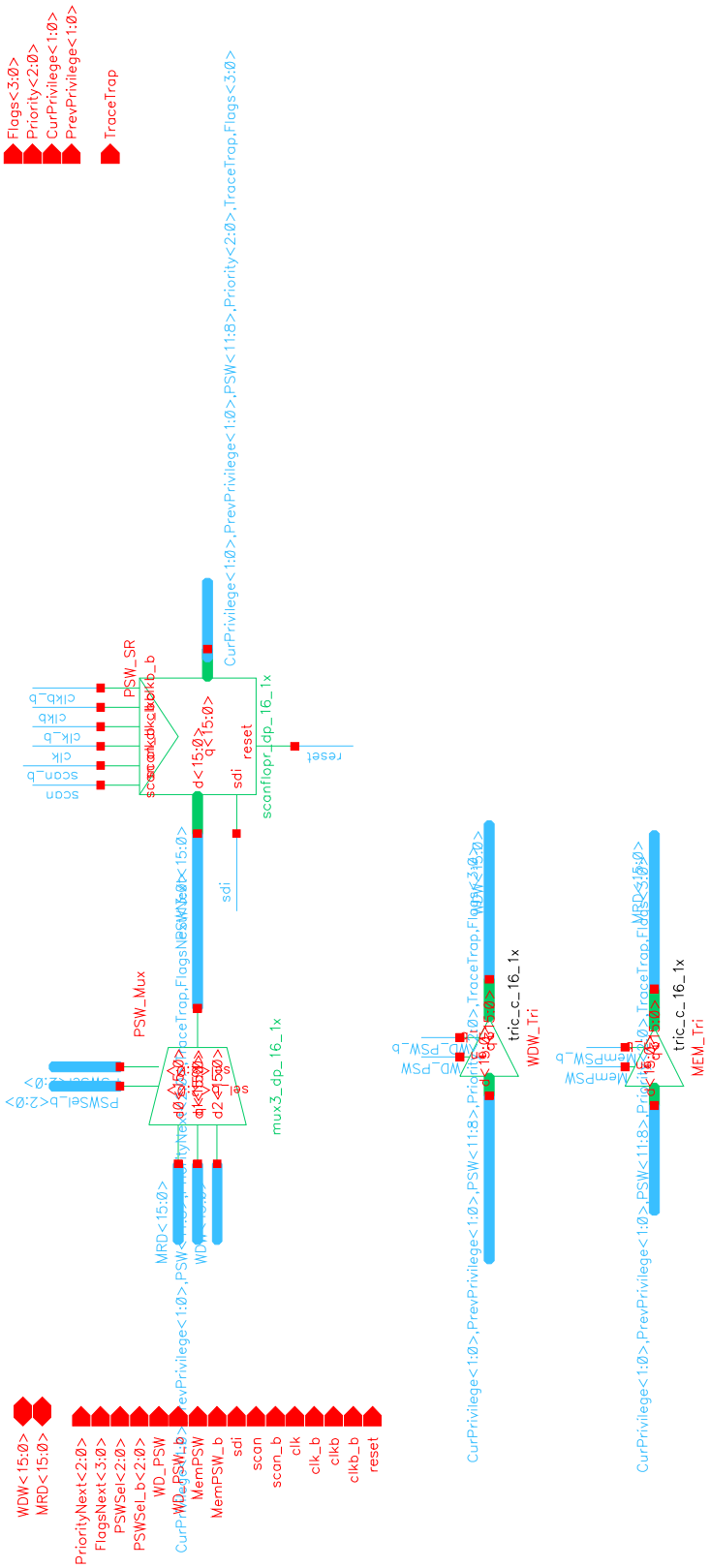
B.32 pdp11_full_add_1x



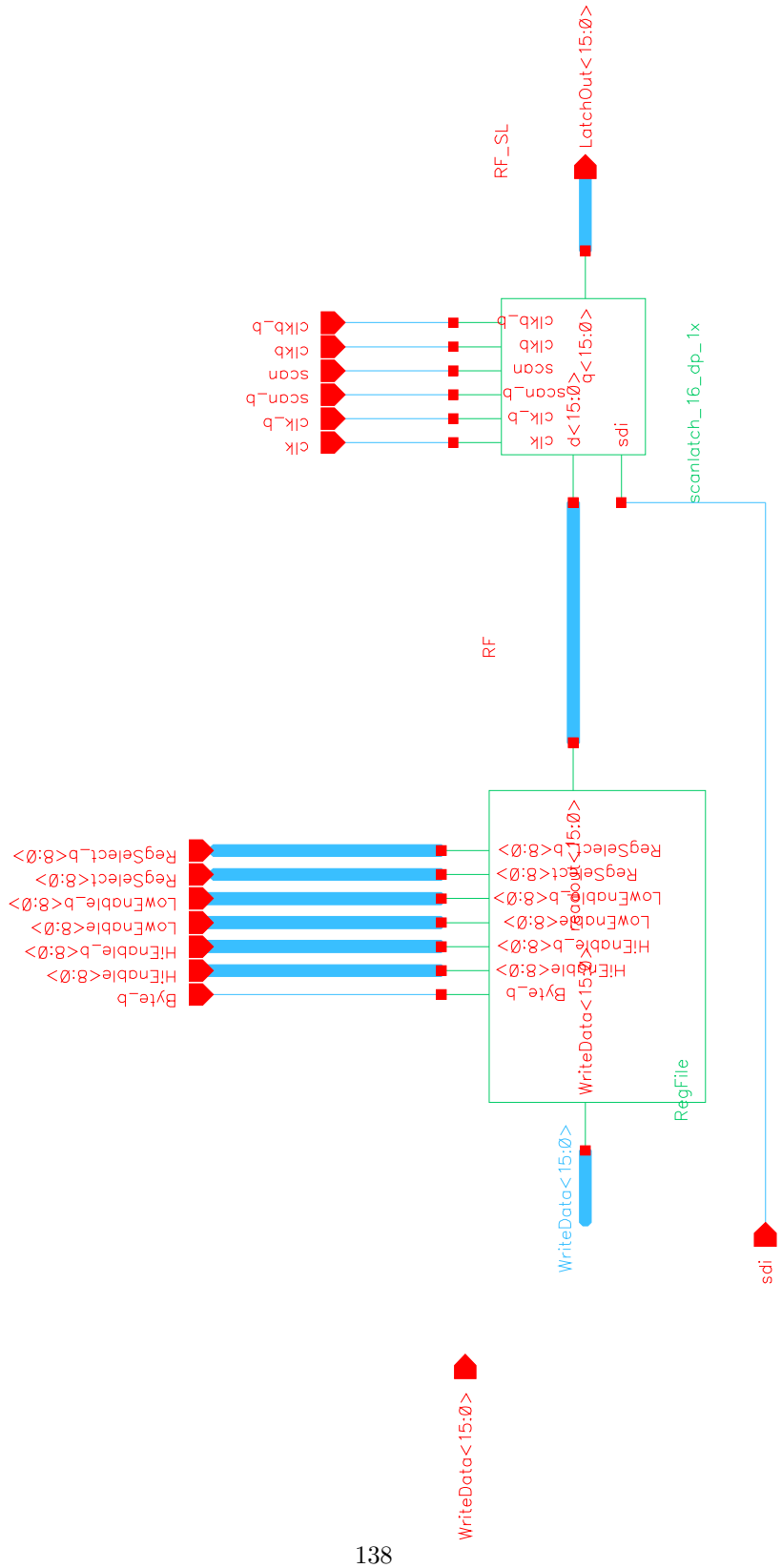
B.33 pdpreg16_sch



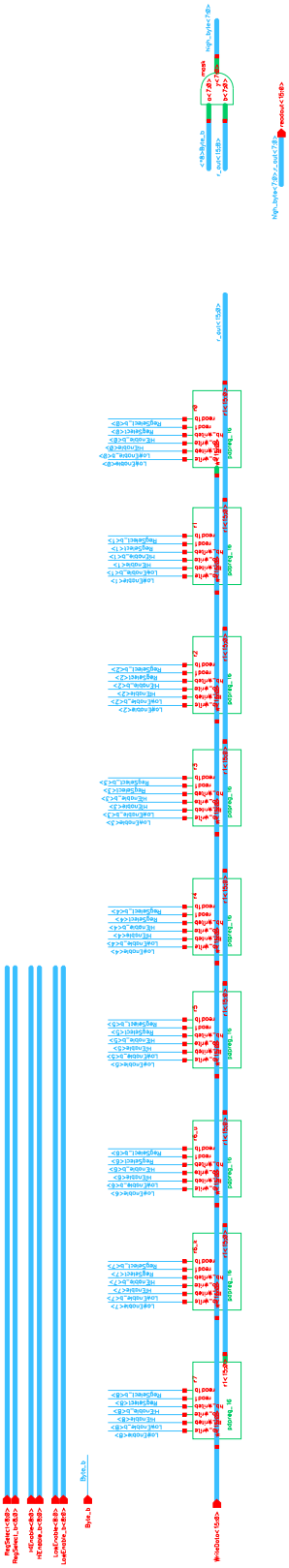
B.34 PSW_Block_sch



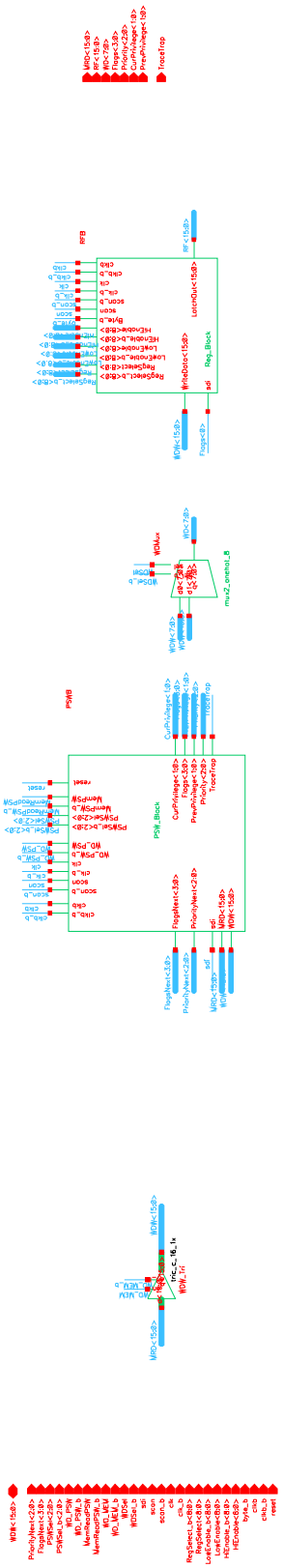
B.35 Reg_Block_sch



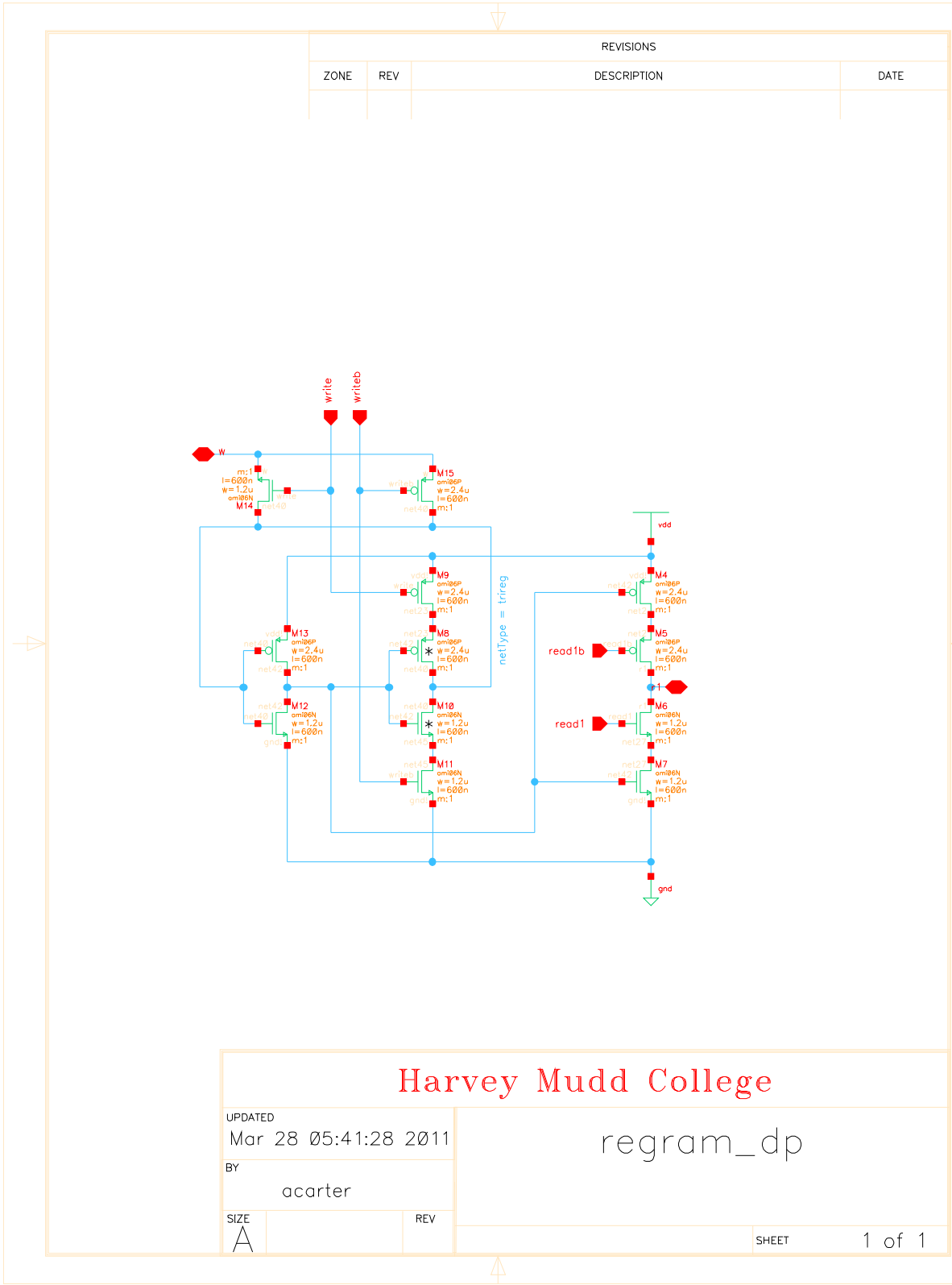
B.36 RegFile_sch



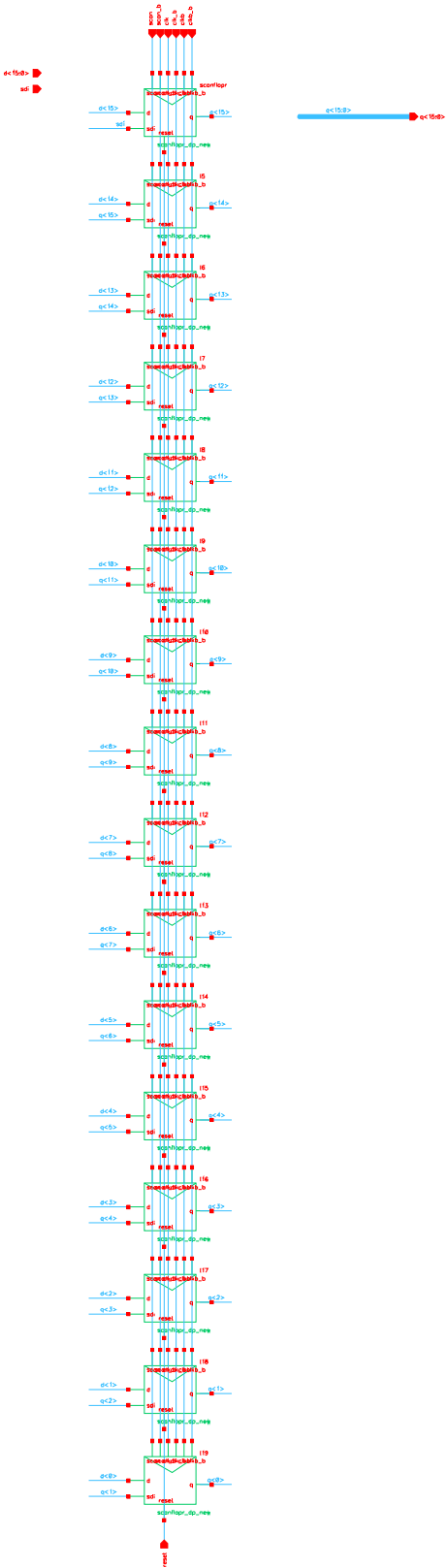
B.37 RegPSW_Block_sch



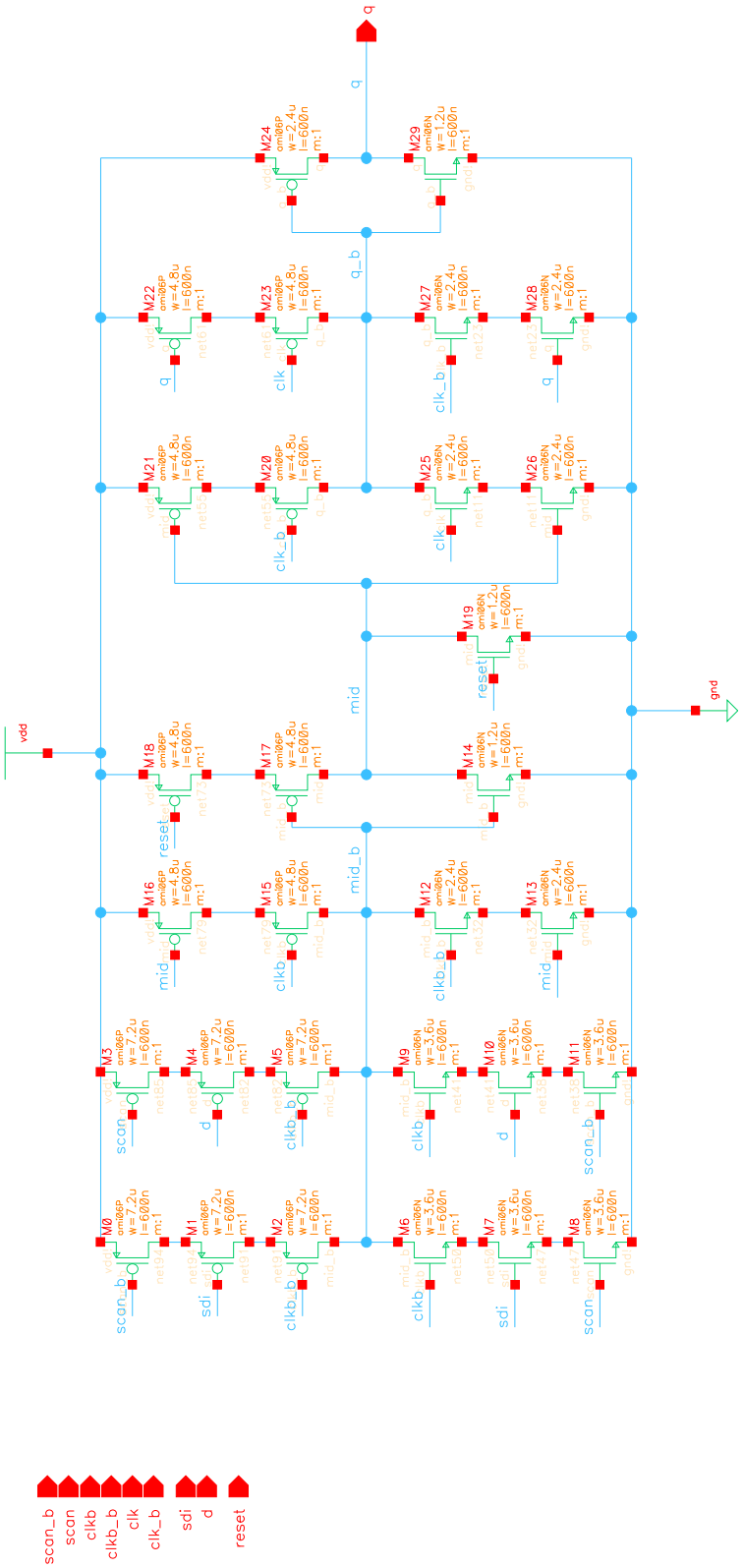
B.38 regram_dp



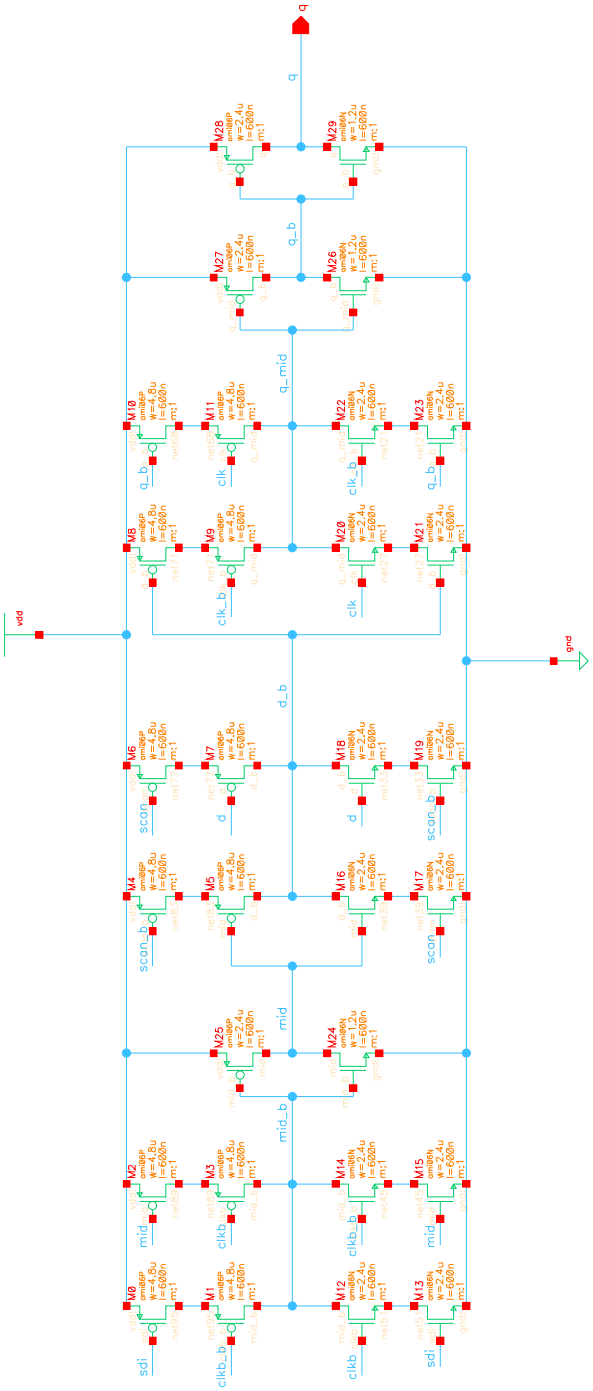
B.39 scanflopr_16_dp_1x



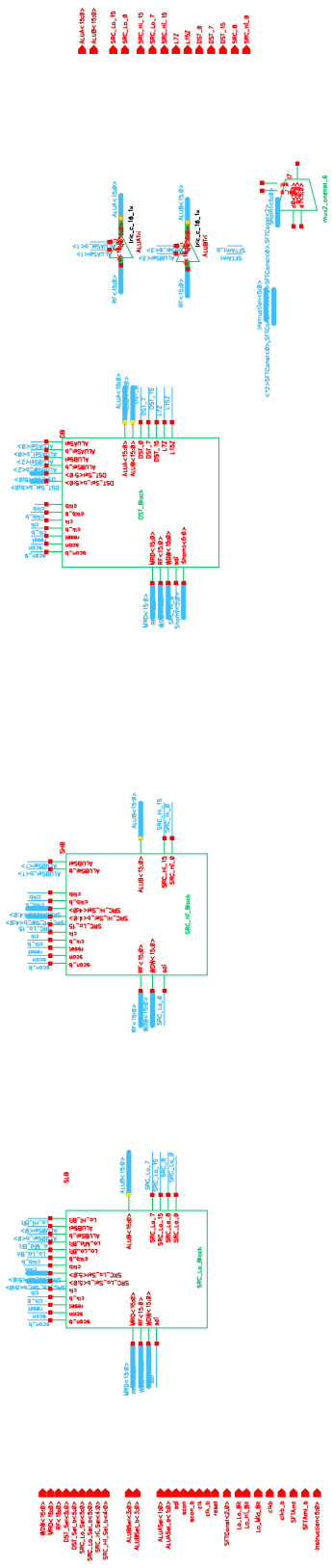
B.40 scanflopr_dp_1x



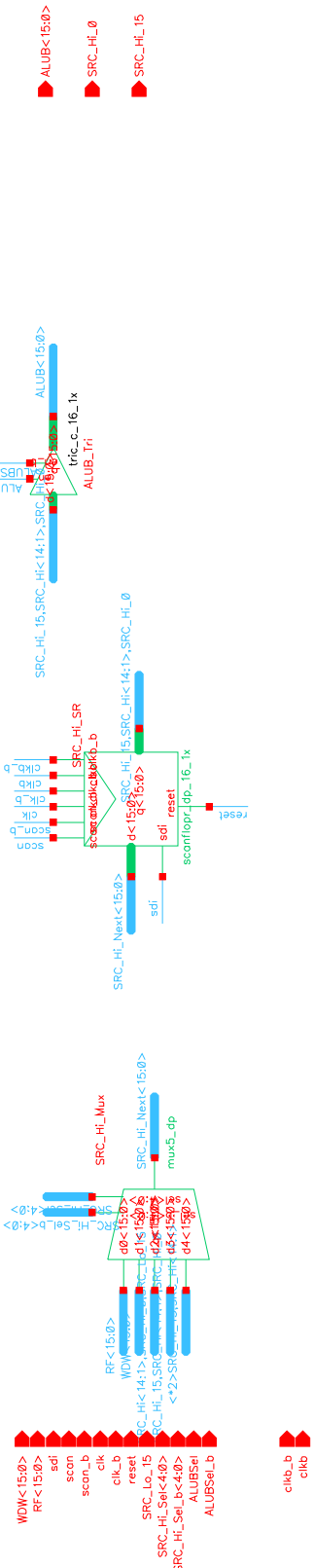
B.41 scanlatch_dp_1x



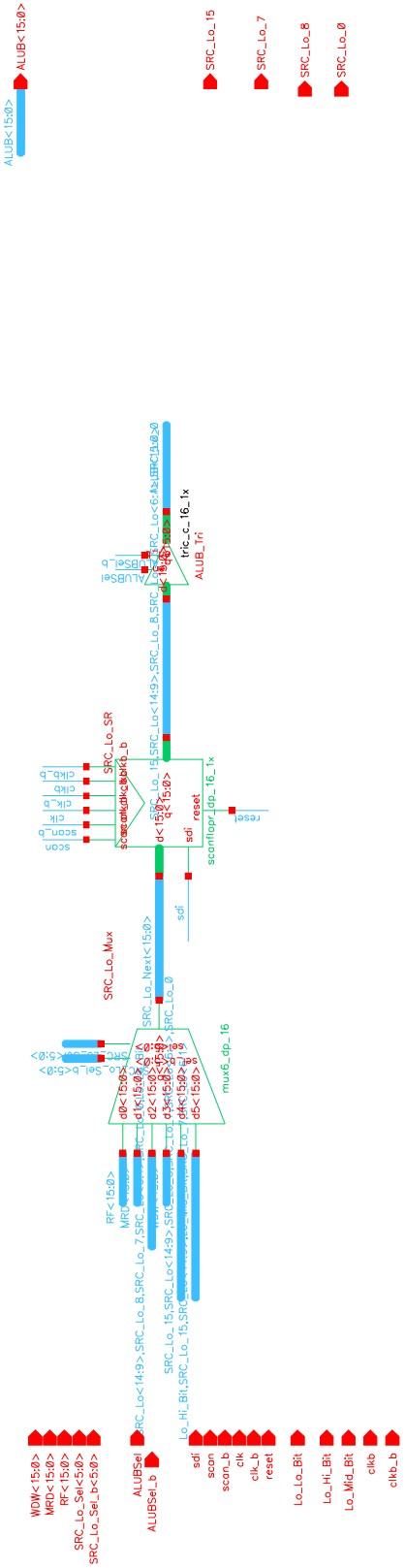
B.42 SD_Block_sch



