

Appendix A. Verilog Code of Design Examples

The next pages contain the Verilog 1364-2001 code of all design examples. The old style Verilog 1364-1995 code can be found in [441]. The synthesis results for the examples are listed on page 881.

```
//*****
// IEEE STD 1364-2001 Verilog file: example.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module example //----> Interface
    #(parameter WIDTH =8) // Bit width
    (input clk, // System clock
     input reset, // Asynchronous reset
     input [WIDTH-1:0] a, b, op1, // Vector type inputs
     output [WIDTH-1:0] sum, // Vector type inputs
     output [WIDTH-1:0] c, // Integer output
     output reg [WIDTH-1:0] d); // Integer output
// -----
    reg [WIDTH-1:0] s; // Infer FF with always
    wire [WIDTH-1:0] op2, op3;
    wire [WIDTH-1:0] a_in, b_in;

    assign op2 = b; // Only one vector type in Verilog;
                  // no conversion int -> logic vector necessary

    lib_add_sub add1 //----> Component instantiation
    ( .result(op3), .dataa(op1), .datab(op2));
    defparam add1.lpm_width = WIDTH;
    defparam add1.lpm_direction = "SIGNED";

    lib_ff reg1
    ( .data(op3), .q(sum), .clock(clk)); // Used ports
    defparam reg1.lpm_width = WIDTH;

    assign c = a + b; //----> Data flow style (concurrent)
    assign a_i = a; // Order of statement does not
```

```

assign b_i = b; // matter in concurrent code

//----> Behavioral style
always @(posedge clk or posedge reset)
begin : p1          // Infer register
    reg [WIDTH-1:0] s;
    if (reset) begin
        s = 0; d = 0;
    end else begin
        //s <= s + a_i;          // Signal assignment statement
        // d = s;
        s = s + b_i;
        d = s;
    end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fun_text.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// A 32 bit function generator using accumulator and ROM
// -----
module fun_text          //----> Interface
    #(parameter WIDTH = 32) // Bit width
    (input  clk,           // System clock
     input  reset,         // Asynchronous reset
     input  [WIDTH-1:0] M, // Accumulator increment
     output reg [7:0] sin,  // System sine output
     output [7:0] acc);    // Accumulator MSBs
// -----
    reg [WIDTH-1:0] acc32;
    wire [7:0] msbs;          // Auxiliary vectors
    reg [7:0] rom[255:0];

    always @(posedge clk or posedge reset)
        if (reset == 1)
            acc32 <= 0;
        else begin
            acc32 <= acc32 + M; //-- Add M to acc32 and
            end                //-- store in register

    assign msbs = acc32[WIDTH-1:WIDTH-8];

```

```

    assign acc = msbs;

    initial
    begin
$readmemh("sine256x8.txt", rom);
    end

    always @ (posedge clk)
    begin
        sin <= rom[msbs];
    end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cmul7p8.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module cmul7p8                                // -----> Interface
    (input  signed [4:0] x,                    // System input
     output signed [4:0] y0, y1, y2, y3);
    // The 4 system outputs y=7*x/8
// -----
    assign y0 = 7 * x / 8;
    assign y1 = x / 8 * 7;
    assign y2 = x/2 + x/4 + x/8;
    assign y3 = x - x/8;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: add1p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module add1p
#(parameter WIDTH    = 19, // Total bit width
          WIDTH1     = 9,  // Bit width of LSBs
          WIDTH2     = 10) // Bit width of MSBs
    (input  [WIDTH-1:0] x, y, // Inputs
     output [WIDTH-1:0] sum, // Result
     input          clk,    // System clock
     output          LSBs_carry); // Test port

    reg [WIDTH1-1:0] l1, l2, s1; // LSBs of inputs

```

```

    reg [WIDTH1:0] r1;          // LSBs of inputs
    reg [WIDTH2-1:0] l3, l4, r2, s2; // MSBs of input
// -----
    always @(posedge clk) begin
        // Split in MSBs and LSBs and store in registers
        // Split LSBs from input x,y
        l1[WIDTH1-1:0] <= x[WIDTH1-1:0];
        l2[WIDTH1-1:0] <= y[WIDTH1-1:0];
        // Split MSBs from input x,y
        l3[WIDTH2-1:0] <= x[WIDTH2-1+WIDTH1:WIDTH1];
        l4[WIDTH2-1:0] <= y[WIDTH2-1+WIDTH1:WIDTH1];
    /****** First stage of the adder *****/
        r1 <= {1'b0, l1} + {1'b0, l2};
        r2 <= l3 + l4;
    /****** Second stage of the adder *****/
        s1 <= r1[WIDTH1-1:0];
        // Add MSBs (x+y) and carry from LSBs
        s2 <= r1[WIDTH1] + r2;
    end

    assign LSBs_carry = r1[WIDTH1]; // Add a test signal

// Build a single registered output word
// of WIDTH = WIDTH1 + WIDTH2
    assign sum = {s2, s1};

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: add2p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// 22-bit adder with two pipeline stages
// uses no components
module add2p
#(parameter WIDTH   = 28,    // Total bit width
   WIDTH1   = 9,    // Bit width of LSBs
   WIDTH2   = 9,    // Bit width of middle
   WIDTH12  = 18,    // Sum WIDTH1+WIDTH2
   WIDTH3   = 10)    // Bit width of MSBs
(input  [WIDTH-1:0] x, y,    // Inputs
 output [WIDTH-1:0] sum,    // Result
 output LSBs_carry, MSBs_carry, // Carry test bits
 input  clk);              // System clock
// -----

```

```

reg [WIDTH1-1:0] l1, l2, v1, s1; // LSBs of inputs
reg [WIDTH1:0]    q1;           // LSBs of inputs
reg [WIDTH2-1:0] l3, l4, s2;    // Middle bits
reg [WIDTH2:0]    q2, v2;       // Middle bits
reg [WIDTH3-1:0] l5, l6, q3, v3, s3; // MSBs of input

// Split in MSBs and LSBs and store in registers
always @(posedge clk) begin
    // Split LSBs from input x,y
    l1[WIDTH1-1:0] <= x[WIDTH1-1:0];
    l2[WIDTH1-1:0] <= y[WIDTH1-1:0];
    // Split middle bits from input x,y
    l3[WIDTH2-1:0] <= x[WIDTH2-1+WIDTH1:WIDTH1];
    l4[WIDTH2-1:0] <= y[WIDTH2-1+WIDTH1:WIDTH1];
    // Split MSBs from input x,y
    l5[WIDTH3-1:0] <= x[WIDTH3-1+WIDTH12:WIDTH12];
    l6[WIDTH3-1:0] <= y[WIDTH3-1+WIDTH12:WIDTH12];
    //***** First stage of the adder *****
    q1 <= {1'b0, l1} + {1'b0, l2}; // Add LSBs of x and y
    q2 <= {1'b0, l3} + {1'b0, l4}; // Add LSBs of x and y
    q3 <= l5 + l6;                // Add MSBs of x and y
    //***** Second stage of the adder *****
    v1 <= q1[WIDTH1-1:0];          // Save q1
    // Add result from middle bits (x+y) and carry from LSBs
    v2 <= q1[WIDTH1] + {1'b0, q2[WIDTH2-1:0]};
    // Add result from MSBs bits (x+y) and carry from middle
    v3 <= q2[WIDTH2] + q3;
    //***** Third stage of the adder *****
    s1 <= v1;                      // Save v1
    s2 <= v2[WIDTH2-1:0];          // Save v2
    // Add result from MSBs bits (x+y) and 2. carry from middle
    s3 <= v2[WIDTH2] + v3;
end

assign LSBs_carry = q1[WIDTH1]; // Provide test signals
assign MSBs_carry = v2[WIDTH2];

// Build a single output word of WIDTH=WIDTH1+WIDTH2+WIDTH3
assign sum ={s3, s2, s1}; // Connect sum to output pins

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: add3p.v

