Basic Logic

```
module not_nand (a, o);
   input a;
   output o;

   nand2$ n1(o, a, a);
endmodule

module xor_nand (a, b, o);
   input a, b;
   output o;

   wire a_bar, b_bar, inter1, inter2;

   not_nand nn1(a, a_bar), nn2(b, b_bar);
   nand2$   n1(inter1, a_bar, b),
            n2(inter2, b_bar, a),
            n3(o, inter1, inter2);
endmodule

module and_nand (a, b, o);
   input a, b;
   output o;

   wire inter;

   nand2$ n1(inter, a, b);
   not_nand nn1(inter, o);
endmodule

module or_nand(a, b, o);
   input a, b;
   output o;

   wire a_bar, b_bar;

   not_nand    nn1(a, a_bar),
               nn2(b, b_bar);

   nand2$   n1(o, a_bar, b_bar);

endmodule
```

Components

```
// Structural for full adder
module full_adder (a, b, cin, s, cout);
   input a, b, cin;
```

```
    output s, cout;

    wire inter_s;
    xor_nand    x1(a, b, inter_s);
    xor_nand    x2(inter_s, cin, s);


    wire n_ab, n_ac, n_bc;
    nand2$   n1(n_ab, a, b),
             n2(n_ac, a, cin),
             n3(n_bc, b, cin);

    // 3 input OR-nand structure
    wire inter1, inter1_bar;
    nand2$   n4(inter1, n_ab, n_ac),
             n5(inter1_bar, inter1, inter1),
             n6(cout, inter1_bar, n_bc);
endmodule

module MUX2 (i0, i1, s, o);
    input i0, i1, s;
    output o;

    wire sbar, inter1, inter2;

    not_nand nn1(s, sbar);

    nand2$   n1(inter1, i0, sbar),
             n2(inter2, i1, s),
             n3(o, inter1, inter2);
endmodule

module MUX2_16 (i0, i1, s, o);
    input [15:0] i0, i1;
    input s;
    output [15:0] o;

    wire [15:0] inter1, inter2;
    wire sbar;

    not_nand nn1(s, sbar);

    genvar i;
    generate
       for (i = 0; i < 16; i = i + 1) begin : mux2_gate_gen
          nand2$   n1(inter1[i], i0[i], sbar),
                   n2(inter2[i], i1[i], s),
                   n3(o[i], inter1[i], inter2[i]);
       end
    endgenerate
endmodule

module MUX4_16 (i0, i1, i2, i3, s, o);
input [15:0] i0, i1, i2, i3;
```

```
    input [1:0] s;
    output [15:0] o;

    wire [3:0] select_line;
    wire s0bar, s1bar;
    not_nand nn1(s[0], s0bar),
             nn2(s[1], s1bar);

    and_nand select0(s1bar, s0bar, select_line[0]),
             select1(s1bar, s[0], select_line[1]),
             select2(s[1], s0bar, select_line[2]),
             select3(s[1], s[0], select_line[3]);


    wire [15:0] inter0, inter1, inter2, inter3, or_inter0, or_inter1;
    genvar i;
    generate
       for (i = 0; i < 16; i = i + 1) begin : gen1
          and_nand    an0_i(i0[i], select_line[0], inter0[i]),
                      an1_i(i1[i], select_line[1], inter1[i]),
                      an2_i(i2[i], select_line[2], inter2[i]),
                      an3_i(i3[i], select_line[3], inter3[i]);
       end
    endgenerate

    generate
       for (i = 0; i < 16; i = i + 1) begin : gen2
          or_nand     on0_i(inter0[i], inter1[i], or_inter0[i]),
                      on1_i(inter2[i], inter3[i], or_inter1[i]);
       end
    endgenerate

    generate
       for (i = 0; i < 16; i = i + 1) begin : gen3
          or_nand     on0_i(or_inter0[i], or_inter1[i], o[i]);
       end
    endgenerate

endmodule

module MUX4_8 (i0, i1, i2, i3, s, o);
    input [7:0] i0, i1, i2, i3;
    input [1:0] s;
    output [7:0] o;

    wire [3:0] select_line;
    wire s0bar, s1bar;
    not_nand nn1(s[0], s0bar),
             nn2(s[1], s1bar);

    and_nand select0(s1bar, s0bar, select_line[0]),
             select1(s1bar, s[0], select_line[1]),
             select2(s[1], s0bar, select_line[2]),
             select3(s[1], s[0], select_line[3]);
```

```
    wire [7:0] inter0, inter1, inter2, inter3, or_inter0, or_inter1;
    genvar i;
    generate
       for (i = 0; i < 8; i = i + 1) begin : gen1
          and_nand    an0_i(i0[i], select_line[0], inter0[i]),
                      an1_i(i1[i], select_line[1], inter1[i]),
                      an2_i(i2[i], select_line[2], inter2[i]),
                      an3_i(i3[i], select_line[3], inter3[i]);
       end
    endgenerate

    generate
       for (i = 0; i < 8; i = i + 1) begin : gen2
          or_nand     on0_i(inter0[i], inter1[i], or_inter0[i]),
                      on1_i(inter2[i], inter3[i], or_inter1[i]);
       end
    endgenerate

    generate
       for (i = 0; i < 8; i = i + 1) begin : gen3
          or_nand     on0_i(or_inter0[i], or_inter1[i], o[i]);
       end
    endgenerate

endmodule
```