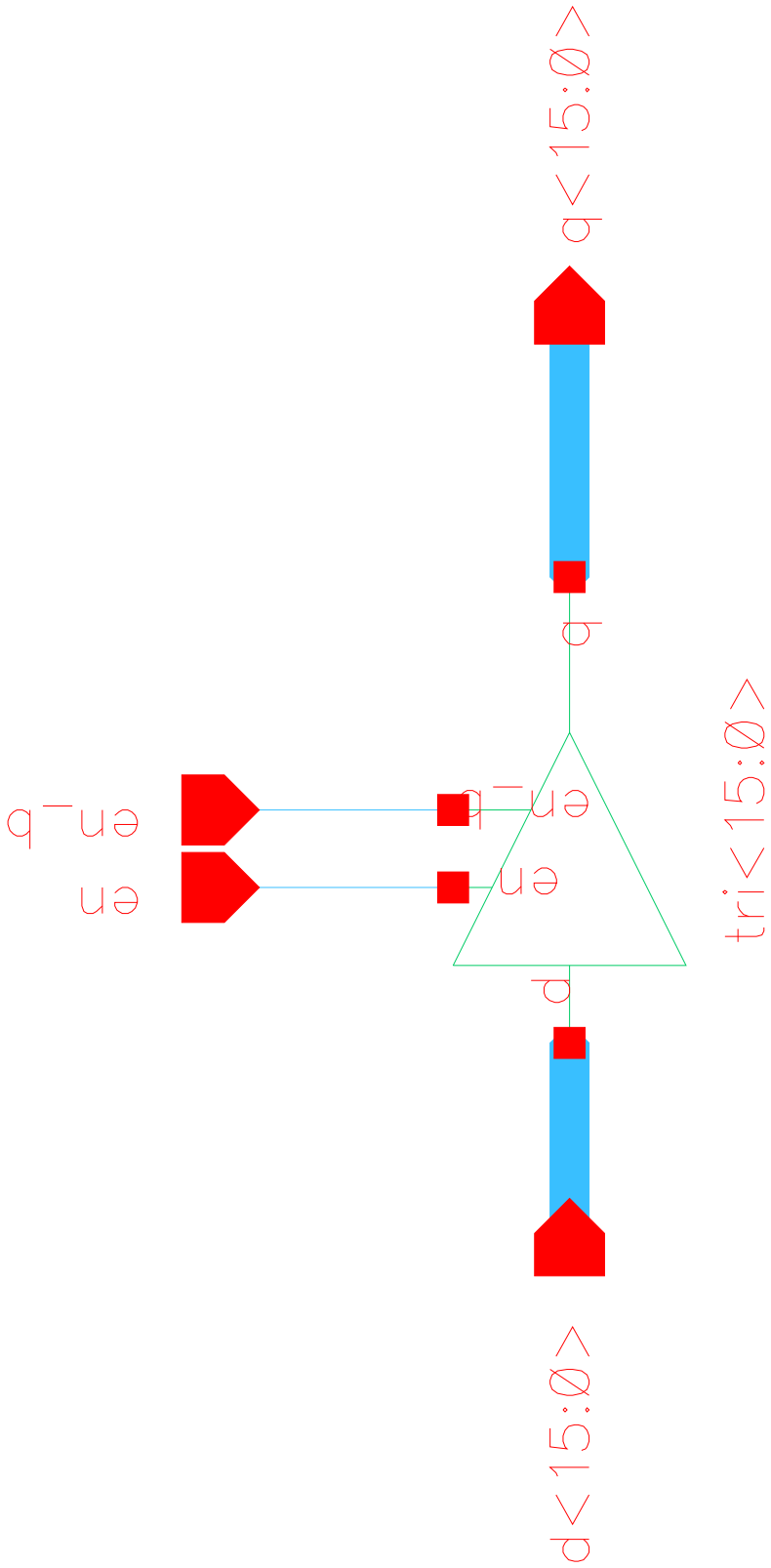
B.43 SRC_Hi_Block_sch



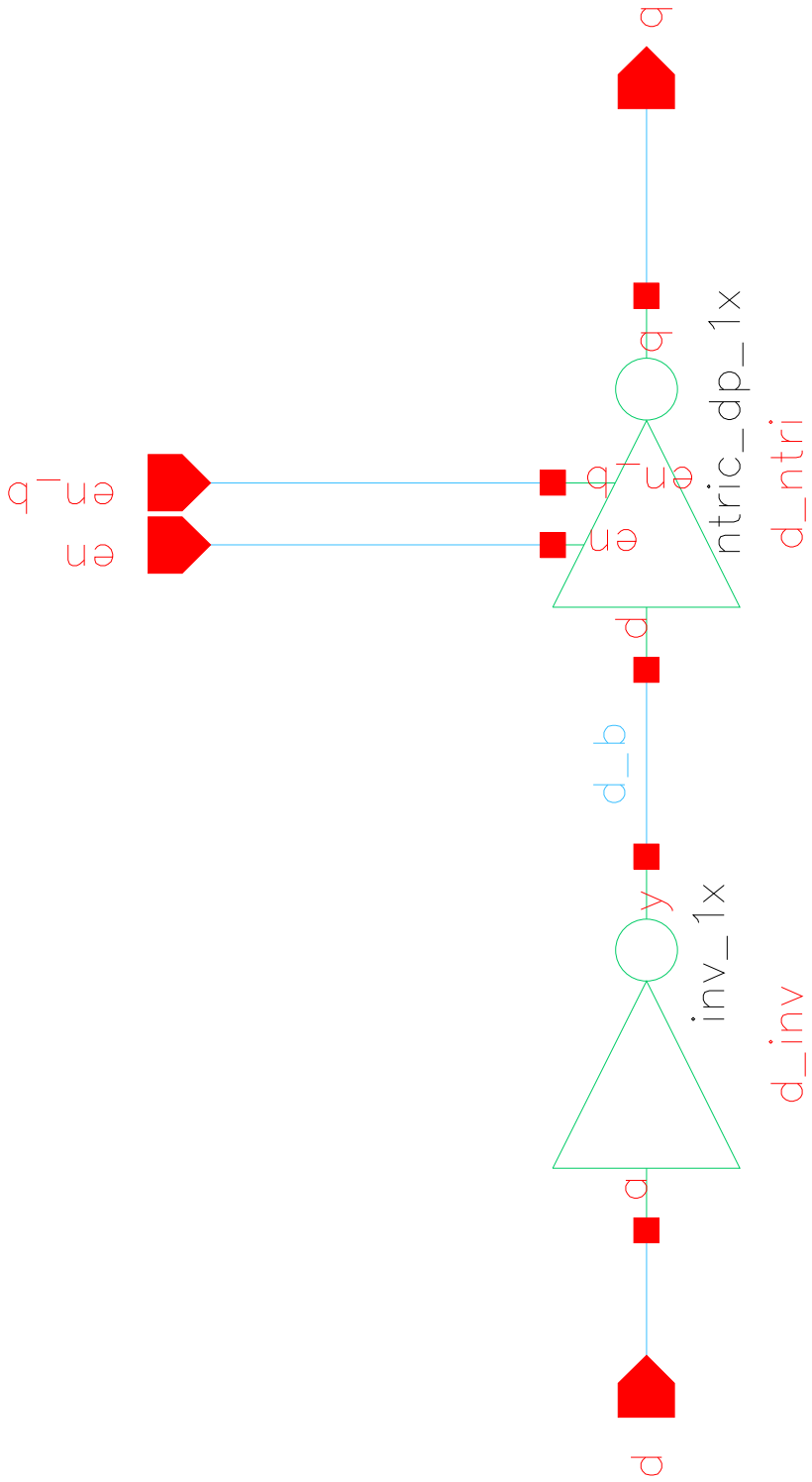
B.44 SRC_Lo_Block_sch



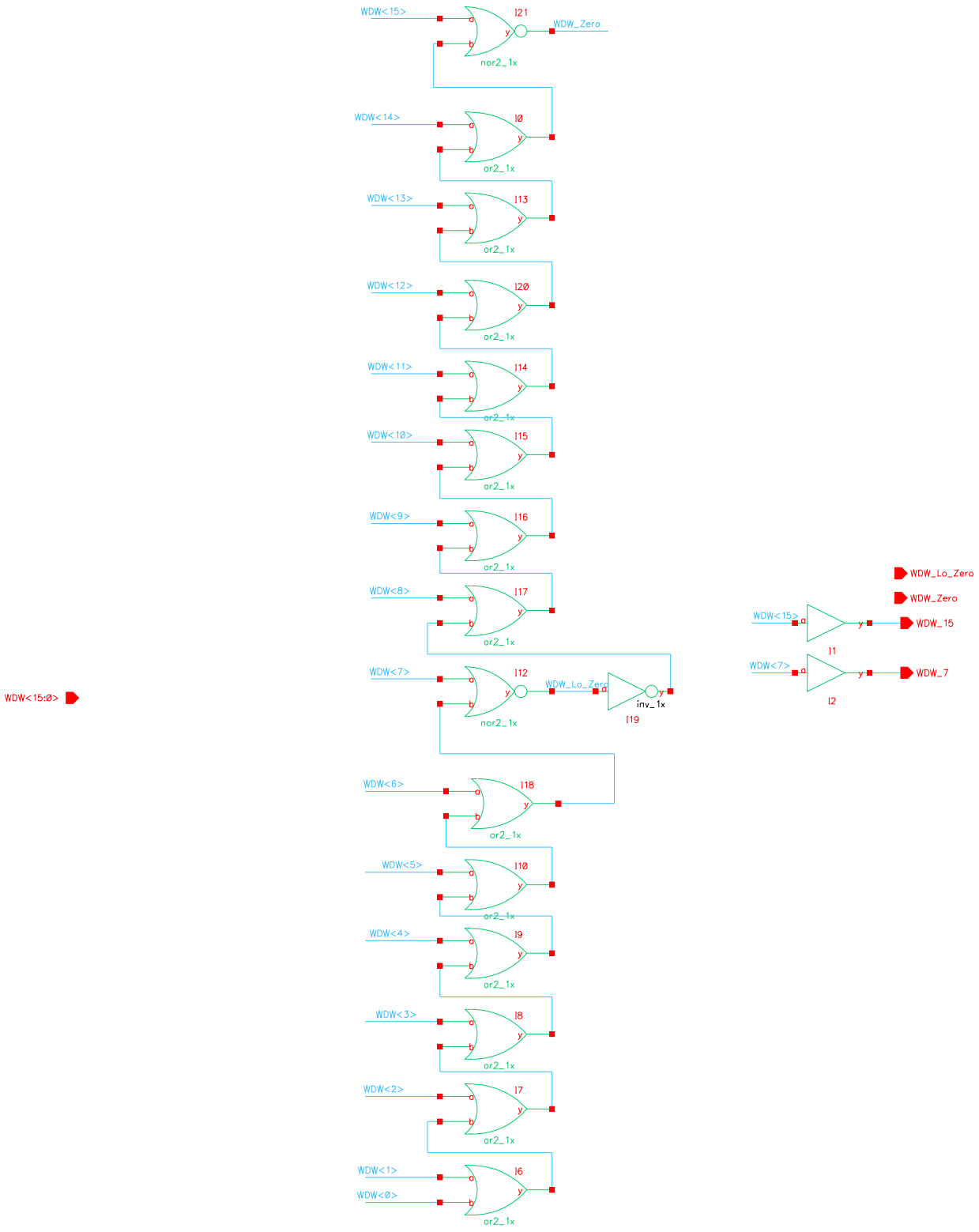
B.45 tri_dp_16_1x_sch



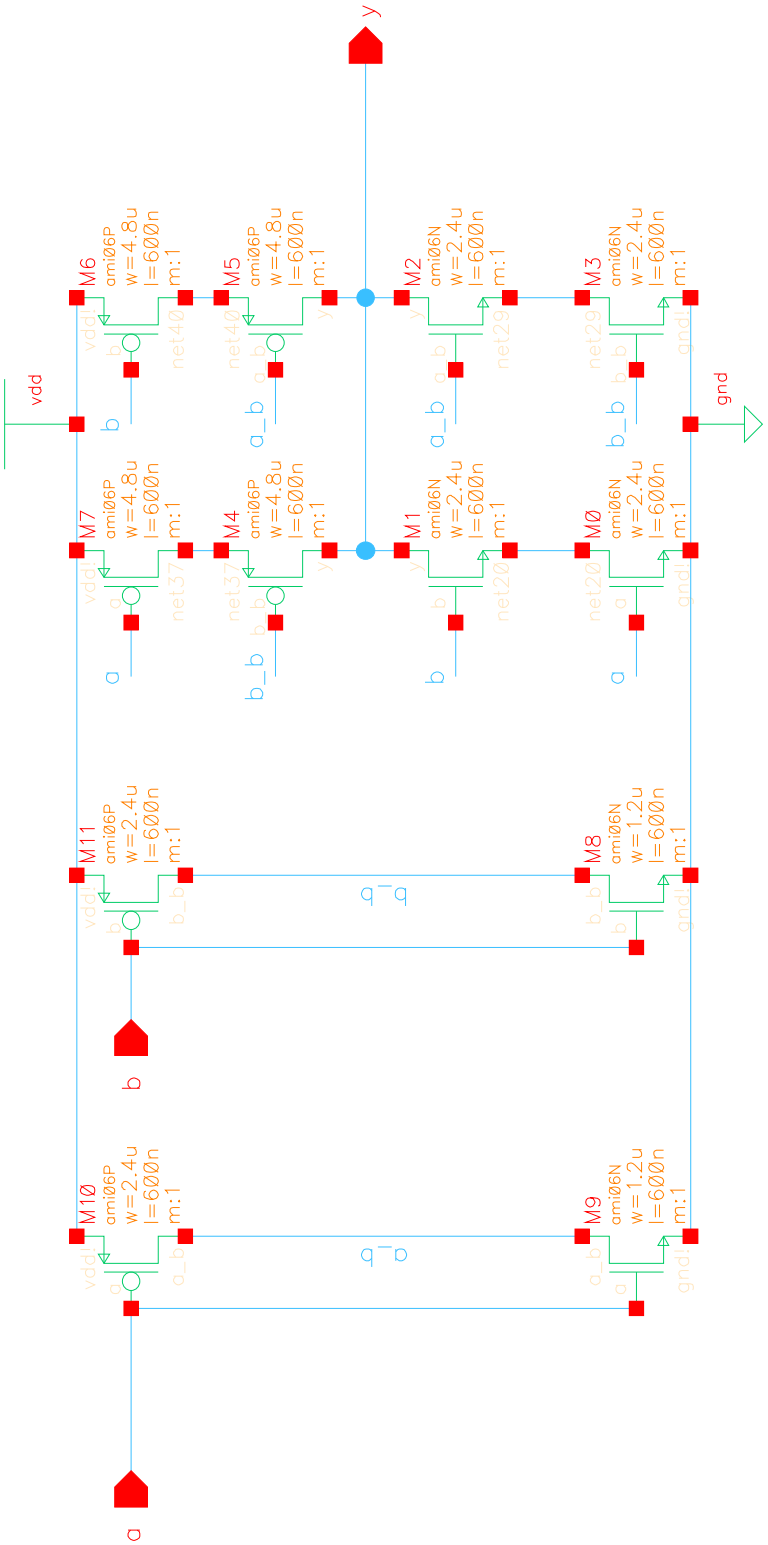
B.46 tri_dp_1x_sch



B.47 WDW_Block_sch

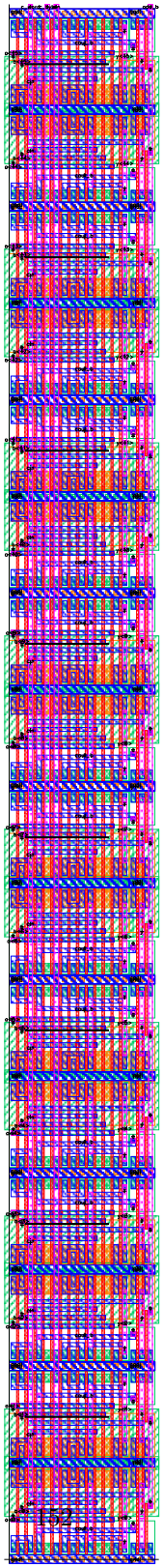


B.48 xor_1x

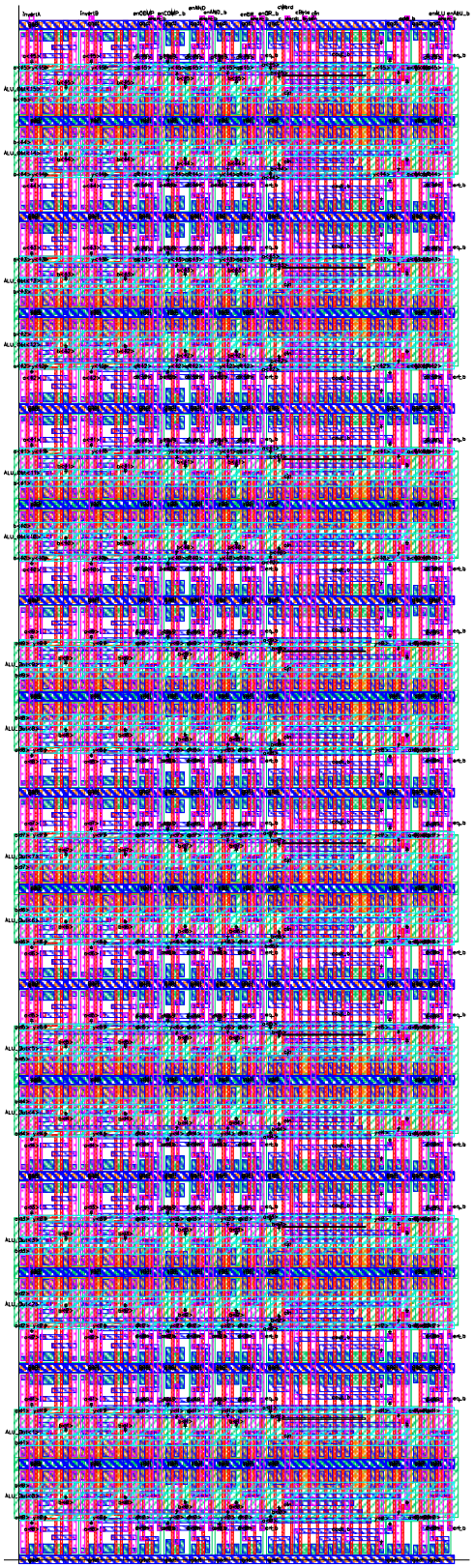

y
vdd
gnd
a
b
M6 ami06P w=4.8u l=600n m:1
M5 ami06P w=4.8u l=600n m:1
M2 ami06N w=2.4u l=600n m:1
M3 ami06N w=2.4u l=600n m:1
M7 ami06P w=4.8u l=600n m:1
M4 ami06P w=4.8u l=600n m:1
M1 ami06N w=2.4u l=600n m:1
M0 ami06N w=2.4u l=600n m:1
M11 ami06P w=2.4u l=600n m:1
M8 ami06N w=1.2u l=600n m:1
M10 ami06P w=2.4u l=600n m:1
M9 ami06N w=1.2u l=600n m:1