```

```

// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// 37-bit adder with three pipeline stage
// uses no components

module add3p
#(parameter WIDTH    = 37, // Total bit width
    WIDTH0    = 9, // Bit width of LSBs
    WIDTH1    = 9, // Bit width of 2. LSBs
    WIDTH01    = 18, // Sum WIDTH0+WIDTH1
    WIDTH2    = 9, // Bit width of 2. MSBs
    WIDTH012    = 27, // Sum WIDTH0+WIDTH1+WIDTH2
    WIDTH3    = 10) // Bit width of MSBs
(input [WIDTH-1:0] x, y, // Inputs
    output [WIDTH-1:0] sum, // Result
    output LSBs_Carry, Middle_Carry, MSBs_Carry, // Test pins
    input clk); // Clock
// -----
    reg [WIDTH0-1:0] 10, 11, r0, v0, s0; // LSBs of inputs
    reg [WIDTH0:0] q0; // LSBs of inputs
    reg [WIDTH1-1:0] 12, 13, r1, s1; // 2. LSBs of input
    reg [WIDTH1:0] v1, q1; // 2. LSBs of input
    reg [WIDTH2-1:0] 14, 15, s2, h7; // 2. MSBs bits
    reg [WIDTH2:0] q2, v2, r2; // 2. MSBs bits
    reg [WIDTH3-1:0] 16, 17, q3, v3, r3, s3, h8;
                                // MSBs of input

always @(posedge clk) begin
// Split in MSBs and LSBs and store in registers
// Split LSBs from input x,y
10[WIDTH0-1:0] <= x[WIDTH0-1:0];
11[WIDTH0-1:0] <= y[WIDTH0-1:0];
// Split 2. LSBs from input x,y
12[WIDTH1-1:0] <= x[WIDTH1-1+WIDTH0:WIDTH0];
13[WIDTH1-1:0] <= y[WIDTH1-1+WIDTH0:WIDTH0];
// Split 2. MSBs from input x,y
14[WIDTH2-1:0] <= x[WIDTH2-1+WIDTH01:WIDTH01];
15[WIDTH2-1:0] <= y[WIDTH2-1+WIDTH01:WIDTH01];
// Split MSBs from input x,y
16[WIDTH3-1:0] <= x[WIDTH3-1+WIDTH012:WIDTH012];
17[WIDTH3-1:0] <= y[WIDTH3-1+WIDTH012:WIDTH012];

//***** First stage of the adder *****
q0 <= {1'b0, 10} + {1'b0, 11}; // Add LSBs of x and y

```

```

    q1 <= {1'b0, 12} + {1'b0, 13}; // Add 2. LSBs of x / y
    q2 <= {1'b0, 14} + {1'b0, 15}; // Add 2. MSBs of x/y
    q3 <= 16 + 17;                // Add MSBs of x and y
//***** Second stage of the adder *****
    v0 <= q0[WIDTH0-1:0];          // Save q0
// Add result from 2. LSBs (x+y) and carry from LSBs
    v1 <= q0[WIDTH0] + {1'b0, q1[WIDTH1-1:0]};
// Add result from 2. MSBs (x+y) and carry from 2. LSBs
    v2 <= q1[WIDTH1] + {1'b0, q2[WIDTH2-1:0]};
// Add result from MSBs (x+y) and carry from 2. MSBs
    v3 <= q2[WIDTH2] + q3;

//***** Third stage of the adder *****
    r0 <= v0; // Delay for LSBs
    r1 <= v1[WIDTH1-1:0]; // Delay for 2. LSBs
// Add result from 2. MSBs (x+y) and carry from 2. LSBs
    r2 <= v1[WIDTH1] + {1'b0, v2[WIDTH2-1:0]};
// Add result from MSBs (x+y) and carry from 2. MSBs
    r3 <= v2[WIDTH2] + v3;
//***** Fourth stage of the adder *****
    s0 <= r0; // Delay for LSBs
    s1 <= r1; // Delay for 2. LSBs
    s2 <= r2[WIDTH2-1:0]; // Delay for 2. MSBs
// Add result from MSBs (x+y) and carry from 2. MSBs
    s3 <= r2[WIDTH2] + r3;
end

assign LSBs_Carry   = q0[WIDTH1]; // Provide test signals
assign Middle_Carry = v1[WIDTH1];
assign MSBs_Carry   = r2[WIDTH2];

// Build a single output word of
// WIDTH = WIDTH0 + WIDTH1 + WIDTH2 + WIDTH3
assign sum = {s3, s2, s1, s0}; // Connect sum to output

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: div_res.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Restoring Division
// Bit width:  WN          WD          WN          WD
//              Nominator / Denominator = Quotient and Remainder

```

```

// OR:      Nominator = Quotient * Denominator + Remainder
// -----
module div_res                //-----> Interface
    (input clk,                // System clock
     input reset,              // Asynchron reset
     input  [7:0] n_in,        // Nominator
     input  [5:0] d_in,        // Denominator
     output reg [5:0] r_out,    // Remainder
     output reg [7:0] q_out);   // Quotient
// -----

    reg [1:0] state;           // FSM state
    parameter ini=0, sub=1, restore=2, done=3; // State
                                           // assignments

    // Divider in behavioral style
    always @(posedge clk or posedge reset)
    begin : States // Finite state machine
        reg [3:0] count;

        reg [13:0] d;          // Double bit width unsigned
        reg signed [13:0] r;    // Double bit width signed
        reg [7:0] q;

        if (reset) begin          // Asynchronous reset
            state <= ini; count <= 0;
            q <= 0; r <= 0; d <= 0; q_out <= 0; r_out <= 0;
        end else
        case (state)
            ini : begin           // Initialization step
                state <= sub;
                count = 0;
                q <= 0;           // Reset quotient register
                d <= d_in << 7;    // Load aligned denominator
                r <= n_in;         // Remainder = nominator
            end
            sub : begin           // Processing step
                r <= r - d;        // Subtract denominator
                state <= restore;
            end
            restore : begin       // Restoring step
                if (r < 0) begin // Check r < 0
                    r <= r + d;    // Restore previous remainder
                    q <= q << 1;    // LSB = 0 and SLL
                end
            end
            else