ALU and Slices

```
module ALU (a, b, s, out);
    input [15:0] a, b; // 16-bit inputs
    input [1:0] s;     // 2-bit select
    output [15:0] out; // 16-bit outputs

    // AND == 00, NOT == 01, ADD == 10, SAT == 11
    wire [15:0] and_line, not_line, add_line, sat_line;

    AND_ALU_slice and_slice(a, b, and_line);
    NOT_ALU_slice not_slice(b, not_line);
    ADD_SAT_slice add_sat_slice(a, b, s, add_line, sat_line);

    MUX4_16 mx4_16_1(and_line, not_line, add_line, sat_line, s, out);
endmodule

module AND_ALU_slice (a, b, out);
    input [15:0] a, b;
    output [15:0] out;


    genvar i;
```

```
    generate
       for (i = 0; i < 16; i = i + 1) begin : gen1
          and_nand an_i(a[i], b[i], out[i]);
       end
    endgenerate
endmodule

module NOT_ALU_slice (b, out);
    input [15:0] b;
    output [15:0] out;

    genvar i;
    generate
       for (i = 0; i < 16; i = i + 1) begin : gen1
          not_nand nn_i(b[i], out[i]);
       end
    endgenerate
endmodule

module ADD_SAT_slice (a, b, s, add_out, sat_out);
    input [15:0] a, b;
    input [1:0] s;
    output [15:0] add_out, sat_out;

    wire [15:0] sum;
    wire [16:0] carry;
    wire ol, oh;

    assign add_out = sum;

    assign carry[0] = 0;

    genvar i;

    // lower 8 FA
    full_adder f_0(a[0], b[0], 1'b0, sum[0], carry[1]);
    generate
       for (i = 1; i < 8; i = i + 1) begin : gen1
          full_adder f_i(a[i], b[i], carry[i], sum[i], carry[i + 1]);
       end
    endgenerate

    wire select_and, carry_upper;
    and_nand an1(s[0], s[1], select_and);
    MUX2 mx1(carry[8], 1'b0, select_and, carry_upper);

    // upper 8 FA
    full_adder f_8(a[8], b[8], carry_upper, sum[8], carry[9]);
    generate
       for (i = 9; i < 16; i = i + 1) begin : gen2
          full_adder f_i(a[i], b[i], carry[i], sum[i], carry[i + 1]);
       end
    endgenerate
```

```
   // overflow bits
   xor_nand x1(carry[7], carry[8], ol),
            x2(carry[15], carry[16], oh);

   // Saturation Mux
   MUX4_8   mx4_8_1(sum[7:0], sum[7:0], 8'h7f, 8'h80, {ol, a[7]},
sat_out[7:0]),
            mx4_8_2(sum[15:8], sum[15:8], 8'h7f, 8'h80, {oh, a[15]},
sat_out[15:8]);


endmodule
```

Register

```
module d_latch (d, q, qbar, wen);
   input d, wen;
   output q, qbar;

   wire   dbar, r, s;

   inv1$ inv1 (dbar, d);
   nand2$ nand1 (s, d, wen),
          nand2 (r, dbar, wen),

          nand3 (q, s, qbar),
          nand4 (qbar, r, q);

endmodule

module d_flip_flop (d, q, clk);
   input d, clk;
   output q;

   wire clk_bar;
   wire q_master, qbar_master, qbar_slave;

   not_nand nn1(clk, clk_bar);

   // master latch, latches on falling edge
   d_latch  m_latch(d, q_master, qbar_master, clk_bar);
   // slave latch
   d_latch  s_latch(q_master, q, qbar_slave, clk);

endmodule

module register_16(d, q, enable, clk);
   input [15:0] d;
   input enable, clk;
   output [15:0] q;
```

```
    wire [15:0] q_bar;

    wire wen;

    and_nand an1(enable, clk, wen);

    // set 16 D flip-flops
    genvar i;
    generate
       for (i = 0; i < 16; i = i + 1) begin : reg_ff_gen
          d_flip_flop dff(d[i], q[i], clk);
       end
    endgenerate
 endmodule
```