C Layouts

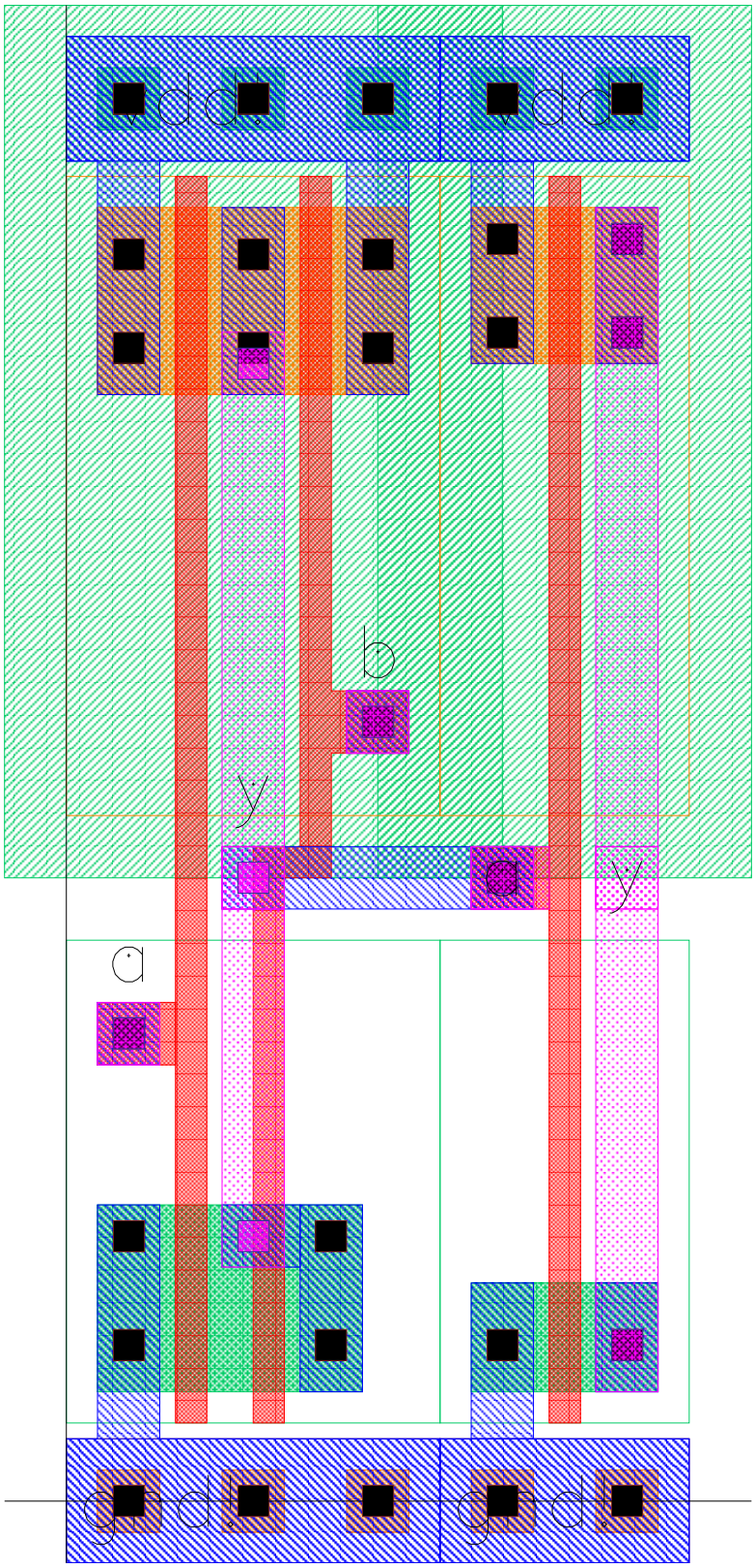
C.1 adder_1x_layout



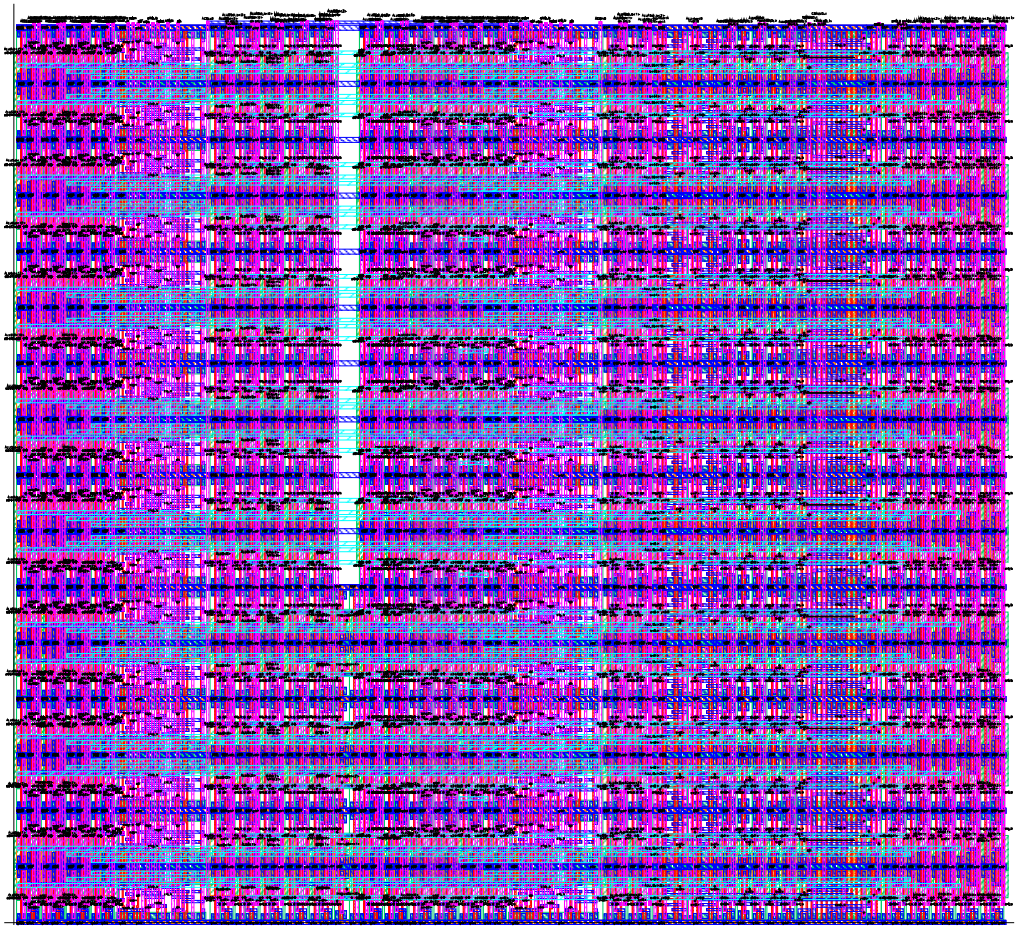
C.2 alu_1x_layout



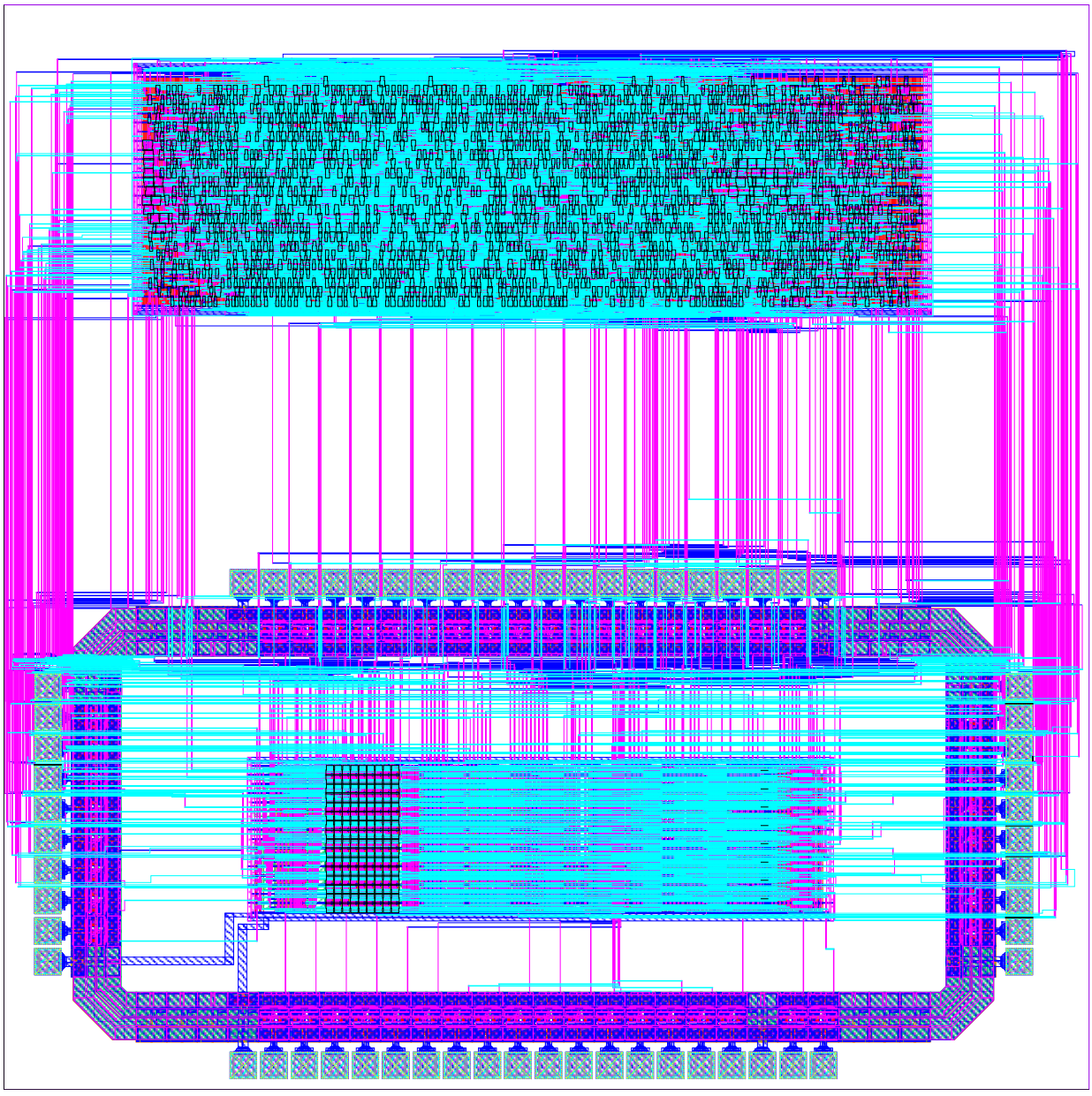
C.3 and2_1x.pdf



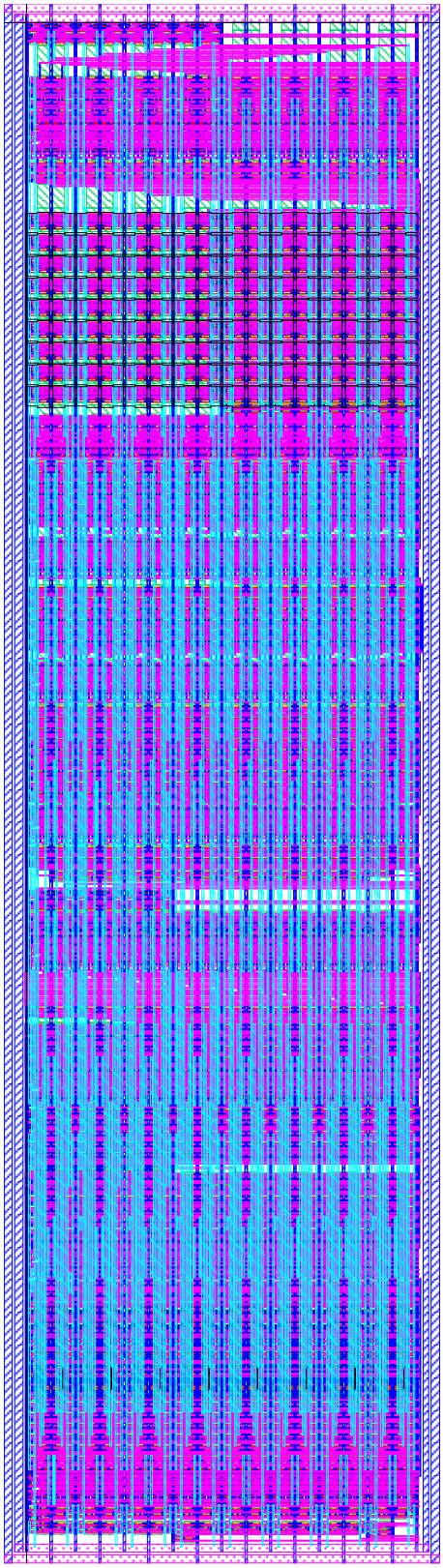
C.4 Ari_Block_layout



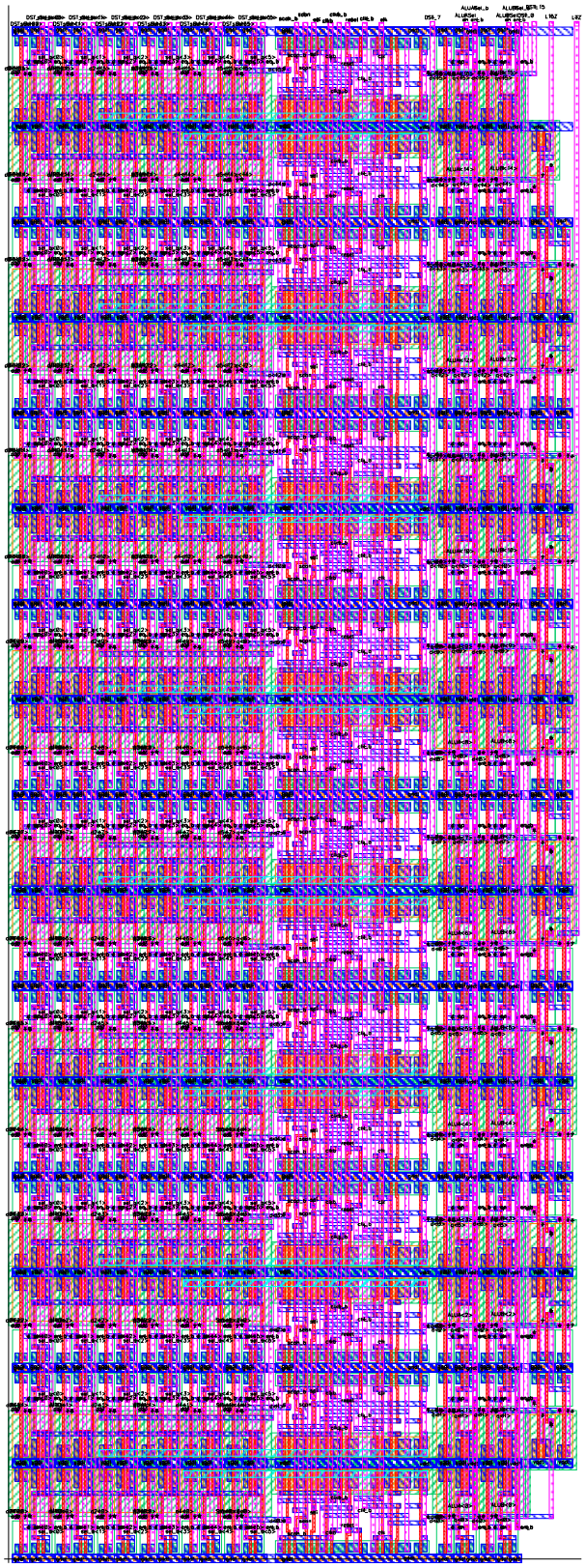
C.5 chip



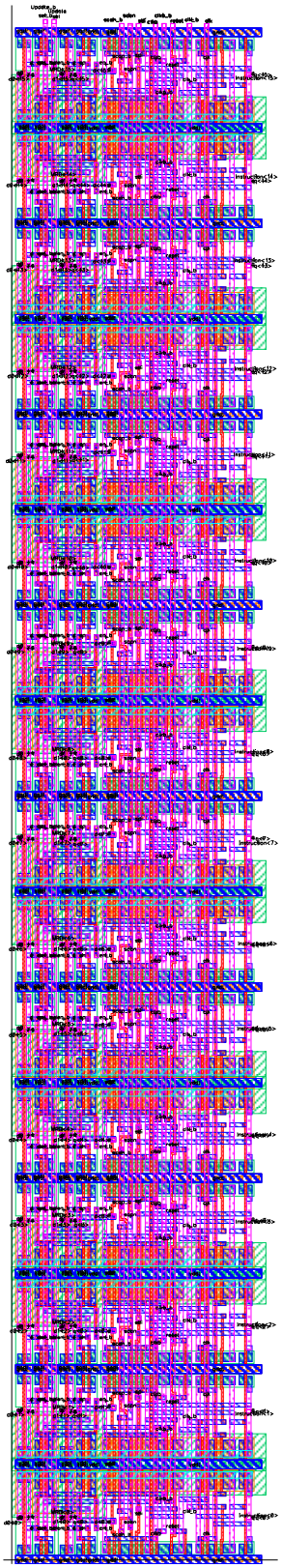
C.6 datapath



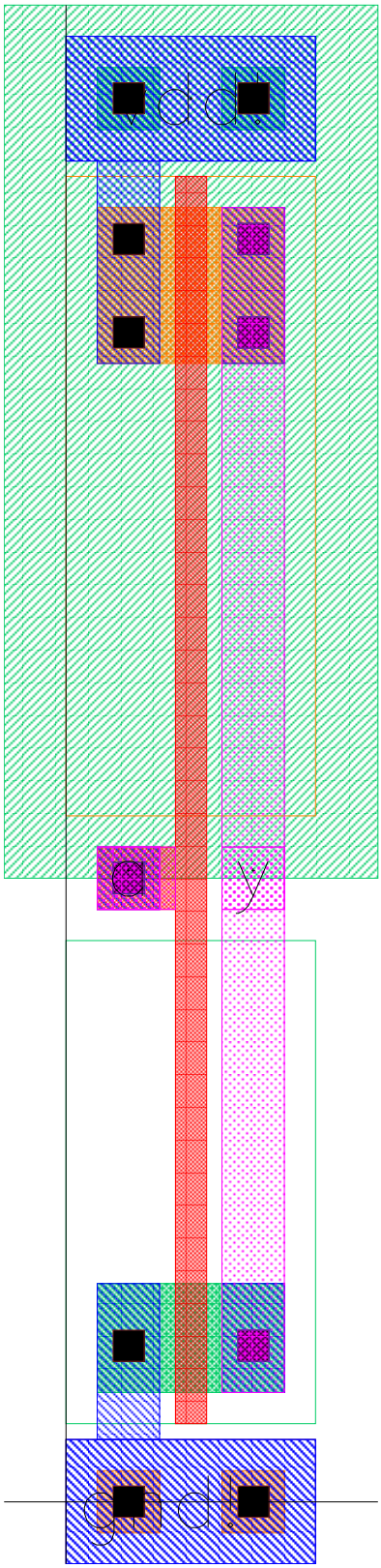
C.7 DST_Block_layout



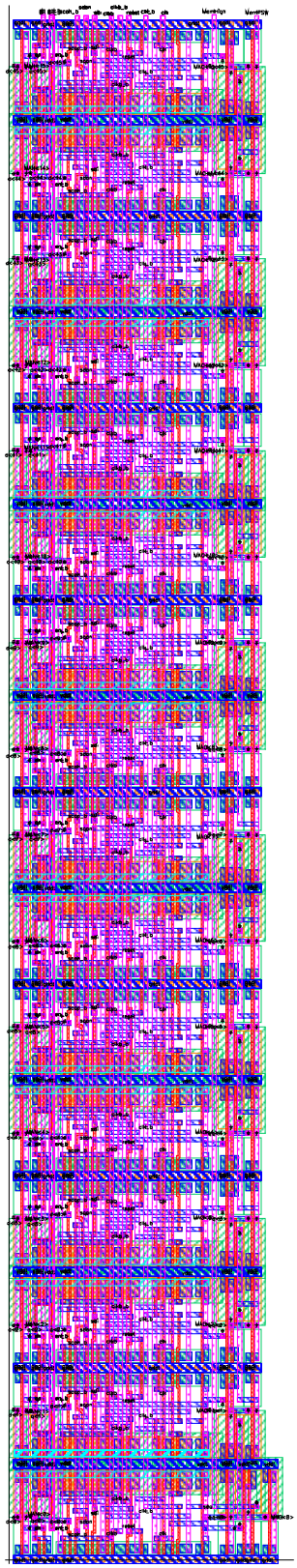
C.8 Instruction_Block_layout