```



```

        q <= (q << 1) + 1; // LSB = 1 and SLL
        count = count + 1;
        d <= d >> 1;

        if (count == 8) // Division ready ?
            state <= done;
        else
            state <= sub;
        end
    done : begin // Output of result
        q_out <= q[7:0];
        r_out <= r[5:0];
        state <= ini; // Start next division
    end
endcase
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: div_aegp.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Convergence division after
// Anderson, Earle, Goldschmidt, and Powers
// Bit width: WN WD WN WD
// Nominator / Denominator = Quotient and Remainder
// OR: Nominator = Quotient * Denominator + Remainder
// -----
module div_aegp
    (input clk, // System clock
     input reset, // Asynchron reset
     input [8:0] n_in, // Nominator
     input [8:0] d_in, // Denominator
     output reg [8:0] q_out); // Quotient
// -----
    reg [1:0] state;
    always @(posedge clk or posedge reset) //-> Divider in
    begin : States // behavioral style
        parameter s0=0, s1=1, s2=2;
        reg [1:0] count;

        reg [9:0] x, t, f; // one guard bit
        reg [17:0] tempx, tempt;

```

```

    if (reset) begin                // Asynchronous reset
        state <= s0; q_out <= 0; count = 0; x <= 0; t <= 0;
    end else
        case (state)
            s0 : begin                // Initialization step
                state <= s1;
                count = 0;
                t <= {1'b0, d_in};    // Load denominator
                x <= {1'b0, n_in};    // Load nominator
            end
            s1 : begin                // Processing step
                f = 512 - t;           // TWO - t
                tempx = (x * f);       // Product in full
                tempt = (t * f);       // bitwidth
                x <= tempx >> 8;       // Fractional f
                t <= tempt >> 8;       // Scale by 256
                count = count + 1;
                if (count == 2)        // Division ready ?
                    state <= s2;
                else
                    state <= s1;
            end
            s2 : begin                // Output of result
                q_out <= x[8:0];
                state <= s0;          // Start next division
            end
        endcase
    end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cordic.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module cordic #(parameter W = 7) // Bit width - 1
(input clk,                        // System clock
 input reset,                      // Asynchronous reset
 input signed [W:0] x_in,         // System real or x input
 input signed [W:0] y_in,         // System imaginary or y input
 output reg signed [W:0] r,        // Radius result
 output reg signed [W:0] phi,      // Phase result
 output reg signed [W:0] eps);    // Error of results
// -----

```

```

// There is bit access in Quartus array types
// in Verilog 2001, therefore use single vectors
// but use a separate lines for each array!
reg signed [W:0] x [0:3];
reg signed [W:0] y [0:3];
reg signed [W:0] z [0:3];

always @(posedge reset or posedge clk) begin : P1
    integer k; // Loop variable
    if (reset) begin // Asynchronous clear
        for (k=0; k<=3; k=k+1) begin
            x[k] <= 0; y[k] <= 0; z[k] <= 0;
        end
        r <= 0; eps <= 0; phi <= 0;
    end else begin
        if (x_in >= 0) // Test for x_in < 0 rotate
            begin // 0, +90, or -90 degrees
                x[0] <= x_in; // Input in register 0
                y[0] <= y_in;
                z[0] <= 0;
            end
        else if (y_in >= 0)
            begin
                x[0] <= y_in;
                y[0] <= - x_in;
                z[0] <= 90;
            end
        else
            begin
                x[0] <= - y_in;
                y[0] <= x_in;
                z[0] <= -90;
            end

        if (y[0] >= 0) // Rotate 45 degrees
            begin
                x[1] <= x[0] + y[0];
                y[1] <= y[0] - x[0];
                z[1] <= z[0] + 45;
            end
        else
            begin
                x[1] <= x[0] - y[0];

```

```

        y[1] <= y[0] + x[0];
        z[1] <= z[0] - 45;
    end

    if (y[1] >= 0)                                // Rotate 26 degrees
    begin
        x[2] <= x[1] + (y[1] >>> 1); // i.e.  $x[1]+y[1]/2$ 
        y[2] <= y[1] - (x[1] >>> 1); // i.e.  $y[1]-x[1]/2$ 
        z[2] <= z[1] + 26;
    end
    else
    begin
        x[2] <= x[1] - (y[1] >>> 1); // i.e.  $x[1]-y[1]/2$ 
        y[2] <= y[1] + (x[1] >>> 1); // i.e.  $y[1]+x[1]/2$ 
        z[2] <= z[1] - 26;
    end

    if (y[2] >= 0)                                // Rotate 14 degrees
    begin
        x[3] <= x[2] + (y[2] >>> 2); // i.e.  $x[2]+y[2]/4$ 
        y[3] <= y[2] - (x[2] >>> 2); // i.e.  $y[2]-x[2]/4$ 
        z[3] <= z[2] + 14;
    end
    else
    begin
        x[3] <= x[2] - (y[2] >>> 2); // i.e.  $x[2]-y[2]/4$ 
        y[3] <= y[2] + (x[2] >>> 2); // i.e.  $y[2]+x[2]/4$ 
        z[3] <= z[2] - 14;
    end

    r    <= x[3];
    phi  <= z[3];
    eps  <= y[3];
end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: arctan.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
// -----
module arctan #(parameter W = 9,           // Bit width
                L   = 5)                  // Array size
    (input clk,                          // System clock

```

```

input reset,                // Asynchron reset
input signed [W-1:0] x_in,  // System input
//output reg signed [W-1:0] d_o [1:L],
output wire signed [W-1:0] d_o1, d_o2 ,d_o3, d_o4 ,d_o5,
                        // Auxiliary recurrence
output reg signed [W-1:0] f_out); // System output
// -----
reg signed [W-1:0] x;    // Auxiliary signals
wire signed [W-1:0] f;
wire signed [W-1:0] d [1:L]; // Auxiliary array
// Chebychev coefficients c1, c2, c3 for 8 bit precision
// c1 = 212; c3 = -12; c5 = 1;

always @(posedge clk or posedge reset) begin
  if (reset) begin // Asynchronous clear
    x <= 0; f_out <= 0;
  end else begin
    x <= x_in;      // FF for input and output
    f_out <= f;
  end
end

// Compute sum-of-products with
// Clenshaw's recurrence formula
assign d[5] = 'sd1; // c5=1
assign d[4] = (x * d[5]) / 128;
assign d[3] = ((x * d[4]) / 128) - d[5] - 12; // c3=-12
assign d[2] = ((x * d[3]) / 128) - d[4];
assign d[1] = ((x * d[2]) / 128) - d[3] + 212; // c1=212
assign f     = ((x * d[1]) / 256) - d[2];
                        // last step is different

assign d_o1 = d[1]; // Provide test signals as outputs
assign d_o2 = d[2];
assign d_o3 = d[3];
assign d_o4 = d[4];
assign d_o5 = d[5];
endmodule

//*****
// IEEE STD 1364-2001 Verilog file: ln.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module ln #(parameter N = 5, // Number of Coefficients-1