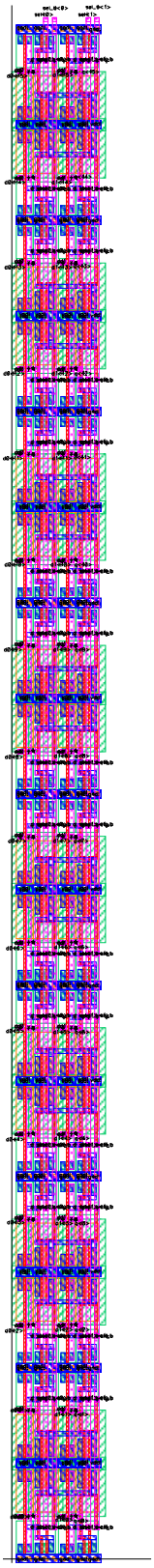
C.9 inv_1x



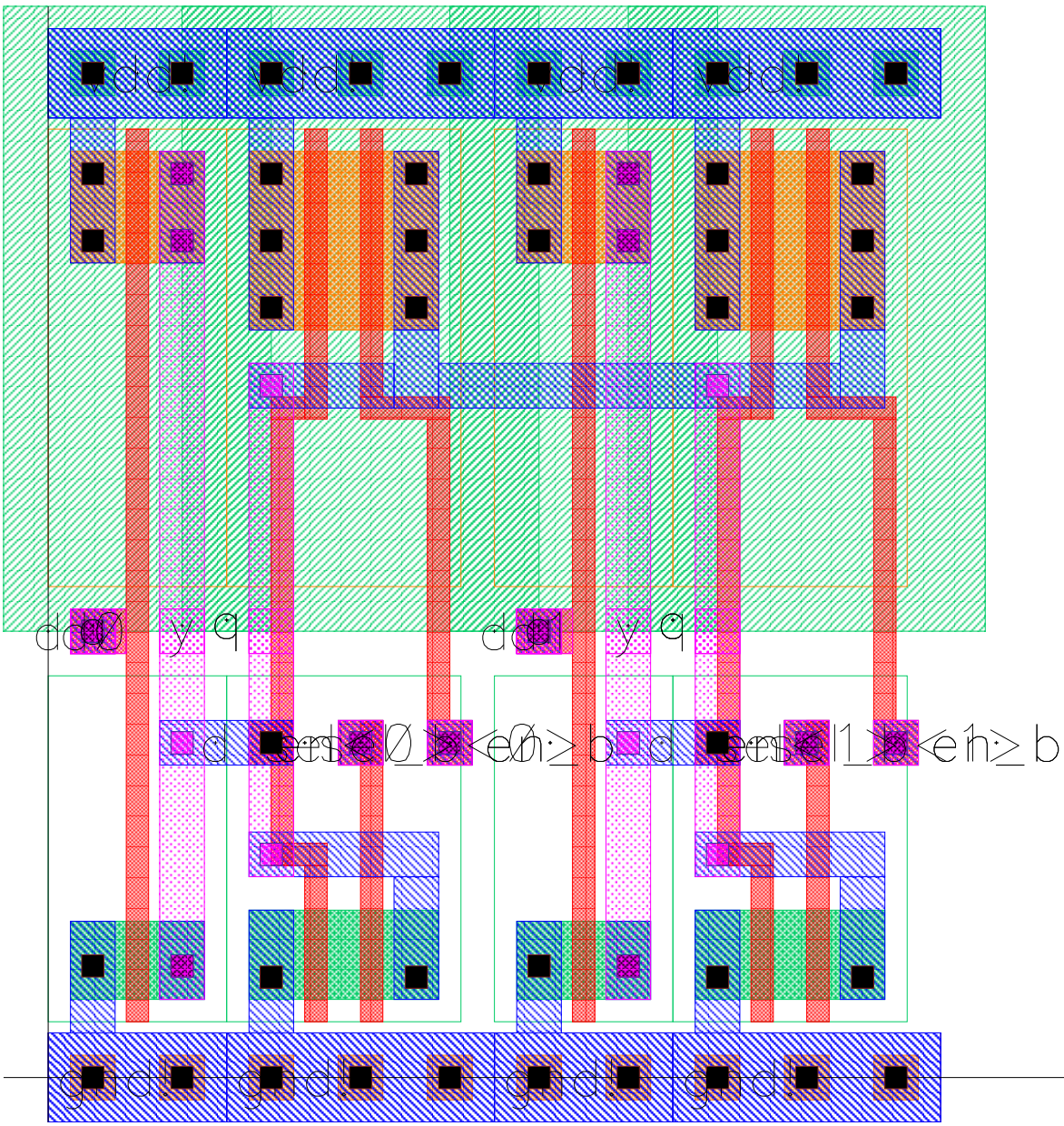
C.10 MemAddr_Block_layout



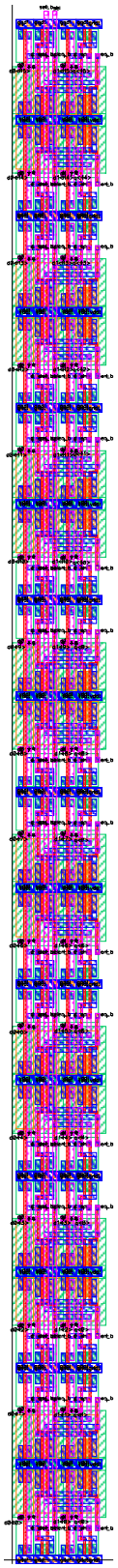
C.11 mux2_16_layout



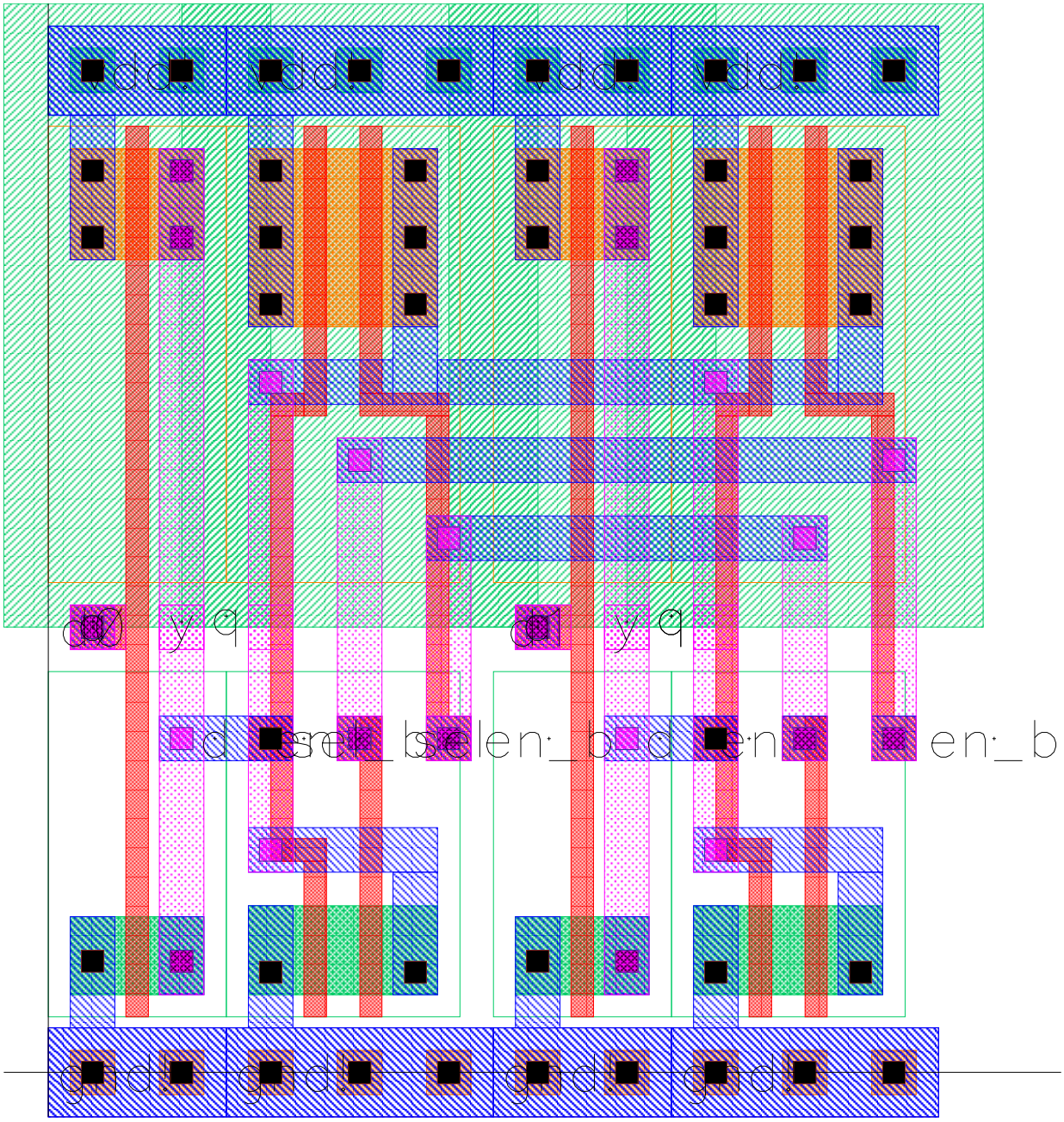
C.12 mux2_1x_layout



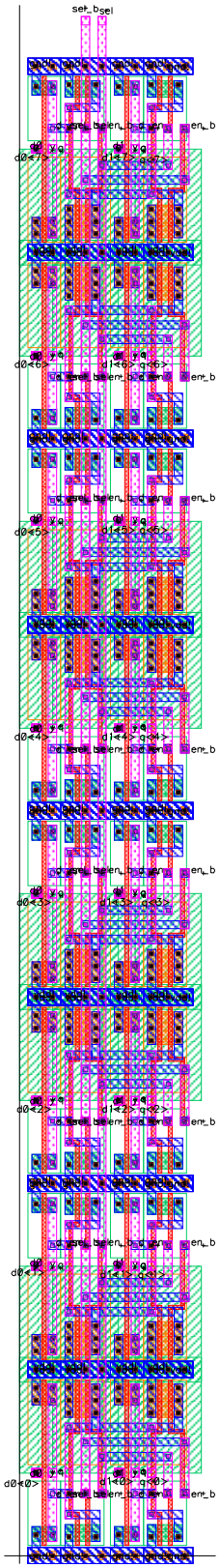
C.13 mux2_onehot_16_layout



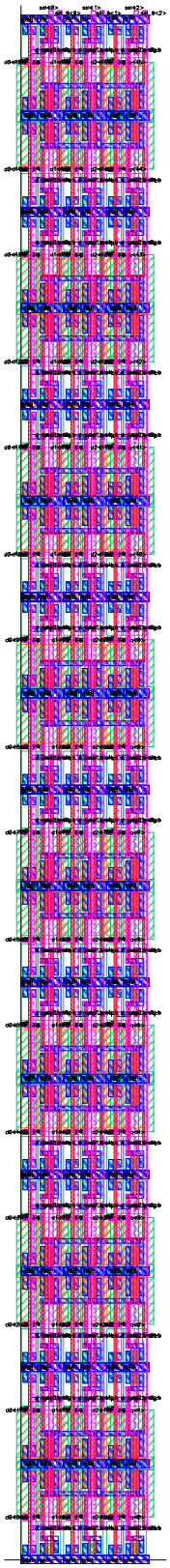
C.14 mux2_onehot_1x_layout



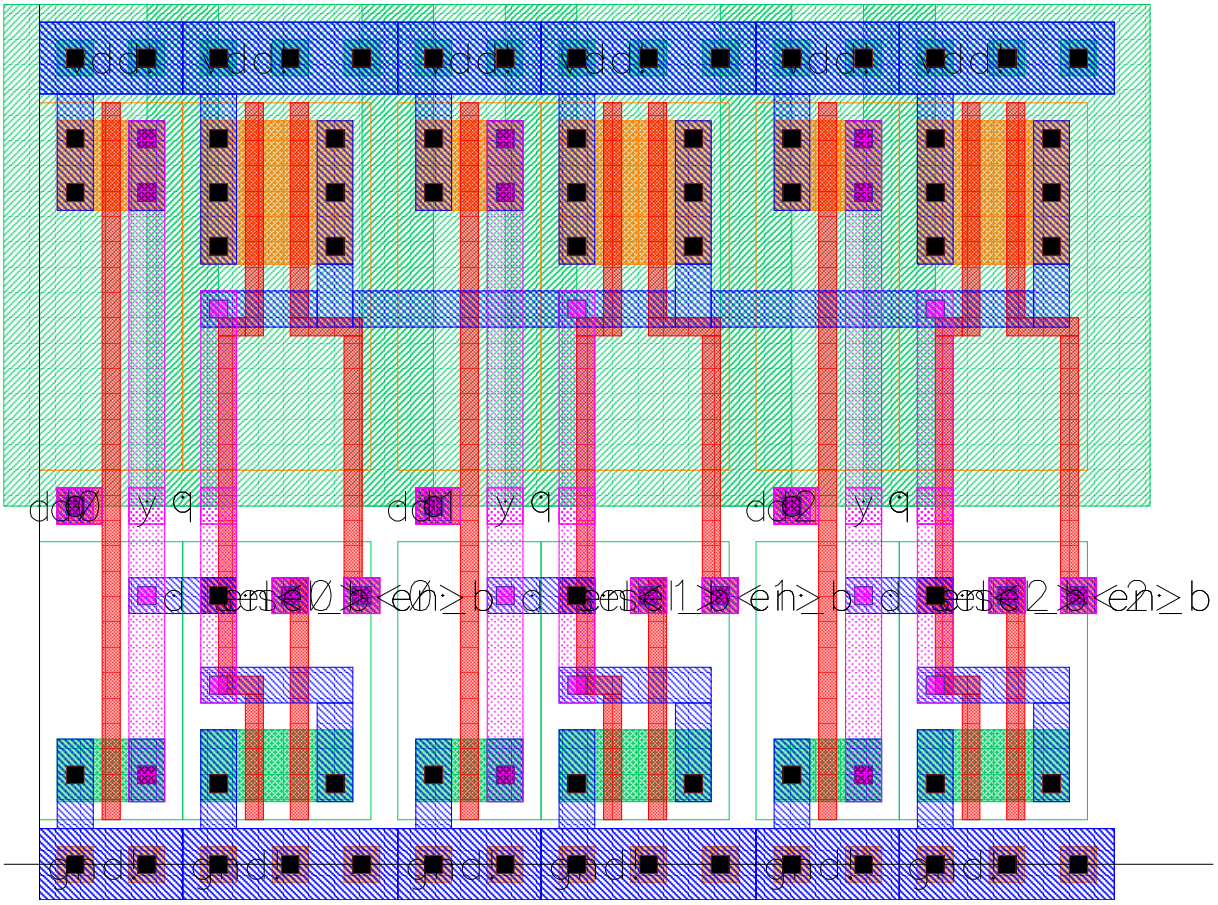
C.15 mux2_onehot_8



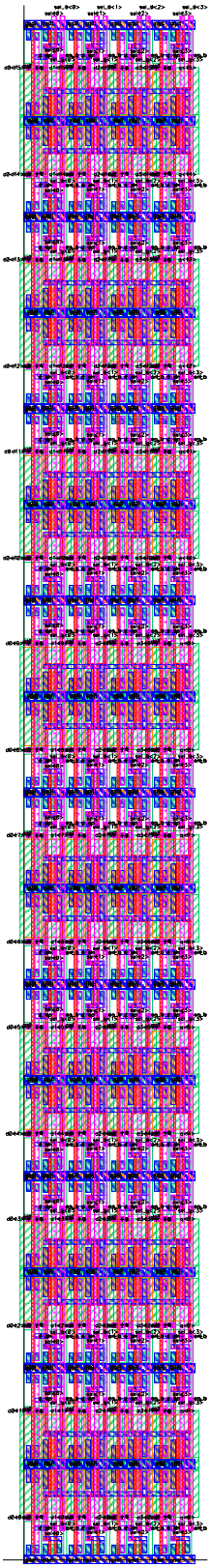
C.16 mux3_dp_16



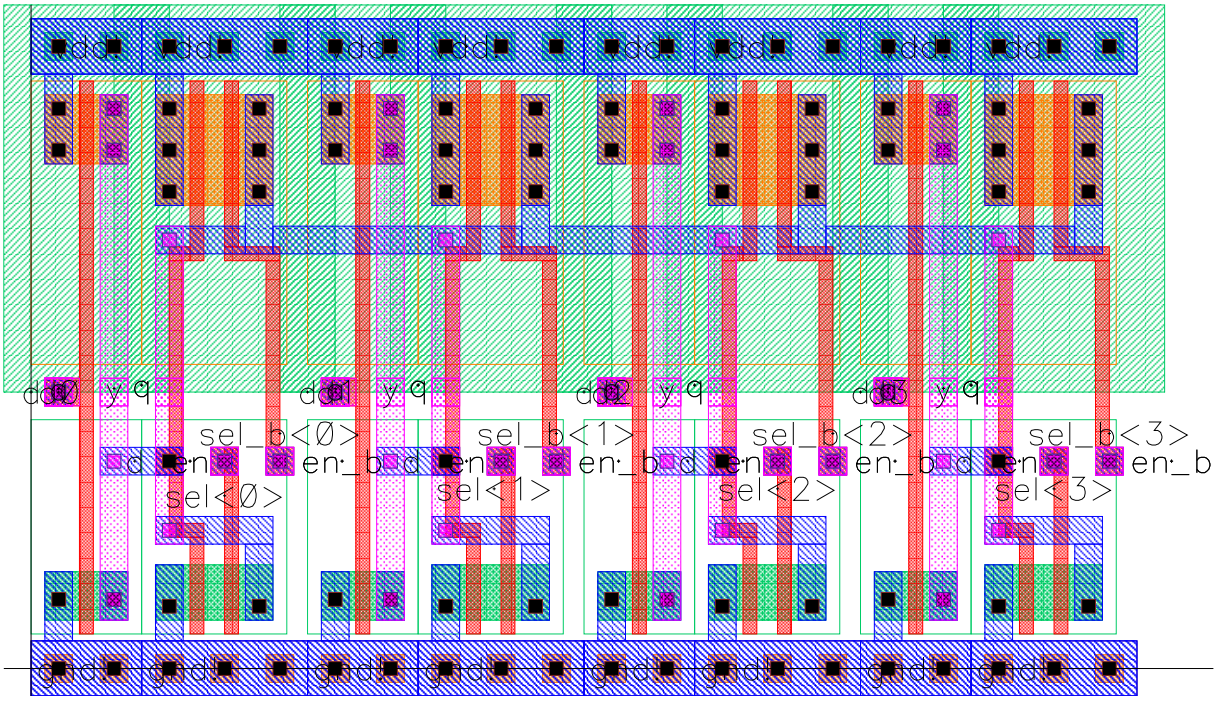
C.17 mux3_dp_1x



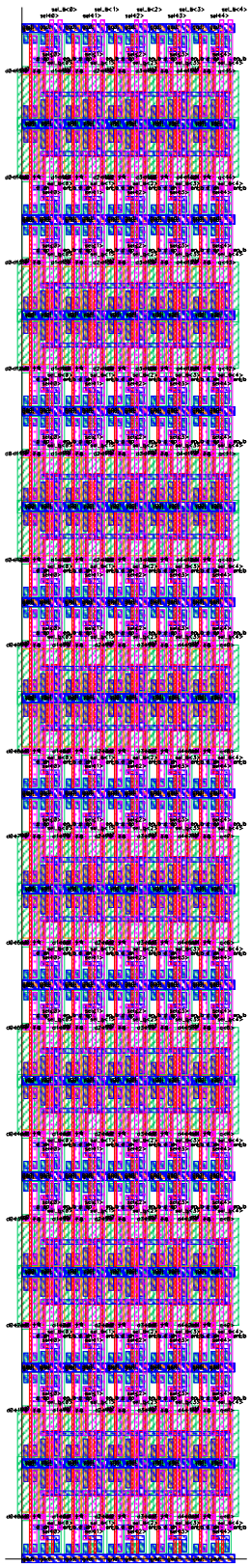
C.18 mux4_16_layout



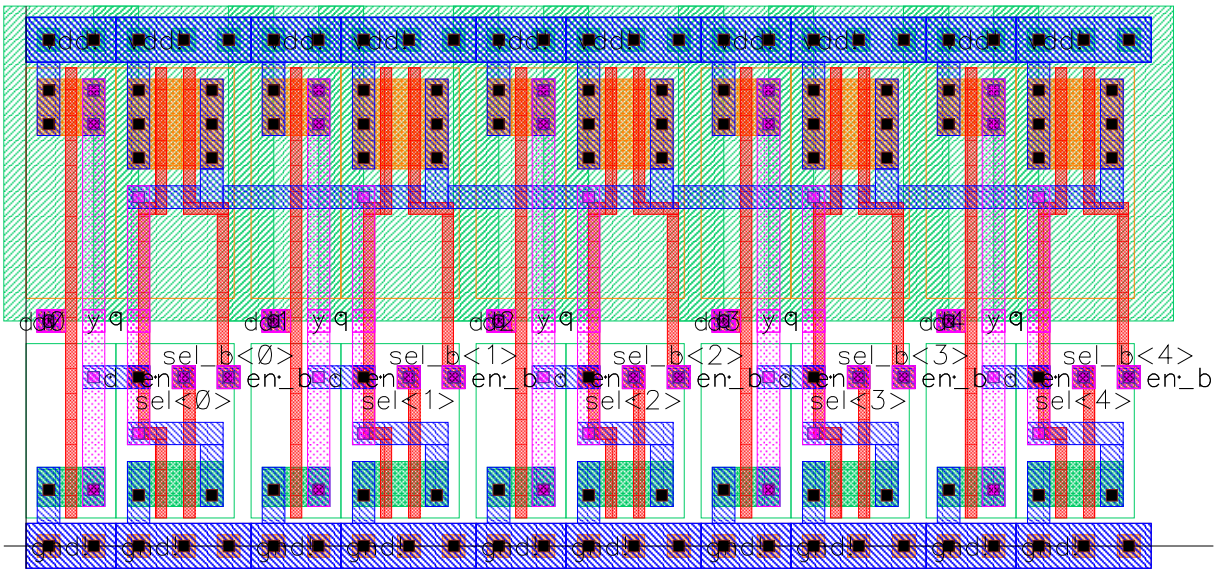
C.19 mux4_1x_layout



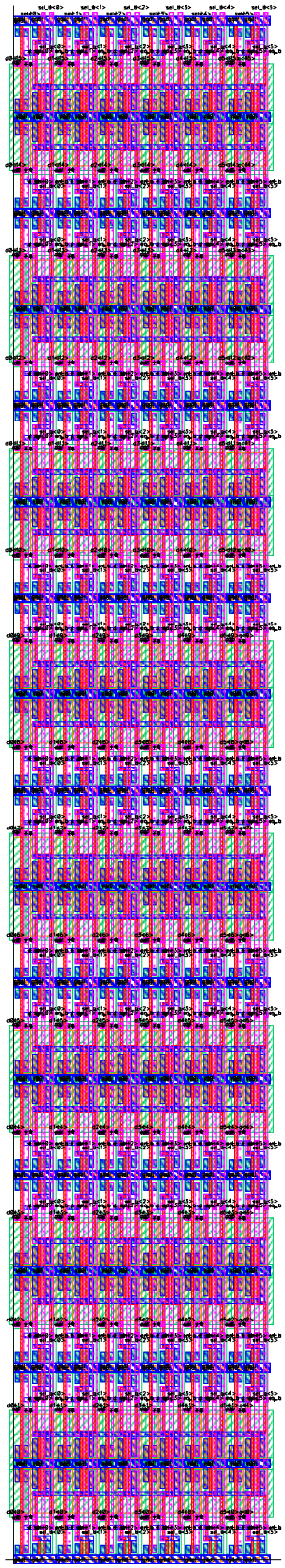
C.20 mux5_16_layout



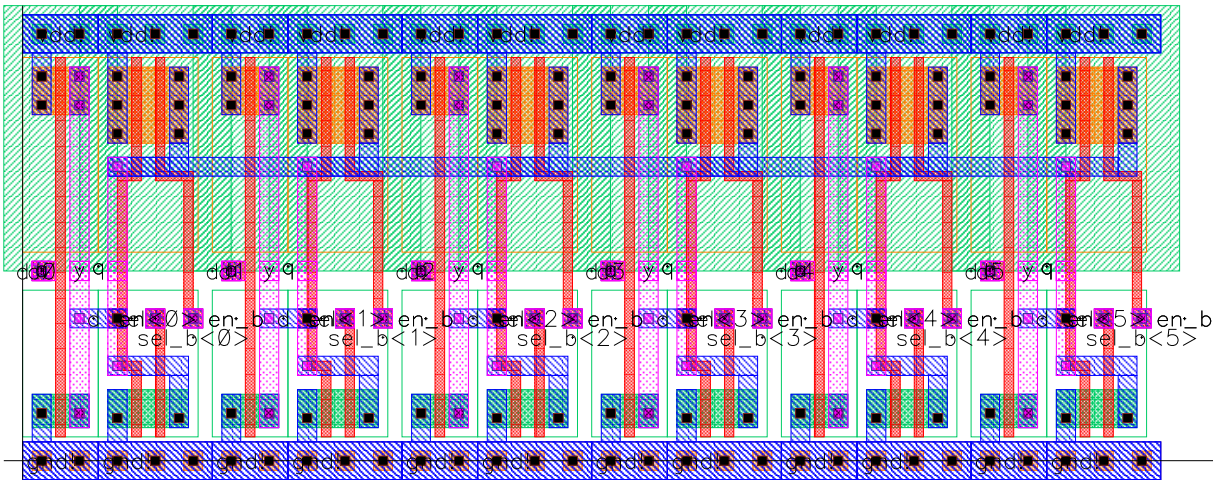
C.21 mux5_1x_layout



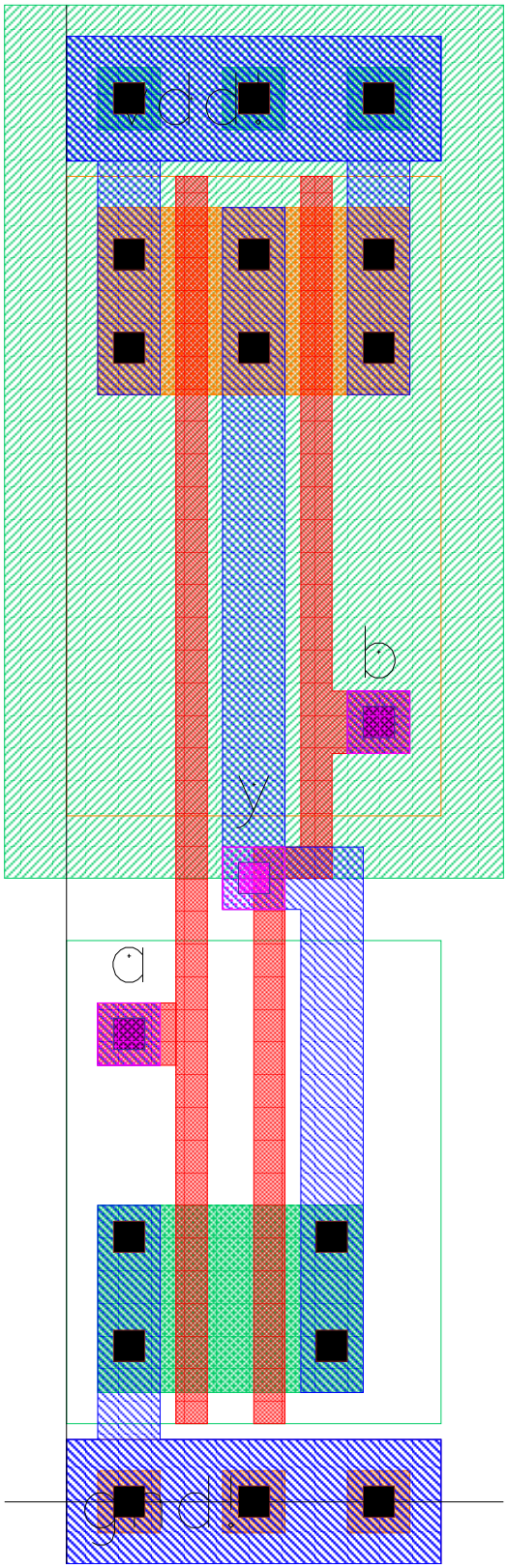
C.22 mux6_16_layout



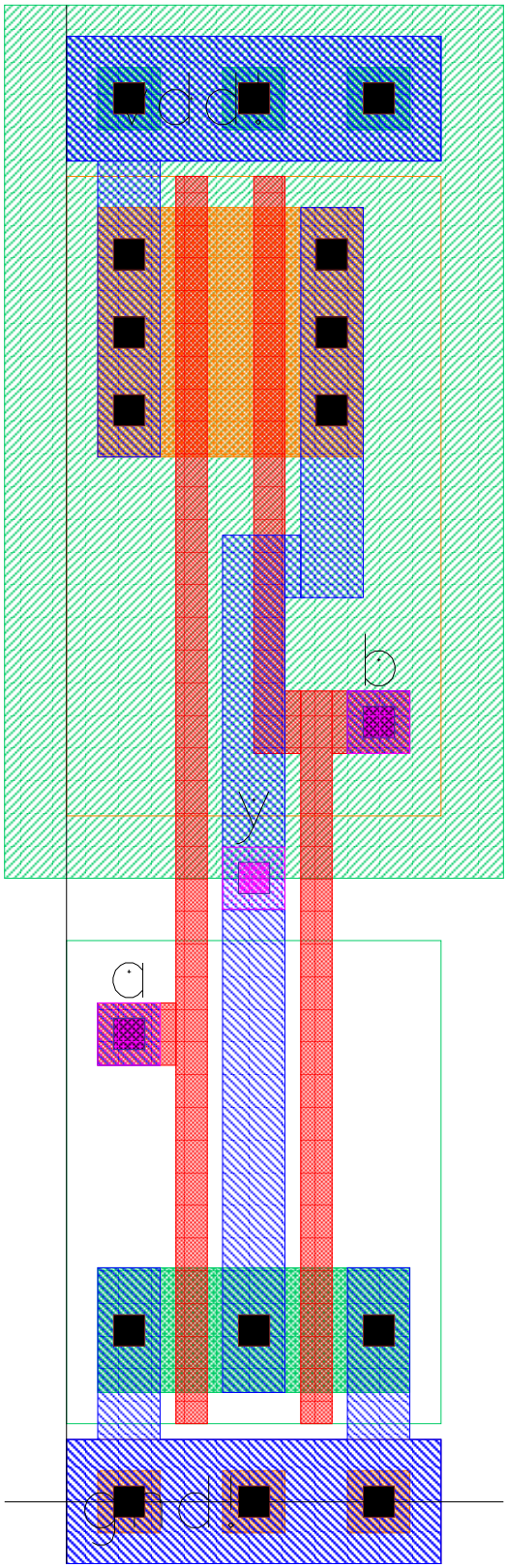
C.23 mux6_1x_layout



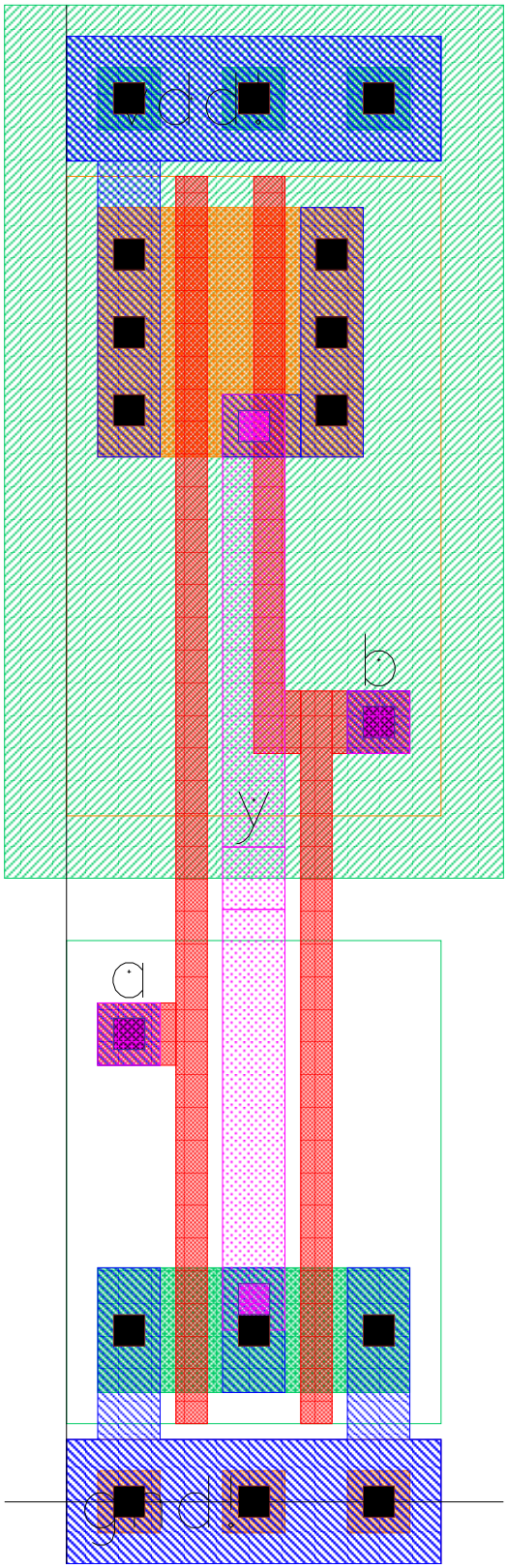