```

```

        parameter W= 17) // Bitwidth -1
(input clk,                      // System clock
 input reset,                    // Asynchronous reset
 input signed [W:0] x_in,       // System input
 output reg signed [W:0] f_out); // System output
// -----
reg signed [W:0] x, f; // Auxiliary register
wire signed [W:0] p [0:5];
reg signed [W:0] s [0:5];

// Polynomial coefficients for 16 bit precision:
//  $f(x) = (1 + 65481 x - 32093 x^2 + 18601 x^3$ 
//  $- 8517 x^4 + 1954 x^5)/65536$ 
assign p[0] = 18'sd1;
assign p[1] = 18'sd65481;
assign p[2] = -18'sd32093;
assign p[3] = 18'sd18601;
assign p[4] = -18'sd8517;
assign p[5] = 18'sd1954;

always @(posedge clk or posedge reset)
begin : Store
    if (reset) begin // Asynchronous clear
        x <= 0; f_out <= 0;
    end else begin
        x <= x_in; // Store input in register
        f_out <= f;
    end
end

always @* // Compute sum-of-products
begin : SOP
    integer k; // define the loop variable
    reg signed [35:0] slv;

    s[N] = p[N];
// Polynomial Approximation from Chebyshev coefficients
    for (k=N-1; k>=0; k=k-1)
    begin
        slv = x * s[k+1]; // no FFs for slv
        s[k] = (slv >>> 16) + p[k];
    end // x*s/65536 problem 32 bits
    f <= s[0]; // make visable outside
end

```

```

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: sqrt.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module sqrt                                //----> Interface
(input clk,                                // System clock
 input reset,                              // Asynchronous reset
 output [1:0] count_o,                    // Counter SLL
 input signed [16:0] x_in,                // System input
 output signed [16:0] pre_o,              // Prescaler
 output signed [16:0] x_o,                // Normalized x_in
 output signed [16:0] post_o,             // Postscaler
 output signed [3:0] ind_o,               // Index to p
 output signed [16:0] imm_o,              // ALU preload value
 output signed [16:0] a_o,                // ALU factor
 output signed [16:0] f_o,                // ALU output
 output reg signed [16:0] f_out); // System output
// -----
// Define the operation modes:
parameter load=0, mac=1, scale=2, denorm=3, nop=4;
// Assign the FSM states:
parameter start=0, leftshift=1, sop=2,
               rightshift=3, done=4;

reg [3:0] s, op;
reg [16:0] x; // Auxilary
reg signed [16:0] a, b, f, imm; // ALU data
reg signed [33:0] af; // Product double width
reg [16:0] pre, post;
reg signed [3:0] ind;
reg [1:0] count;
// Chebychev poly coefficients for 16 bit precision:
wire signed [16:0] p [0:4];

assign p[0] = 7563;
assign p[1] = 42299;
assign p[2] = -29129;
assign p[3] = 15813;
assign p[4] = -3778;

always @(posedge reset or posedge clk) //-----> SQRT FSM
begin : States                          // sample at clk rate

```

```

if (reset) begin                                // Asynchronous reset
    s <= start; f_out <= 0; op <= 0; count <= 0;
    imm <= 0; ind <= 0; a <= 0; x <= 0;
end else begin
    case (s)                                     // Next State assignments
        start : begin                           // Initialization step
            s <= leftshift; ind = 4;
            imm <= x_in;                        // Load argument in ALU
            op <= load; count = 0;
        end
        leftshift : begin                       // Normalize to 0.5 .. 1.0
            count = count + 1; a <= pre; op <= scale;
            imm <= p[4];
            if (count == 2) op <= nop;
            if (count == 3) begin // Normalize ready ?
                s <= sop; op <= load; x <= f;
            end
        end
        sop : begin                             // Processing step
            ind = ind - 1; a <= x;
            if (ind == -1) begin // SOP ready ?
                s <= rightshift; op <= denorm; a <= post;
            end else begin
                imm <= p[ind]; op <= mac;
            end
        end
        rightshift : begin // Denormalize to original range
            s <= done; op <= nop;
        end
        done : begin                            // Output of results
            f_out <= f;                          // I/O store in register
            op <= nop;
            s <= start;                          // start next cycle
        end
    endcase
end
end

always @(posedge reset or posedge clk)
begin : ALU                                     // Define the ALU operations
    if (reset)                                // Asynchronous clear
        f <= 0;
    else begin

```



```

    af = a * f;
    case (op)
        load      : f <= imm;
        mac       : f <= (af >>> 15) + imm;
        scale     : f <= af;
        denorm    : f <= af >>> 15;
        nop       : f <= f;
        default   : f <= f;
    endcase
end
end

always @(x_in)
begin : EXP
    reg [16:0] slv;
    reg [16:0] po, pr;
    integer K; // Loop variable

    slv = x_in;
    // Compute pre-scaling:
    for (K=0; K <= 15; K= K+1)
        if (slv[K] == 1)
            pre = 1 << (14-K);
    // Compute post scaling:
    po = 1;
    for (K=0; K <= 7; K= K+1) begin
        if (slv[2*K] == 1) // even 2^k gets 2^k/2
            po = 1 << (K+8);
    // sqrt(2): CSD Error = 0.0000208 = 15.55 effective bits
    // +1 +0. -1 +0 -1 +0 +1 +0 +1 +0 +0 +0 +0 +0 +1
    // 9      7      5      3      1      -5
        if (slv[2*K+1] == 1) // odd k has sqrt(2) factor
            po = (1<<(K+9)) - (1<<(K+7)) - (1<<(K+5))
                + (1<<(K+3)) + (1<<(K+1)) + (1<<(K-5));
    end
    post <= po;
end

assign a_o = a; // Provide some test signals as outputs
assign imm_o = imm;
assign f_o = f;
assign pre_o = pre;
assign post_o = post;
assign x_o = x;

```

```

    assign ind_o = ind;
    assign count_o = count;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file:  magnitude.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module magnitude
    (input clk,                // System clock
     input reset,              // Asynchron reset
     input signed [15:0] x, y, // System input
     output reg signed [15:0] r); // System output
// -----
    reg signed [15:0] x_r, y_r, ax, ay, mi, ma;
    // Approximate the magnitude via
    //  $r = \alpha \cdot \max(|x|, |y|) + \beta \cdot \min(|x|, |y|)$ 
    // use  $\alpha=1$  and  $\beta=1/4$ 

    always @(posedge reset or posedge clk) // Control the
        if (reset) begin                // system sample at clk rate
            x_r <= 0; y_r <= 0; // Asynchronous clear
        end else begin
            x_r <= x; y_r <= y;
        end

    always @* begin
        ax = (x_r>=0)? x_r : -x_r; // Take absolute values first
        ay = (y_r>=0)? y_r : -y_r;

        if (ax > ay) begin // Determine max and min values
            mi = ay;
            ma = ax;
        end else begin
            mi = ax;
            ma = ay;
        end
    end

    always @(posedge reset or posedge clk)
        if (reset) // Asynchronous clear
            r <= 0;
        else

```

```

        r <= ma + mi/4; // Compute r=alpha*max+beta*min

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fir_gen.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// This is a generic FIR filter generator
// It uses W1 bit data/coefficients bits
module fir_gen
#(parameter W1 = 9,      // Input bit width
        W2 = 18,      // Multiplier bit width 2*W1
        W3 = 19,      // Adder width = W2+log2(L)-1
        W4 = 11,      // Output bit width
        L = 4)      // Filter length
(input clk,                // System clock
 input reset,              // Asynchronous reset
 input Load_x,             // Load/run switch
 input signed [W1-1:0] x_in, // System input
 input signed [W1-1:0] c_in, //Coefficient data input
 output signed [W4-1:0] y_out); // System output
// -----
    reg signed [W1-1:0] x;
    wire signed [W3-1:0] y;
// 1D array types i.e. memories supported by Quartus
// in Verilog 2001; first bit then vector size
    reg signed [W1-1:0] c [0:3]; // Coefficient array
    wire signed [W2-1:0] p [0:3]; // Product array
    reg signed [W3-1:0] a [0:3]; // Adder array

//----> Load Data or Coefficient
    always @(posedge clk or posedge reset)
        begin: Load
            integer k;      // loop variable
            if (reset) begin // Asynchronous clear
                for (k=0; k<=L-1; k=k+1) c[k] <= 0;
                x <= 0;
            end else if (! Load_x) begin
                c[3] <= c_in; // Store coefficient in register
                c[2] <= c[3]; // Coefficients shift one
                c[1] <= c[2];
                c[0] <= c[1];
            end else

```

```

        x <= x_in; // Get one data sample at a time
    end

//----> Compute sum-of-products
    always @(posedge clk or posedge reset)
    begin: SOP
        // Compute the transposed filter additions
        integer k;    // loop variable
        if (reset)    // Asynchronous clear
            for (k=0; k<=3; k=k+1) a[k] <= 0;
        else begin
            a[0] <= p[0] + a[1];
            a[1] <= p[1] + a[2];
            a[2] <= p[2] + a[3];
            a[3] <= p[3]; // First TAP has only a register
        end
    end
    end
    assign y = a[0];

    genvar I; //Define loop variable for generate statement
    generate
        for (I=0; I<L; I=I+1) begin : MulGen
            // Instantiate L multipliers
            assign p[I] = x * c[I];
        end
    endgenerate

    assign y_out = y[W3-1:W3-W4];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fir_srg.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module fir_srg //----> Interface
    (input clk, // System clock
     input reset, // Asynchronous reset
     input signed [7:0] x, // System input
     output reg signed [7:0] y); // System output
// -----
// Tapped delay line array of bytes
    reg signed [7:0] tap [0:3];
    integer I; // Loop variable

```

```

always @(posedge clk or posedge reset)
begin : P1 //----> Behavioral Style
// Compute output y with the filter coefficients weight.
// The coefficients are [-1 3.75 3.75 -1].
// Multiplication and division can
// be done in Verilog 2001 with signed shifts.
if (reset) begin // Asynchronous clear
for (I=0; I<=3; I=I+1) tap[I] <= 0;
y <= 0;
end else begin
y <= (tap[1] <<< 1) + tap[1] + (tap[1] >>> 1) - tap[0]
+ ( tap[1] >>> 2) + (tap[2] <<< 1) + tap[2]
+ (tap[2] >>> 1) + (tap[2] >>> 2) - tap[3];

for (I=3; I>0; I=I-1) begin
tap[I] <= tap[I-1]; // Tapped delay line: shift one
end
tap[0] <= x; // Input in register 0
end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: case5p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module case5p
(input      clk,
input  [4:0] table_in,
output reg [4:0] table_out); // range 0 to 25
// -----
reg [3:0] lsbs;
reg [1:0] msbs0;
reg [4:0] table0out00, table0out01;

// These are the distributed arithmetic CASE tables for
// the 5 coefficients: 1, 3, 5, 7, 9

always @(posedge clk) begin
lsbs[0] = table_in[0];
lsbs[1] = table_in[1];
lsbs[2] = table_in[2];

```

```

    lsbs[3] = table_in[3];
    msbs0[0] = table_in[4];
    msbs0[1] = msbs0[0];
end

// This is the final DA MPX stage.
always @(posedge clk) begin
    case (msbs0[1])
        0 : table_out <= table0out00;
        1 : table_out <= table0out01;
        default : ;
    endcase
end

// This is the DA CASE table 00 out of 1.
always @(posedge clk) begin
    case (lsbs)
        0 : table0out00 = 0;
        1 : table0out00 = 1;
        2 : table0out00 = 3;
        3 : table0out00 = 4;
        4 : table0out00 = 5;
        5 : table0out00 = 6;
        6 : table0out00 = 8;
        7 : table0out00 = 9;
        8 : table0out00 = 7;
        9 : table0out00 = 8;
        10 : table0out00 = 10;
        11 : table0out00 = 11;
        12 : table0out00 = 12;
        13 : table0out00 = 13;
        14 : table0out00 = 15;
        15 : table0out00 = 16;
        default ;
    endcase
end

// This is the DA CASE table 01 out of 1.
always @(posedge clk) begin
    case (lsbs)
        0 : table0out01 = 9;
        1 : table0out01 = 10;
        2 : table0out01 = 12;
        3 : table0out01 = 13;

```



```

integer k;

reg [2:0] count;           // Counts the shifts
reg [6:0] p;               // Temporary register

if (reset) begin           // Asynchronous reset
    state <= s0;
    x0 <= 0; x1 <= 0; x2 <= 0; p <= 0; y <= 0;
    count <= 0;
end else
    case (state)
        s0 : begin         // Initialization step
            state <= s1;
            count = 0;
            p <= 0;
            x0 <= x0_in;
            x1 <= x1_in;
            x2 <= x2_in;
        end
        s1 : begin         // Processing step
            if (count == 4) begin // Is sum of product done?
                y <= p;         // Output of result to y and
                state <= s0;    // start next sum of product
            end else begin //Subtract for last accumulator step
                if (count ==3) // i.e. p/2 +/- table_out * 8
                    p <= (p >>> 1) - (table_out <<< 3);
                else // Accumulation for all other steps
                    p <= (p >>> 1) + (table_out <<< 3);
                for (k=0; k<=2; k= k+1) begin // Shift bits
                    x0[k] <= x0[k+1];
                    x1[k] <= x1[k+1];
                    x2[k] <= x2[k+1];
                end
                count = count + 1;
                state <= s1;
            end
        end
    endcase
end

case3s LC_Table0
( .table_in(table_in), .table_out(table_out));

assign lut = table_out; // Provide test signal

```



```
endmodule
```

```
//*****
// IEEE STD 1364-2001 Verilog file: case3s.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module case3s
    (input  [2:0] table_in, // Three bit
     output reg [3:0] table_out); // Range -2 to 4 -> 4 bits
    // -----
    // This is the DA CASE table for
    // the 3 coefficients: -2, 3, 1

    always @(table_in)
    begin
        case (table_in)
            0 :    table_out = 0;
            1 :    table_out = -2;
            2 :    table_out = 3;
            3 :    table_out = 1;
            4 :    table_out = 1;
            5 :    table_out = -1;
            6 :    table_out = 4;
            7 :    table_out = 2;
            default : ;
        endcase
    end

endmodule
```

```
//*****
// IEEE STD 1364-2001 Verilog file: dapara.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
`include "case3s.v" // User defined component

module dapara //----> Interface
    (input      clk, // System clock
     input reset, // Asynchron reset
     input signed [3:0] x_in, // System input
     output reg signed [6:0] y); // System output
    // -----
```

```

reg signed [2:0] x [0:3];
wire signed [3:0] h [0:3];
reg signed [4:0] s0, s1;
reg signed [3:0] t0, t1, t2, t3;

always @(posedge clk or posedge reset)
begin : DA //----> DA in behavioral style
    integer k,l;
    if (reset) begin // Asynchronous clear
        for (k=0; k<=3; k=k+1) x[k] <= 0;
        y <= 0;
        t0 <= 0; t1 <= 0; t2 <= 0; t3 <= 0; s0 <= 0; s1 <= 0;
    end else begin
        for (l=0; l<=3; l=l+1) begin // For all 4 vectors
            for (k=0; k<=1; k=k+1) begin // shift all bits
                x[l][k] <= x[l][k+1];
            end
        end
        for (k=0; k<=3; k=k+1) begin // Load x_in in the
            x[k][2] <= x_in[k]; // MSBs of the registers
        end
        y <= h[0] + (h[1] << 1) + (h[2] << 2) - (h[3] << 3);
        // Sign extensions, pipeline register, and adder tree:
        // t0 <= h[0]; t1 <= h[1]; t2 <= h[2]; t3 <= h[3];
        // s0 <= t0 + (t1 << 1);
        // s1 <= t2 - (t3 << 1);
        // y <= s0 + (s1 << 2);
    end
end

genvar i; // Need to declare loop variable in Verilog 2001
generate // One table for each bit in x_in
    for (i=0; i<=3; i=i+1) begin:LC_Tables
        case3s LC_Table0 ( .table_in(x[i]), .table_out(h[i]));
    end
endgenerate

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module iir #(parameter W = 14) // Bit width - 1