C.24 nand2_1x



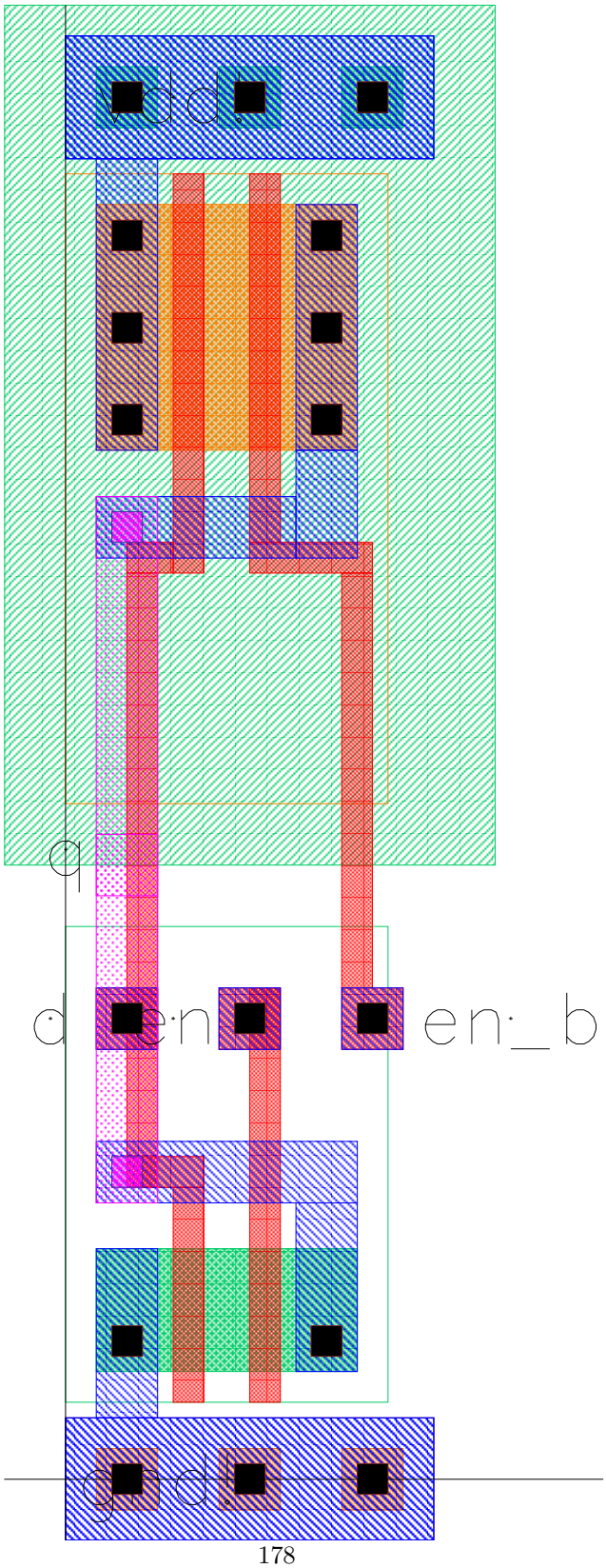
C.25 nor2_1x



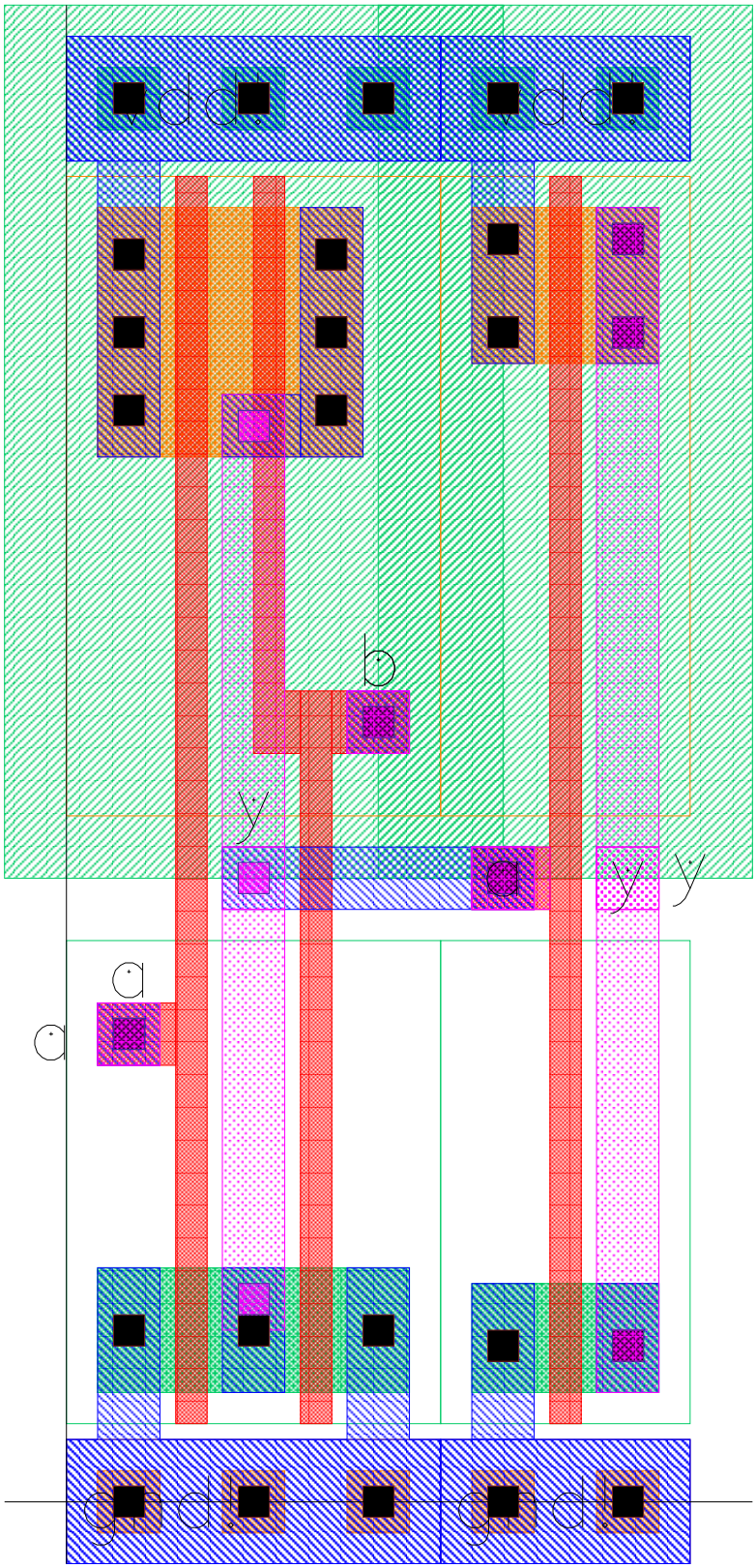
C.26 nor2_m2_1x



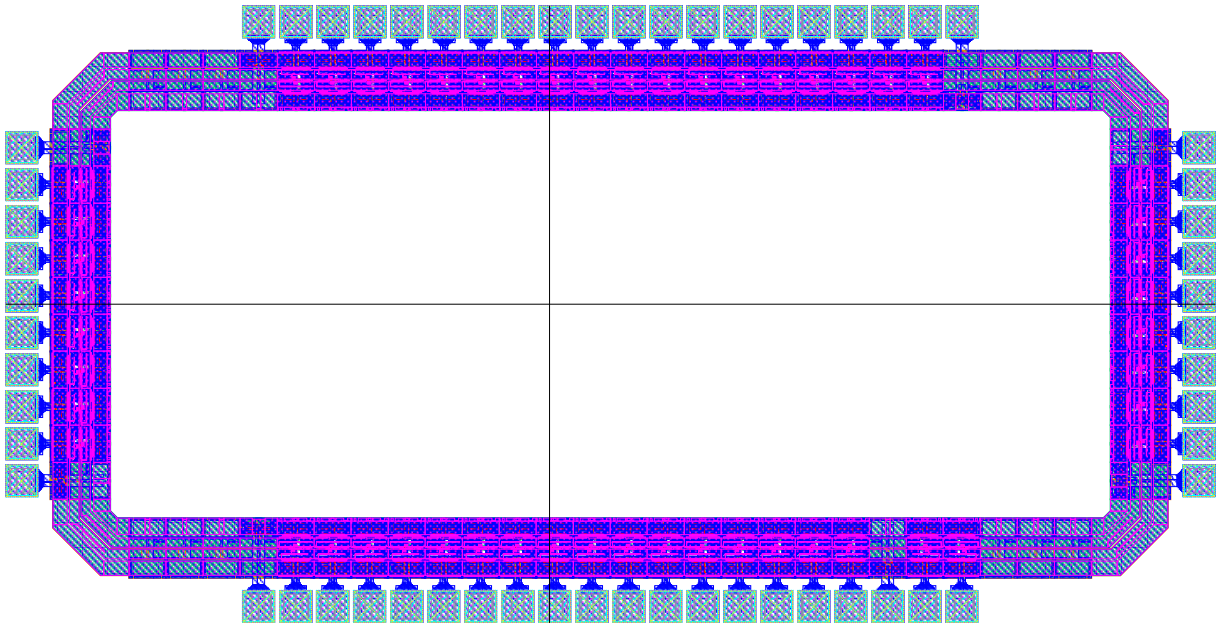
C.27 ntri_dp_1x



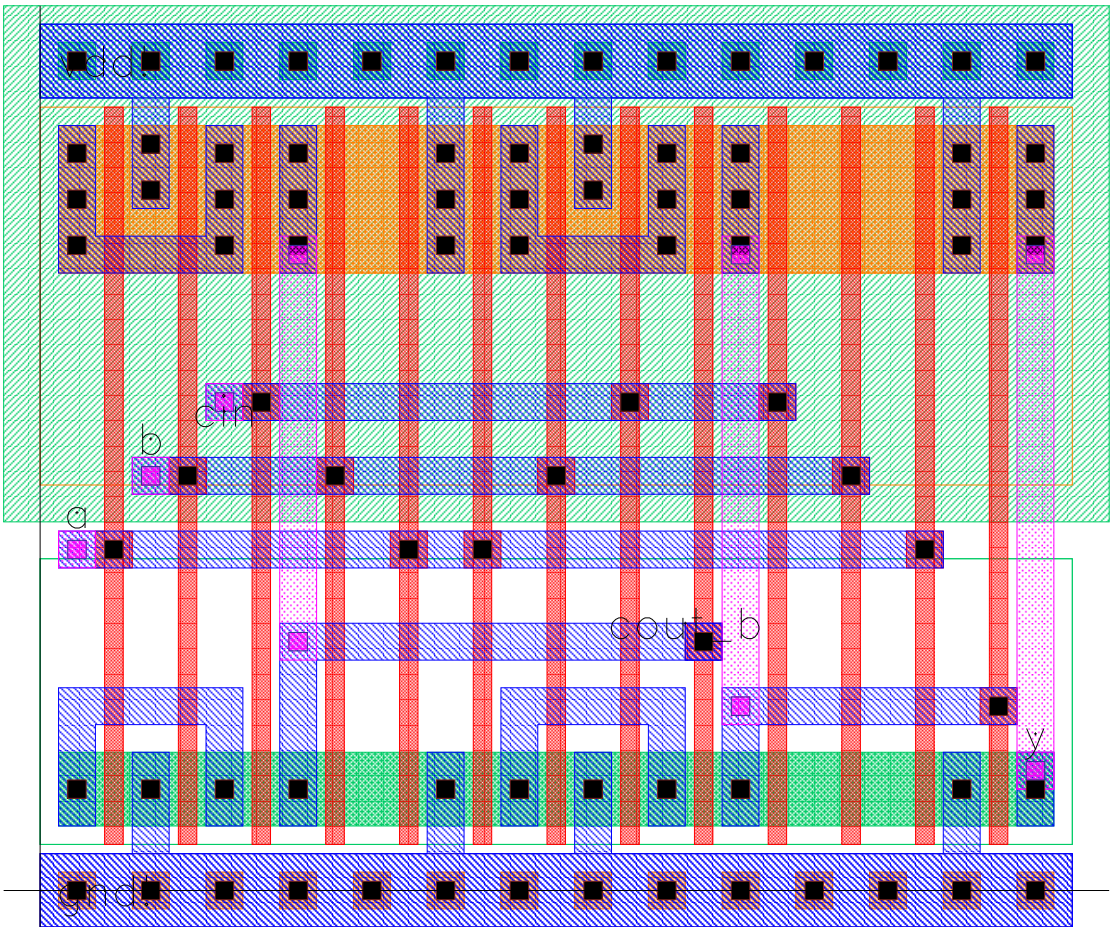
C.28 or2_1x



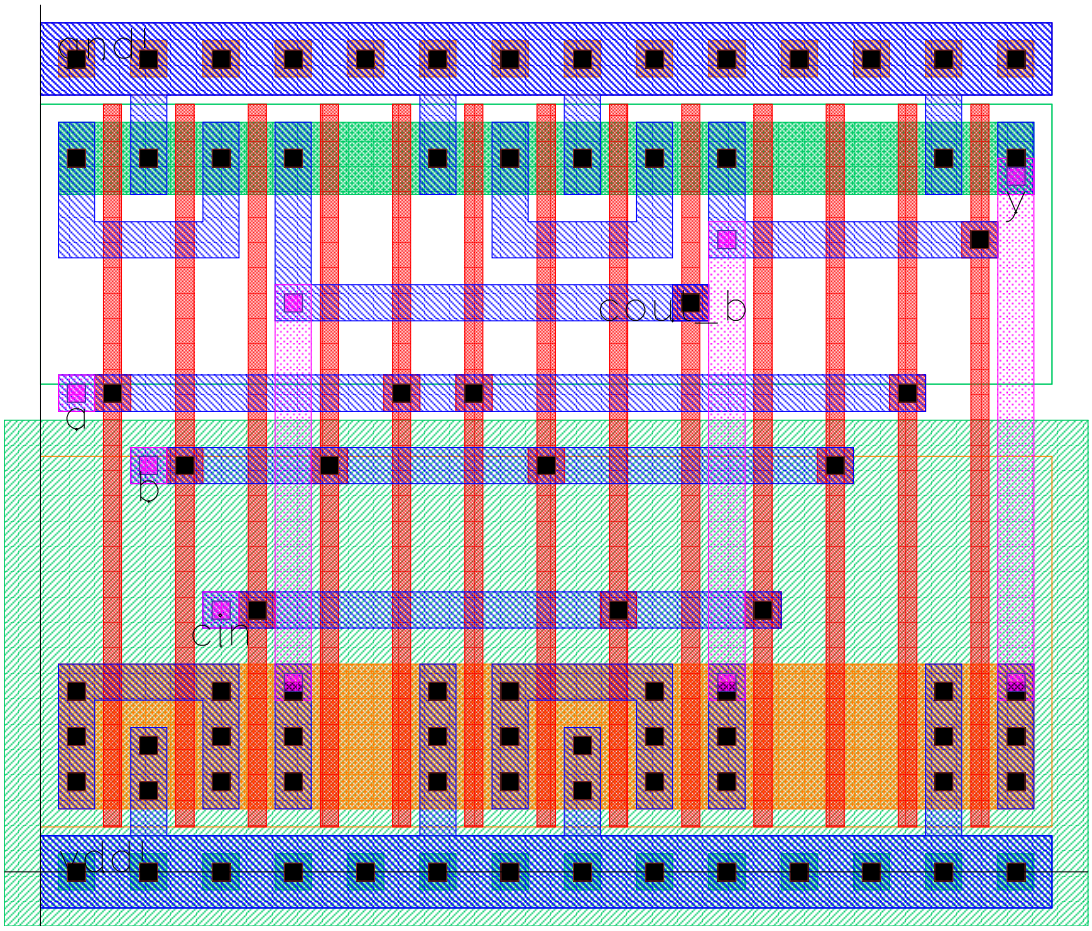
C.29 padframes_full



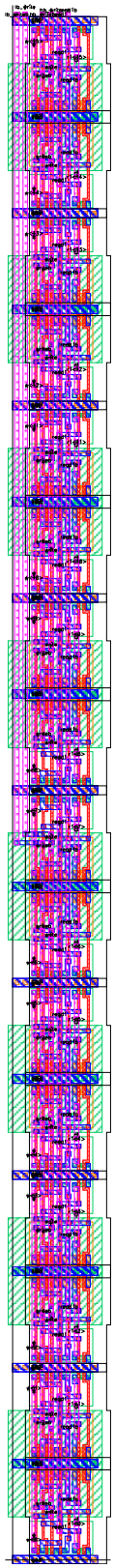
C.30 pdp11_full_add_1x



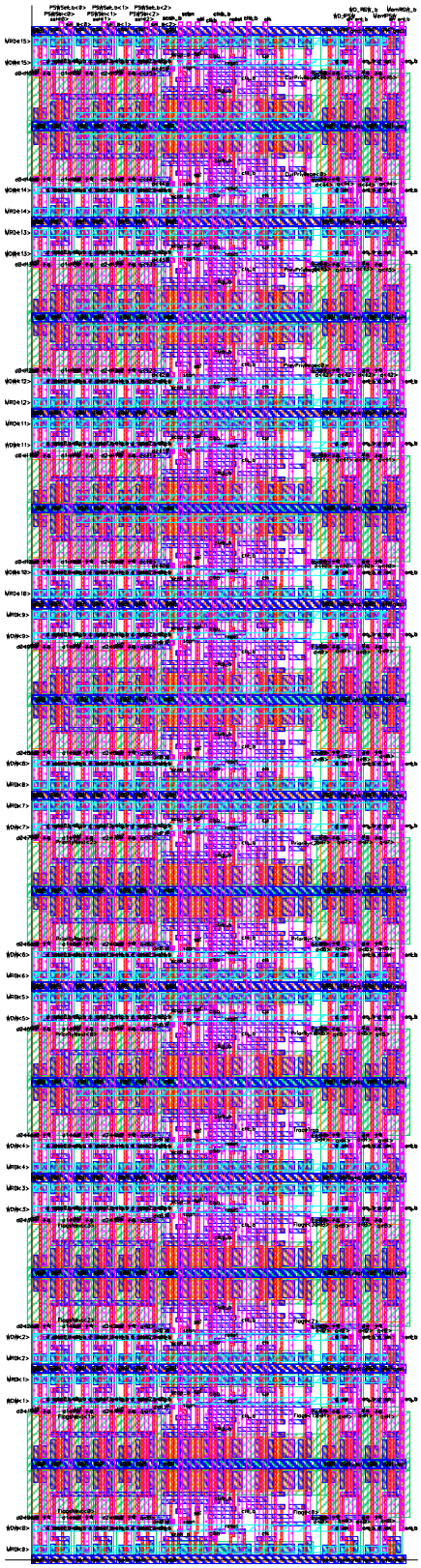
C.31 pdp11_full_add_flip_1x



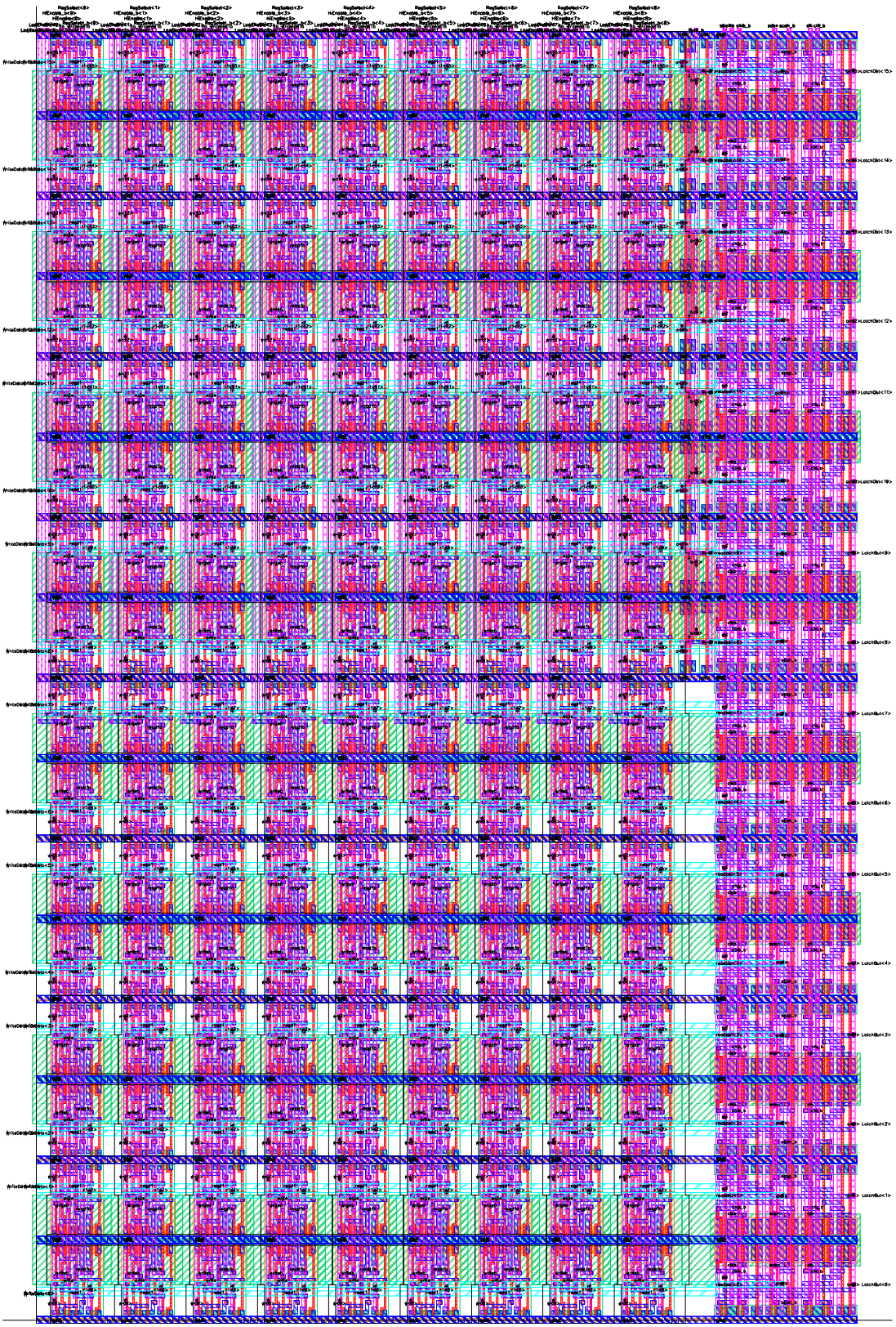