```

```

(input  clk,                // System clock
 input  reset,              // Asynchronous reset
 input  signed [W:0] x_in,  // System input
 output signed [W:0] y_out); // System output
// -----
reg signed [W:0] x, y;

// Use FFs for input and recursive part
always @(posedge clk or posedge reset)
  if (reset) begin          // Note: there is a signed
    x <= 0; y <= 0;          // integer in Verilog 2001
  end else begin
    x <= x_in;
    y <= x + (y >>> 1) + (y >>> 2); // >>> uses less LEs
    // y <= x + y / 'sd2 + y / 'sd4; // same as /2 and /4
    //y <= x + y / 2 + y / 4; // div with / uses more LEs
    //y <= x + {y[W],y[W:1]}+ {y[W],y[W],y[W:2]};
  end

assign y_out = y;           // Connect y to output pins

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir_pipe.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module iir_pipe              //----> Interface
  #(parameter W = 14)        // Bit width - 1
  (input clk,                 // System clock
   input reset,               // Asynchronous reset
   input signed [W:0] x_in,   // System input
   output signed [W:0] y_out); // System output
// -----
  reg signed [W:0] x, x3, sx;
  reg signed [W:0] y, y9;

  always @(posedge clk or posedge reset) // Infer FFs for
  begin                                  // input, output and pipeline stages;
    if (reset) begin // Asynchronous clear
      x <= 0; x3 <= 0; sx <= 0; y9 <= 0; y <= 0;
    end else begin
      x <= x_in;          // use non-blocking FF assignments

```

```

    x3  <= (x >>> 1) + (x >>> 2);
           // i.e.  $x / 2 + x / 4 = x*3/4$ 
    sx  <= x + x3; // Sum of x element i.e. output FIR part
    y9  <= (y >>> 1) + (y >>> 4);
           // i.e.  $y / 2 + y / 16 = y*9/16$ 
    y    <= sx + y9; // Compute output
end
end

assign y_out = y ; // Connect register y to output pins

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir_par.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module iir_par //----> Interface
#(parameter W = 14) // bit width - 1
(input  clk, // System clock
 input  reset, // Asynchronous reset
 input  signed [W:0] x_in, // System input
 output signed [W:0] x_e, x_o, // Even/odd input x_in
 output signed [W:0] y_e, y_o, // Even/odd output y_out
 output  clk2, // Clock divided by 2
 output signed [W:0] y_out); // System output
// -----
    reg signed [W:0] x_even, xd_even, x_odd, xd_odd, x_wait;
    reg signed [W:0] y_even, y_odd, y_wait, y;
    reg signed [W:0] sum_x_even, sum_x_odd;
    reg      clk_div2;
    reg [0:0] state;

    always @(posedge clk or posedge reset) // Clock divider
    begin : clk_divider // by 2 for input clk
        if (reset) clk_div2 <= 0;
        else      clk_div2 <= ! clk_div2;
    end

    always @(posedge clk or posedge reset) // Split x into
    begin : Multiplex // even and odd samples;
        parameter even=0, odd=1; // recombine y at clk rate

```

```

if (reset) begin                                // Asynchronous reset
    state <= even; x_even <= 0; x_odd <= 0;
    y <= 0; x_wait <= 0; y_wait <= 0;
end else
    case (state)
        even : begin
            x_even <= x_in;
            x_odd <= x_wait;
            y <= y_wait;
            state <= odd;
        end
        odd : begin
            x_wait <= x_in;
            y <= y_odd;
            y_wait <= y_even;
            state <= even;
        end
    endcase
end

assign y_out = y;
assign clk2 = clk_div2;
assign x_e = x_even; // Monitor some extra test signals
assign x_o = x_odd;
assign y_e = y_even;
assign y_o = y_odd;

always @(negedge clk_div2 or posedge reset)
begin: Arithmetic
    if (reset) begin
        xd_even <= 0; sum_x_even <= 0; y_even <= 0;
        xd_odd <= 0; sum_x_odd <= 0; y_odd <= 0;
    end else begin
        xd_even <= x_even;
        sum_x_even <= x_odd + (xd_even >>> 1) + (xd_even >>> 2);
            // i.e. x_odd + x_even/2 + x_even / 4
        y_even <= sum_x_even + (y_even >>> 1) + (y_even >>> 4);
            // i.e. sum_x_even + y_even / 2 + y_even / 16
        xd_odd <= x_odd;
        sum_x_odd <= xd_even + (xd_odd >>> 1) + (xd_odd >>> 4);
            // i.e. x_even + xd_odd / 2 + xd_odd / 4
        y_odd <= sum_x_odd + (y_odd >>> 1) + (y_odd >>> 4);
            // i.e. sum_x_odd + y_odd / 2 + y_odd / 16
    end
end

```

```

end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir5sfix.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// 5th order IIR in direct form implementation
module iir5sfix
    (input clk,                // System clock
     input reset,              // Asynchron reset
     input switch,             // Feedback switch
     input signed [63:0] x_in, // System input
     output signed [39:0] t_out, // Feedback
     output signed [39:0] y_out); // System output
// -----
    wire signed[63:0] a2, a3, a4, a5, a6; // Feedback A
    wire signed[63:0] b1, b2, b3, b4, b5, b6; // Feedforward B
    wire signed [63:0] h;
    reg signed [63:0] s1, s2, s3, s4;
    reg signed [63:0] y, t, r2, r3, r4;
    reg signed [127:0] a6h, a5h, a4h, a3h, a2h;
    reg signed [127:0] b6h, b5h, b4h, b3h, b2h, b1h;

// Feedback A scaled by 2^30
    assign a2 = 64'h000000013DF707FA; // (-)4.9682025852
    assign a3 = 64'h0000000277FBF6D7; // 9.8747536754
    assign a4 = 64'h00000002742912B6; // (-)9.8150069021
    assign a5 = 64'h00000001383A6441; // 4.8785639415
    assign a6 = 64'h000000003E164061; // (-)0.9701081227
// Feedforward B scaled by 2^30
    assign b1 = 64'h000000000004F948; // 0.0003035737
    assign b2 = 64'h00000000000EE2A2; // (-)0.0009085259
    assign b3 = 64'h000000000009E95E; // 0.0006049556
    assign b4 = 64'h000000000009E95E; // 0.0006049556
    assign b5 = 64'h00000000000EE2A2; // (-)0.0009085259
    assign b6 = 64'h000000000004F948; // 0.0003035737
    assign h = (switch) ? x_in-t // Switch is closed
                : x_in; // Switch is open

    always @(posedge clk or posedge reset)
    begin : P1 // First equations without infering registers
        if (reset) begin // Asynchronous clear

```

```

    t <= 0; y <= 0; r2 <= 0; r3 <= 0; r4 <= 0; s1 <= 0;
    s2 <= 0; s3 <= 0; s4 <= 0; a6h <= 0; b6h <= 0;
end else begin // IIR filter in direct form
    a6h <= a6 * h; // h*a[6] use register
    a5h = a5 * h; // h*a[5]
    r4 <= (a5h >>> 30) - (a6h >>> 30); // h*a[5]+r5
    a4h = a4 * h; // h*a[4]
    r3 <= r4 - (a4h >>> 30); // h*a[4]+r4
    a3h = a3 * h; // h*a[3]
    r2 <= r3 + (a3h >>> 30); // h*a[3]+r3
    a2h = a2 * h; // h*a[2]+r2
    t <= r2 - (a2h >>> 30); // h*a[2]+r2
    b6h <= b6 * h; // h*b[6] use register
    b5h = b5 * h; // h*b[5]+s5 no register
    s4 <= (b6h >>> 30) - (b5h >>> 30); // h*b[5]+s5
    b4h = b4 * h; // h*b[4]+s4
    s3 <= s4 + (b4h >>> 30); // h*b[4]+s4
    b3h = b3 * h; // h*b[3]
    s2 <= s3 + (b3h >>> 30); // h*b[3]+s3
    b2h = b2 * h; // h*b[2]
    s1 <= s2 - (b2h >>> 30); // h*b[2]+s2
    b1h = b1 * h; // h*b[1]
    y <= s1 + (b1h >>> 30); // h*b[1]+s1
end
end

// Redefine bits as 40 bit SLV
// Change 30 to 16 bit fraction, i.e. cut 14 LSBs
assign y_out = y[53:14];
assign t_out = t[53:14];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir5para.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Description: 5th order IIR parallel form implementation
// Coefficients:
// D = 0.00030357
// B1 = 0.0031 -0.0032 0
// A1 = 1.0000 -1.9948 0.9959
// B2 = -0.0146 0.0146 0
// A2 = 1.0000 -1.9847 0.9852
// B3 = 0.0122

```

```

// A3 = 0.9887
// -----
module iir5para // I/O data in 16.16 format
  (input clk, // System clock
   input reset, // Asynchron reset
   input signed [31:0] x_in, // System input
   output signed [31:0] y_Dout, // 0 order
   output signed [31:0] y_1out, // 1. order
   output signed [31:0] y_21out, // 2. order 1
   output signed [31:0] y_22out, // 2. order 2
   output signed [31:0] y_out); // System output
// -----
// Internal type has 2 bits integer and 18 bits fraction:
reg signed [19:0] s11, r13, r12; // 1. BiQuad regs.
reg signed [19:0] s21, r23, r22; // 2. BiQuad regs.
reg signed [19:0] r32, r41, r42, r43, y;
reg signed [19:0] x;

// Products have double bit width
reg signed [39:0] b12x, b11x, a13r12, a12r12; // 1. BiQuad
reg signed [39:0] b22x, b21x, a23r22, a22r22; // 2. BiQuad
reg signed [39:0] b31x, a32r32, Dx;

// All coefficients use 5*4=20 bits and are scaled by 2^18
wire signed [19:0] a12, a13, b11, b12; // First BiQuad
wire signed [19:0] a22, a23, b21, b22; // Second BiQuad
wire signed [19:0] a32, b31, D; // First order and direct
// First BiQuad coefficients
assign a12 = 20'h7FAB9; // (-)1.99484680
assign a13 = 20'h3FBD0; // 0.99591112
assign b11 = 20'h00340; // 0.00307256
assign b12 = 20'h00356; // (-)0.00316061
// Second BiQuad coefficients
assign a22 = 20'h7F04F; // (-)1.98467605
assign a23 = 20'h3F0E4; // 0.98524428
assign b21 = 20'h00F39; // (-)0.01464265
assign b22 = 20'h00F38; // 0.01464684
// First order system with R(5) and P(5)
assign a32 = 20'h3F468; // 0.98867974
assign b31 = 20'h00C76; // 0.012170
// Direct system
assign D = 20'h0004F; // 0.000304

always @(posedge clk or posedge reset)

```



```

begin : P1
    if (reset) begin                // Asynchronous clear
        b12x <= 0; s11 <= 0; r13 <= 0; r12 <= 0;
        b22x <= 0; s21 <= 0; r23 <= 0; r22 <= 0;
        b31x <= 0; r32 <= 0; r41 <= 0;
        r42 <= 0; r43 <= 0; y <= 0; x <= 0;
    end else begin                  // SOS modified BiQuad form
        // Redefine bits as FIX 16.16 number to
        x <= {x_in[17:0], 2'b00};
        // Internal precision 2.19 format, i.e. 21 bits
        // 1. BiQuad is 2. order
        b12x <= b12 * x;
        b11x = b11 * x;
        s11 <= (b11x >>> 18) - (b12x >>> 18); // was +
        a13r12 = a13 * r12;
        r13 <= s11 - (a13r12 >>> 18);
        a12r12 = a12 * r12;
        r12 <= r13 + (a12r12 >>> 18); // was -
        // 2. BiQuad is 2. order
        b22x <= b22 * x;
        b21x = b21 * x;
        s21 <= (b22x >>> 18) - (b21x >>> 18); // was +
        a23r22 = a23 * r22;
        r23 <= s21 - (a23r22 >>> 18);
        a22r22 = a22 * r22;
        r22 <= r23 + (a22r22 >>> 18); // was -
        // 3. Section is 1. order
        b31x <= b31 * x;
        a32r32 = a32 * r32;
        r32 <= (b31x >>> 18) + (a32r32 >>> 18);
        // 4. Section is assign
        Dx = D * x;
        r41 <= Dx >>> 18;
        // Output adder tree
        r42 <= r41;
        r43 <= r42 + r32;
        y <= r12 + r22 + r43;
    end
end

// Change 19 to 16 bit fraction, i.e. cut 2 LSBs
// Redefine bits as 32 bit SLV
assign y_out    = {{14{y[19]}},y[19:2]};
assign y_Dout   = {{14{r42[19]}},r42[19:2]};