C.32 pdpreg16



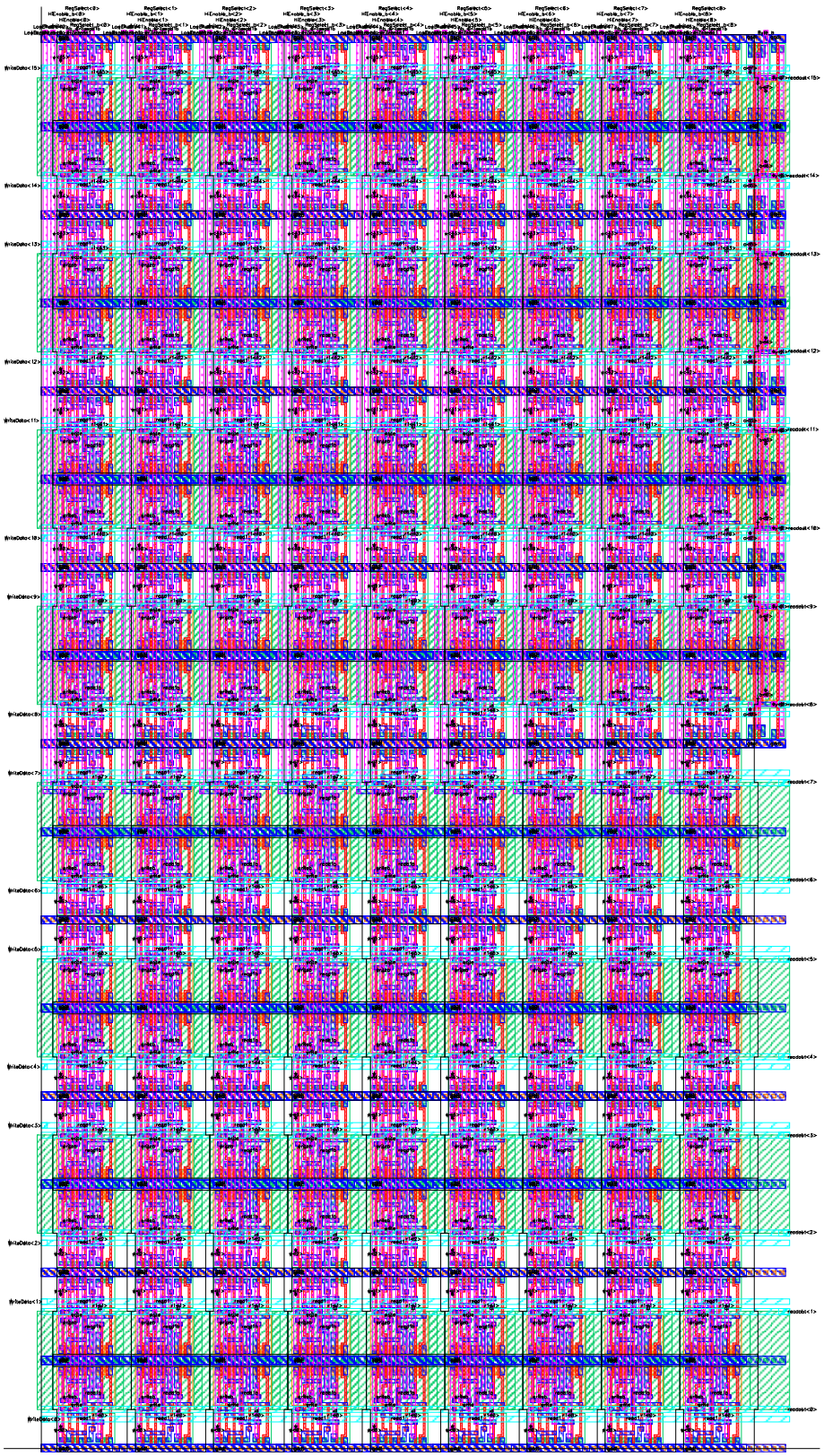
C.33 PSW_Block_layout



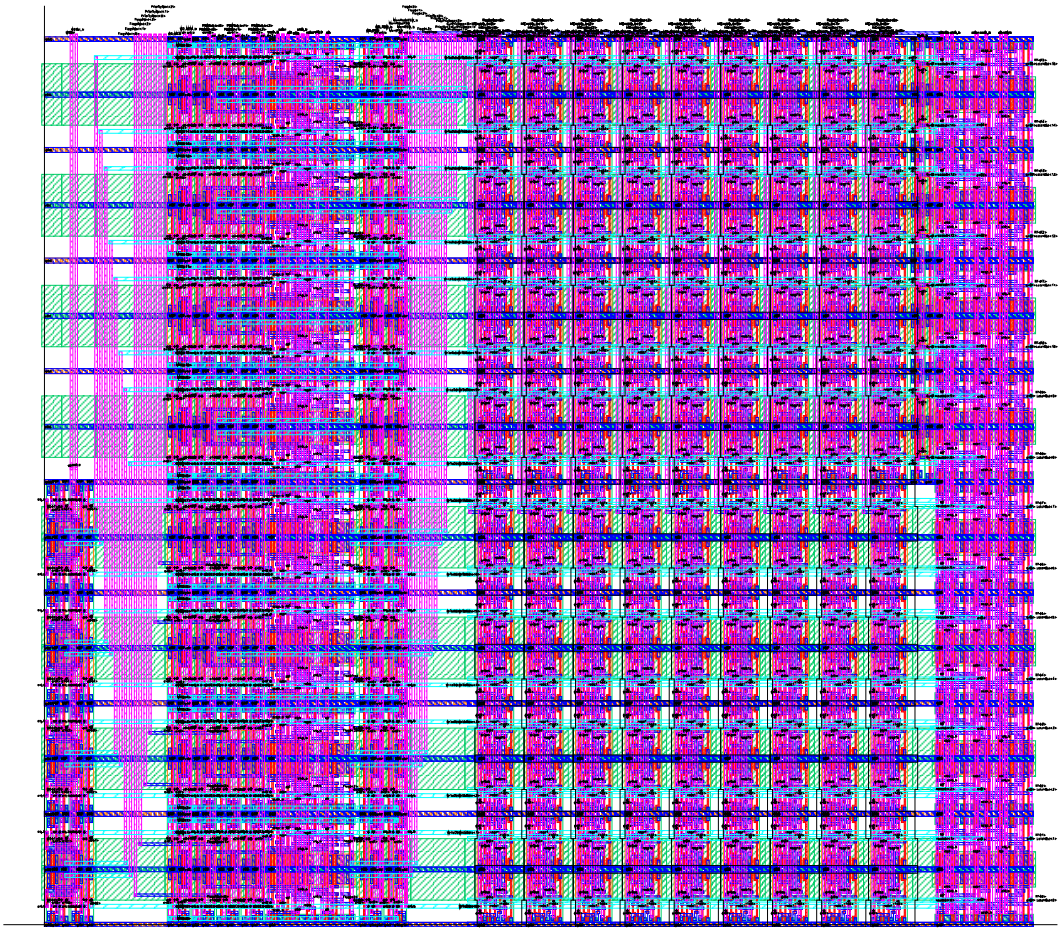
C.34 Reg_Block



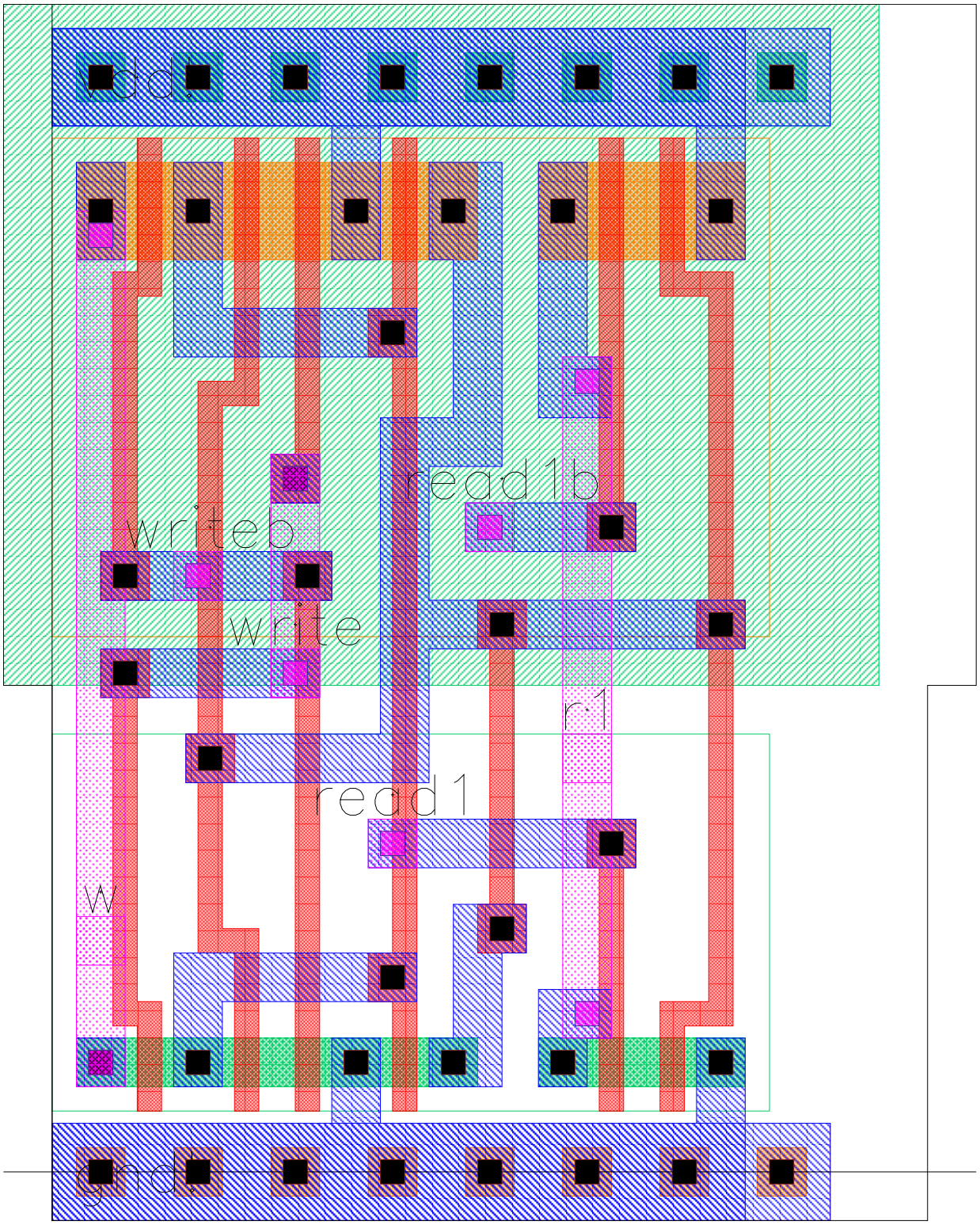
C.35 RegFile_layout



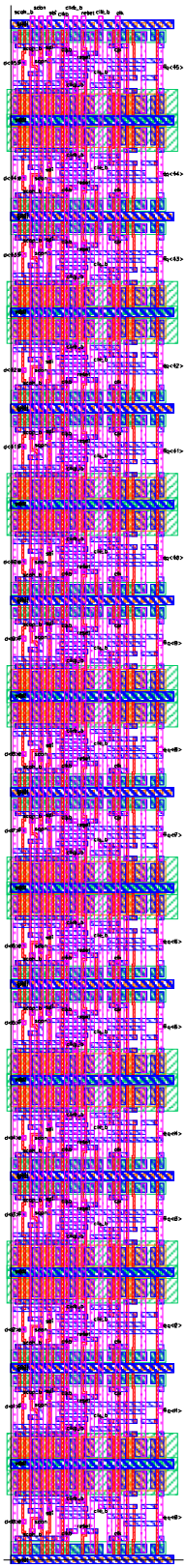
C.36 RegPSW_Block_layout



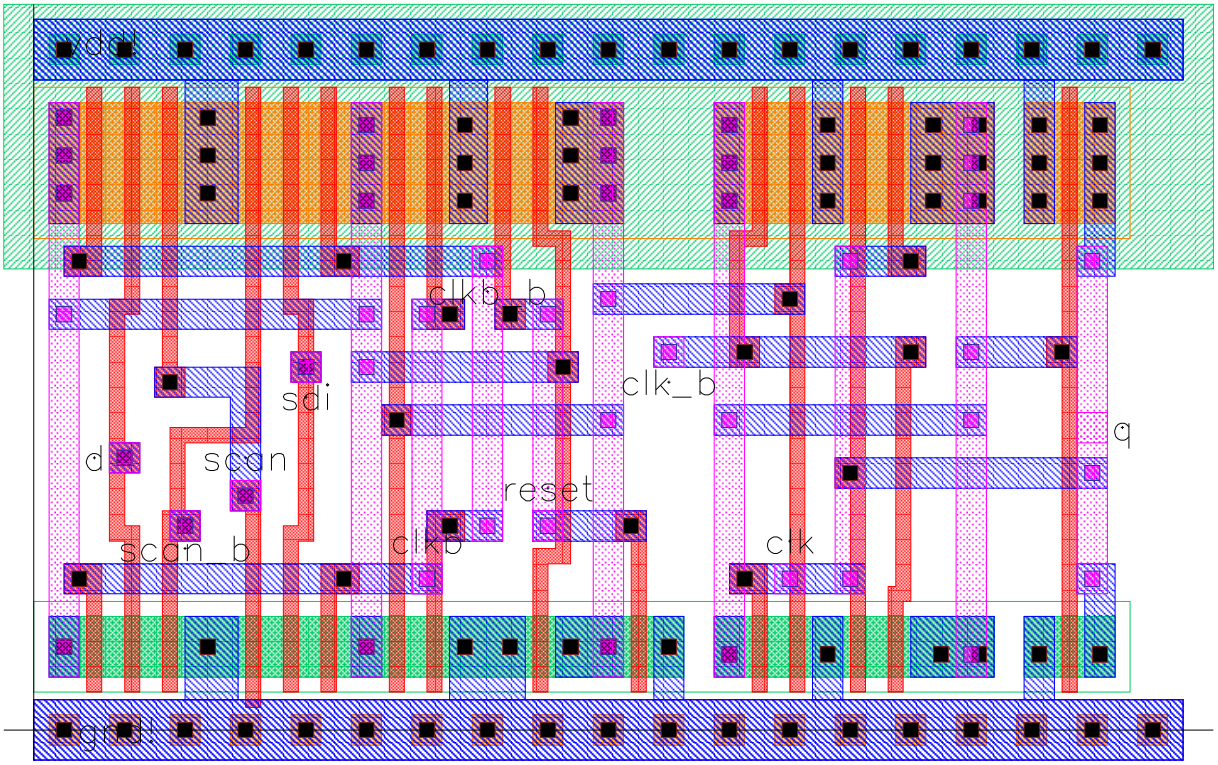
C.37 regram_dp



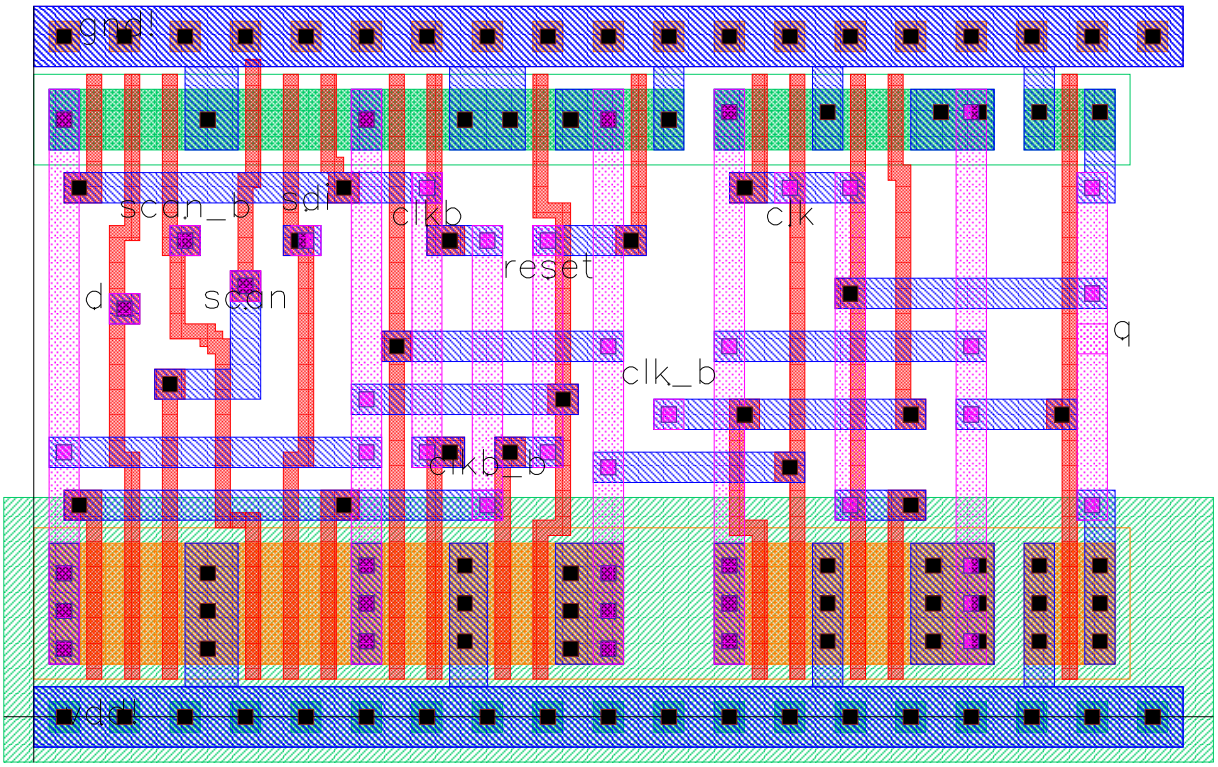
C.38 scanflopr_16_dp_1x



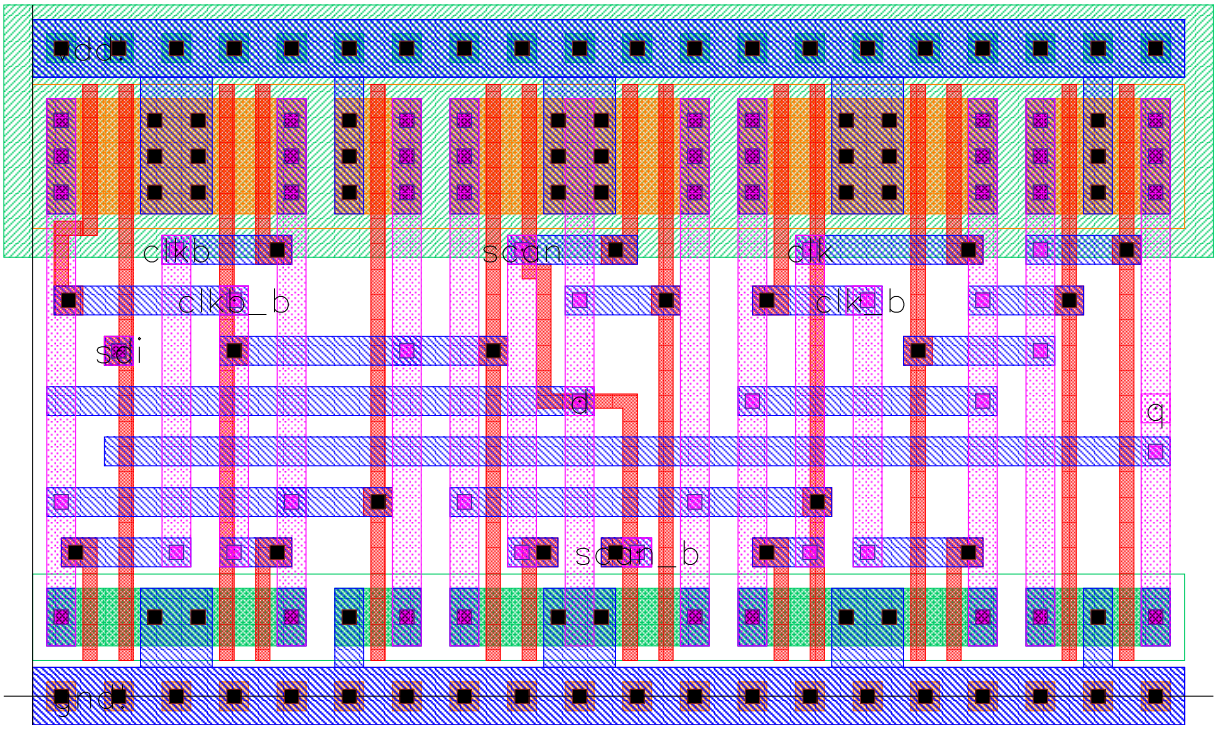
C.39 scanflopr_dp_1x



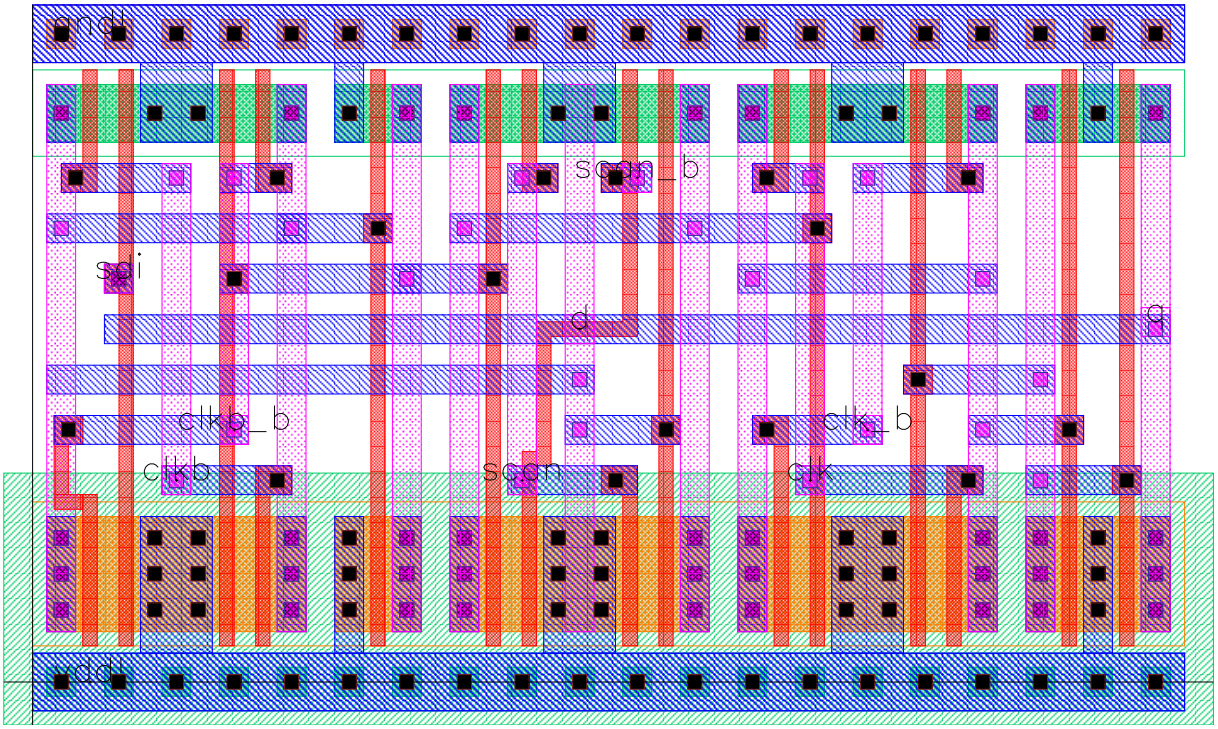
C.40 scanflopr_dp_1x_flip



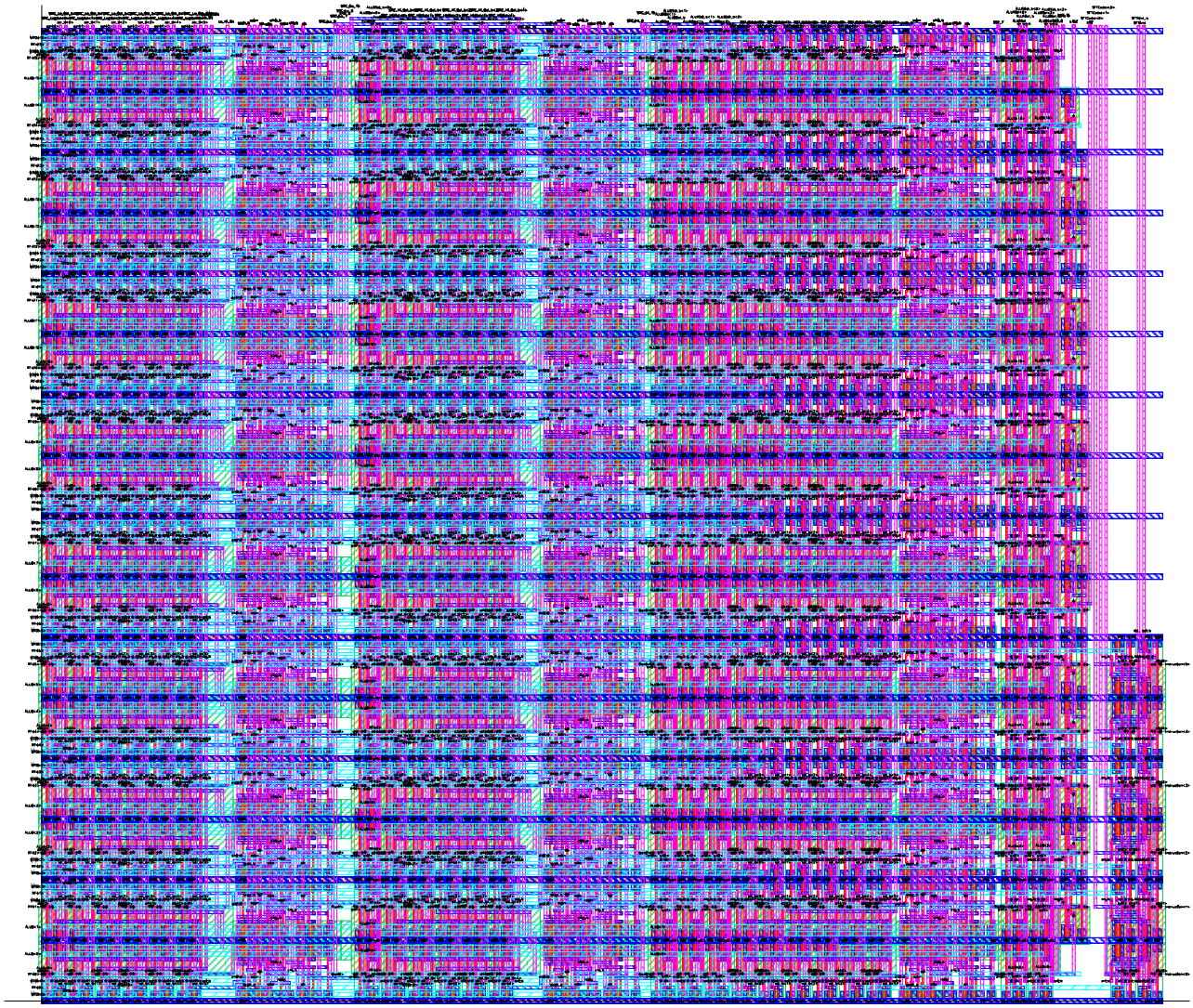
C.41 scanlatch_dp_1x



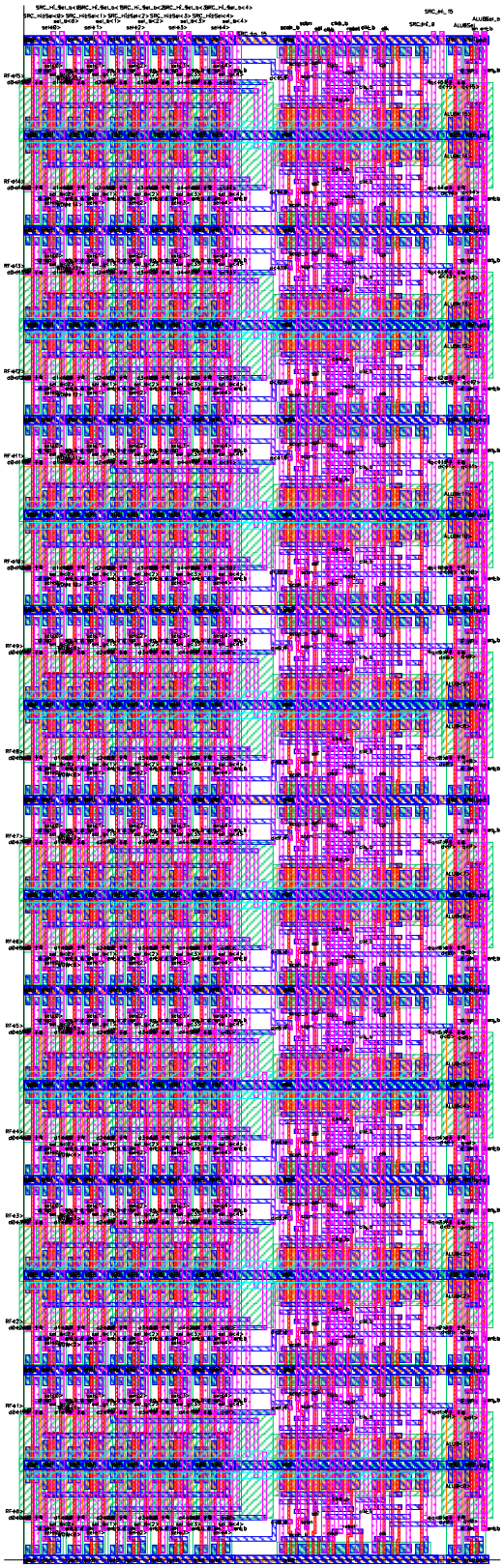
C.42 scanlatch_dp_1x_flip



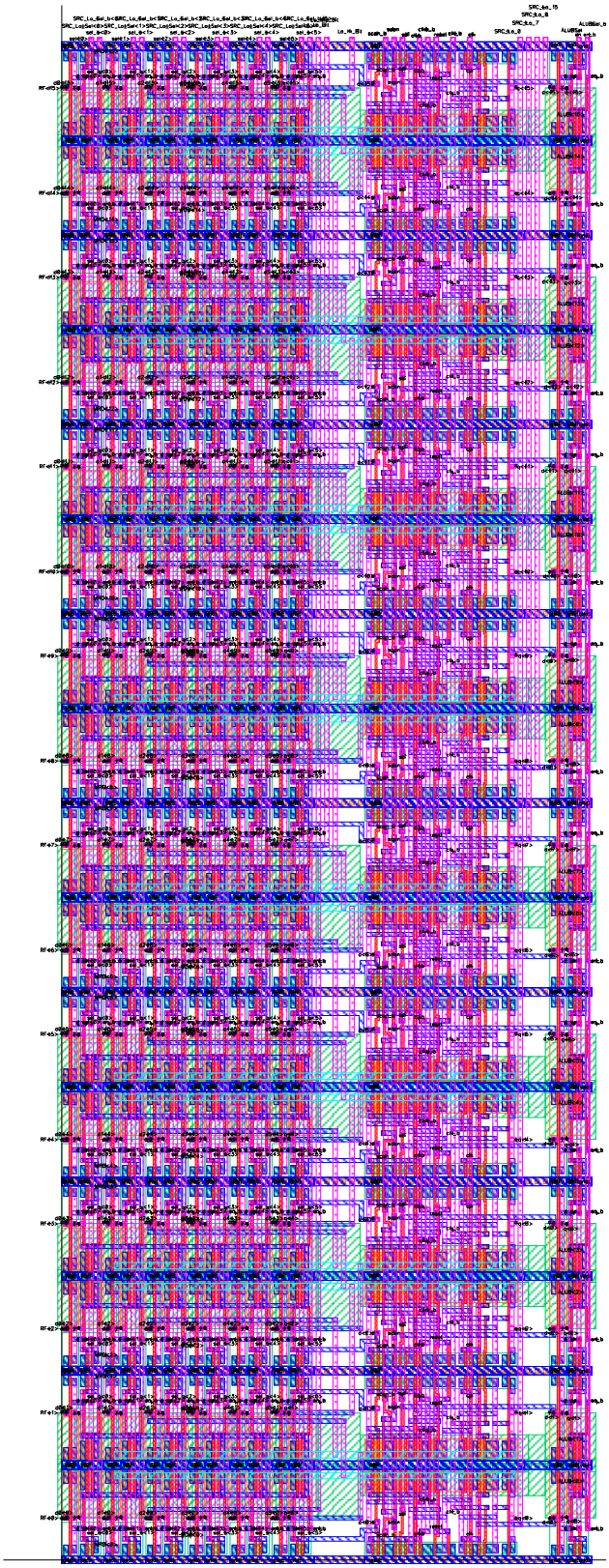
C.43 sdblock



C.44 SRC_Hi_Block_layout

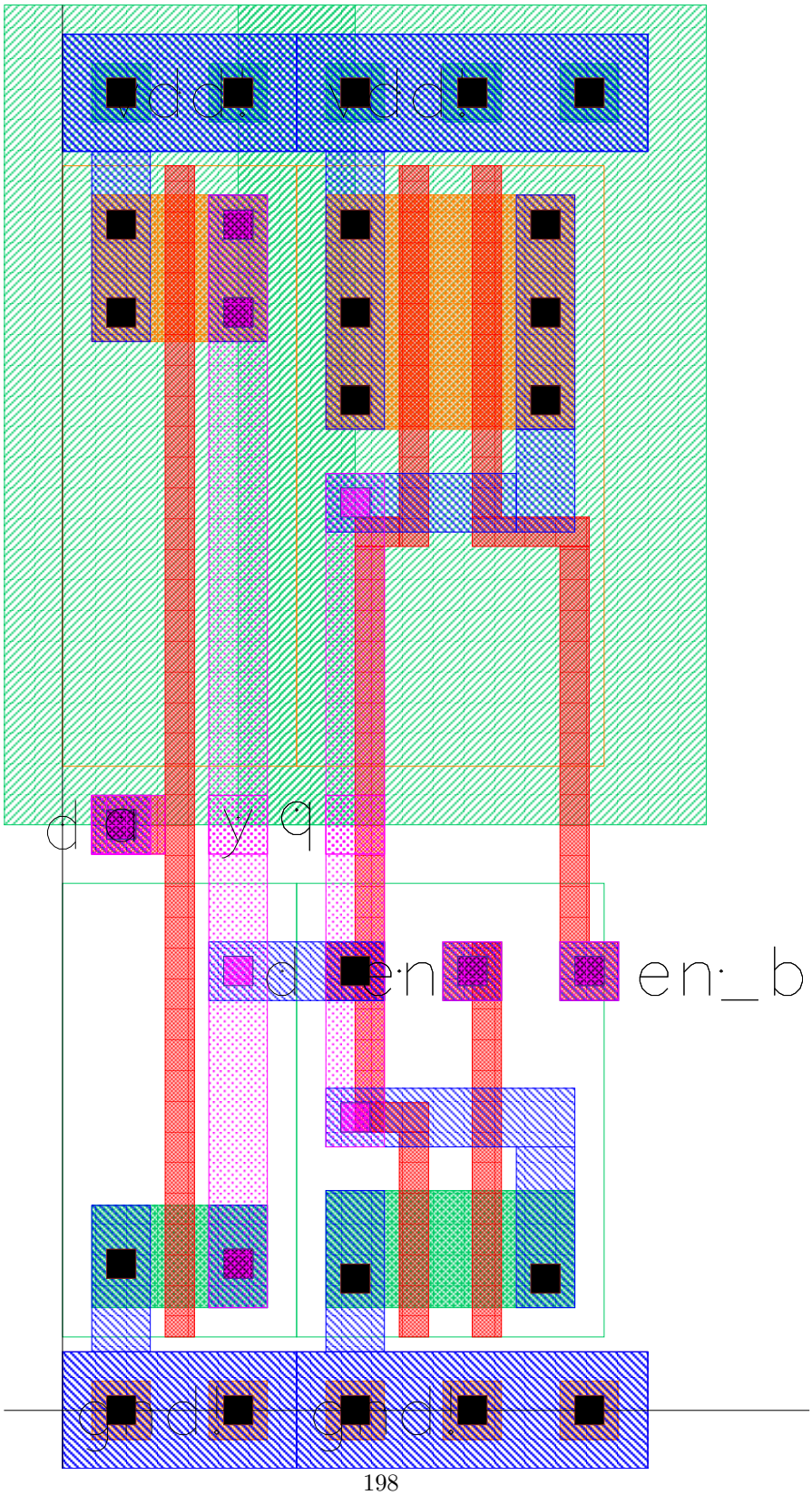


C.45 SRC_Lo_Block_layout

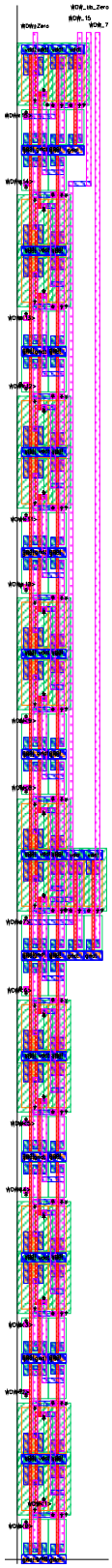


C.46 tri_dp_16_1x

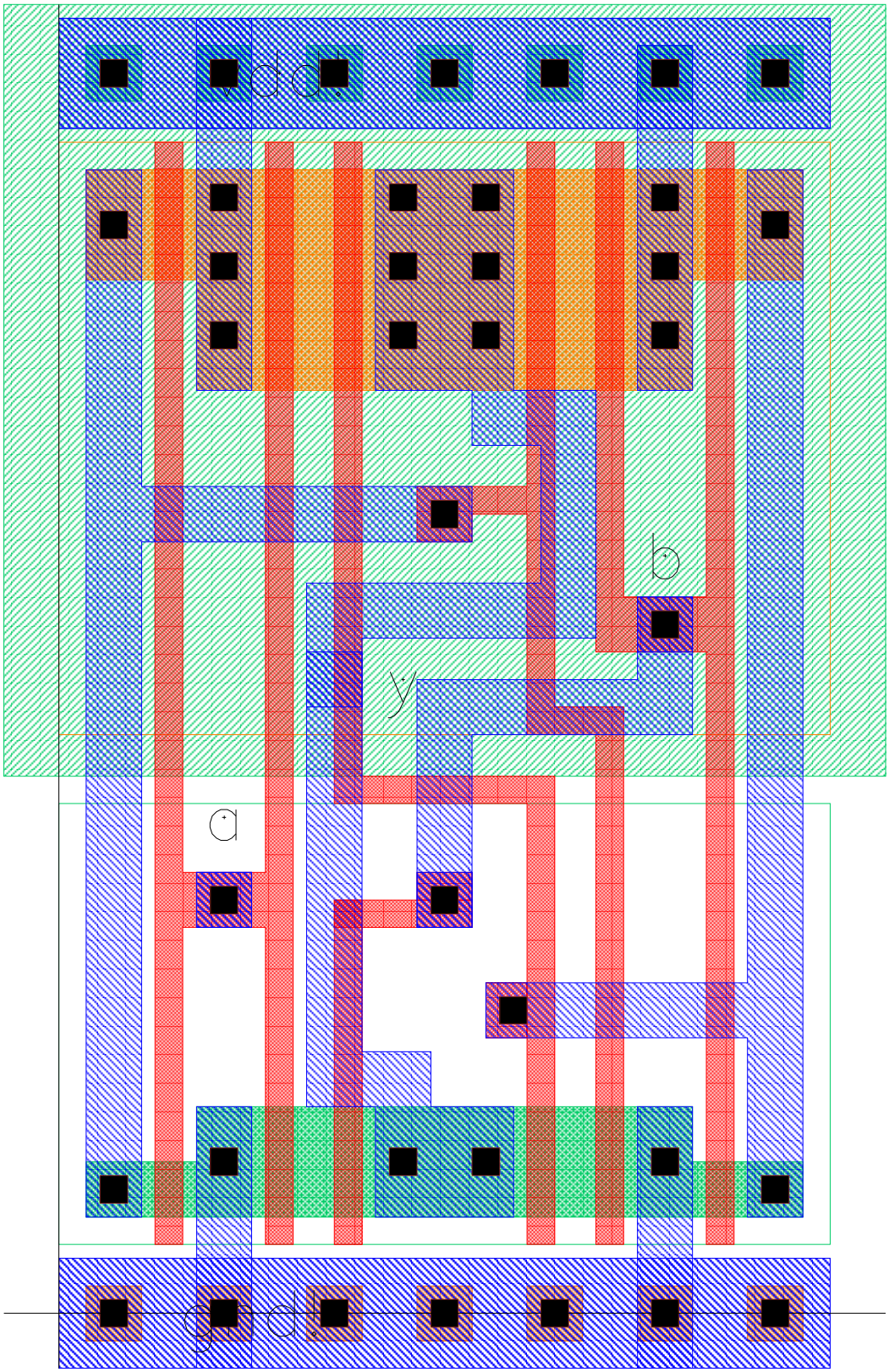
C.47 tri_dp_1x



C.48 WDW_Block_layout



C.49 xor_1x



C.50 xor_m2_1x