```

```

    assign y_1out  = {{14{r32[19]}}},r32[19:2]};
    assign y_21out = {{14{r22[19]}}},r22[19:2]};
    assign y_22out = {{14{r12[19]}}},r12[19:2]};

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir5lwdf.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Description: 5th order Lattice Wave Digital Filter
// Coefficients gamma:
// 0.988727 -0.000528 -1.995400 -0.000282 -1.985024
// -----
module iir5lwdf                      //----> Interface
    (input clk,                      // System clock
     input reset,                    // Asynchronous reset
     input signed [31:0] x_in,      // System input
     output signed [31:0] y_ap1out, // AP1 out
     output signed [31:0] y_ap2out, // AP2 out
     output signed [31:0] y_ap3out, // AP3 out
     output signed [31:0] y_out);    // System output
// -----
// Internal signals have 7.15 format
    reg signed [21:0] c1, c2, c3, l2, l3;
    reg signed [21:0] a4, a5, a6, a8, a9, a10;
    reg signed [21:0] x, ap1, ap2, ap3, ap3r, y;

// Products have double bit width
    reg signed [41:0] p1, a4g2, a4g3, a8g4, a8g5;

//Coefficients gamma use 5*4=20 bits and are scaled by 2^15
    wire signed [19:0] g1, g2, g3, g4, g5;
    assign g1 = 20'h07E8F; // 0.988739
    assign g2 = 20'h00011; // (-)0.000519
    assign g3 = 20'h0FF69; // (-)1.995392
    assign g4 = 20'h00009; // (-)0.000275
    assign g5 = 20'h0FE15; // (-)1.985016

    always @(posedge clk or posedge reset)
    begin : P1
        if (reset) begin // Asynchronous clear
            c1 <= 0; ap1 <= 0; c2 <= 0; l2 <= 0;
            ap2 <= 0; c3 <= 0; l3 <= 0; ap3 <= 0;
            ap3r <= 0; y <= 0; x <= 0;

```

```

        end else begin // AP LWDF form
            // Redefine 16.16 input bits as internal precision
            // in 7.15 format, i.e. 20 bits
            x <= x_in[22:1];
// 1. AP section is 1. order
            p1 = g1 * (c1 - x);
            c1 <= x + (p1 >>> 15);
            ap1 <= c1 + (p1 >>> 15);
// 2. AP section is 2. order
            a4 = ap1 - l2 + c2 ;
            a4g2 = a4 * g2;
            a5 = c2 - (a4g2 >>> 15); // was +
            a4g3 = a4 * g3;
            a6 = -(a4g3 >>> 15) - l2; // was +
            c2 <= a5;
            l2 <= a6;
            ap2 <= -a5 - a6 - a4;
// 3. AP section is 2. order
            a8 = x - l3 + c3;
            a8g4 = a8 * g4;
            a9 = c3 - (a8g4 >>> 15); // was +
            a8g5 = a8 * g5;
            a10 = -(a8g5 >>> 15) - l3; // was +
            c3 <= a9;
            l3 <= a10;
            ap3 <= -a9 - a10 - a8;
            ap3r <= ap3; // Extra register due to AP1
// Output adder
            y <= ap3r + ap2;
        end
    end

// Change 15 to 16 bit fraction, i.e. add 1 LSBs
// Redefine bits as 32 bit SLV 1+22+9=32
    assign y_out = {{9{y[21]}},y,1'b0};
    assign y_ap1out = {{9{ap1[21]}},ap1,1'b0};
    assign y_ap2out = {{9{ap2[21]}},ap2,1'b0};
    assign y_ap3out = {{9{ap3r[21]}},ap3r,1'b0};

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: db4poly.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org

```

```

//*****
module db4poly                                //----> Interface
(input clk,                                  // System clock
 input reset,                               // Asynchron reset
 input signed [7:0] x_in,                   // System input
 output clk2,                               // Clock divided by 2
 output signed [16:0] x_e, x_o, // Even/odd x_in
 output signed [16:0] g0, g1, // Poly filter 0/1
 output signed [8:0] y_out); // System output
// -----
    reg signed [7:0] x_odd, x_even, x_wait;
    reg clk_div2;

// Register for multiplier, coefficients, and taps
    reg signed [16:0] m0, m1, m2, m3, r0, r1, r2, r3;
    reg signed [16:0] x33, x99, x107;
    reg signed [16:0] y;

    always @(posedge clk or posedge reset) // Split into even
begin : Multiplex // and odd samples at clk rate
    parameter even=0, odd=1;
    reg [0:0] state;

    if (reset) begin // Asynchronous reset
        state <= even;
        clk_div2 = 0; x_even <= 0; x_odd <= 0; x_wait <= 0;
    end else
        case (state)
            even : begin
                x_even <= x_in;
                x_odd <= x_wait;
                clk_div2 = 1;
                state <= odd;
            end
            odd : begin
                x_wait <= x_in;
                clk_div2 = 0;
                state <= even;
            end
        endcase
    end

    always @(x_odd, x_even)
begin : RAG

```

```

// Compute auxiliary multiplications of the filter
x33 = (x_odd <<< 5) + x_odd;
x99 = (x33 <<< 1) + x33;
x107 = x99 + (x_odd << 3);
// Compute all coefficients for the transposed filter
m0 = (x_even <<< 7) - (x_even <<< 2); // m0 = 124
m1 = x107 <<< 1;                      // m1 = 214
m2 = (x_even <<< 6) - (x_even <<< 3)
      + x_even; // m2 = 57
m3 = x33;       // m3 = -33
end

always @(posedge reset or negedge clk_div2)
begin : AddPolyphase // use non-blocking assignments
  if (reset) begin // Asynchronous clear
    r0 <= 0; r1 <= 0; r2 <= 0; r3 <= 0;
    y <= 0;
  end else begin
//----- Compute filter G0
    r0 <= r2 + m0; // g0 = 128
    r2 <= m2;      // g2 = 57
//----- Compute filter G1
    r1 <= -r3 + m1; // g1 = 214
    r3 <= m3;      // g3 = -33
// Add the polyphase components
    y <= r0 + r1;
  end
end

// Provide some test signals as outputs
assign x_e = x_even;
assign x_o = x_odd;
assign clk2 = clk_div2;
assign g0 = r0;
assign g1 = r1;

assign y_out = y >>> 8; // Connect y / 256 to output

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cic3r32.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module cic3r32 //----> Interface

```

```

(input      clk,          // System clock
 input      reset,        // Asynchronous reset
 input signed [7:0] x_in, // System input
 output signed [9:0] y_out, // System output
 output reg clk2);        // Clock divider
// -----
parameter hold=0, sample=1;
reg [1:0] state;
reg [4:0] count;
reg signed [7:0] x;      // Registered input
reg signed [25:0] i0, i1, i2; // I section 0, 1, and 2
reg signed [25:0] i2d1, i2d2, c1, c0; // I + COMB 0
reg signed [25:0] c1d1, c1d2, c2; // COMB section 1
reg signed [25:0] c2d1, c2d2, c3; // COMB section 2

always @(posedge clk or posedge reset)
begin : FSM
  if (reset) begin // Asynchronous reset
    count <= 0;
    state <= hold;
    clk2 <= 0;
  end else begin
    if (count == 31) begin
      count <= 0;
      state <= sample;
      clk2 <= 1;
    end else begin
      count <= count + 1;
      state <= hold;
      clk2 <= 0;
    end
  end
end

always @(posedge clk or posedge reset)
begin : Int // 3 stage integrator sections
  if (reset) begin // Asynchronous clear
    x <= 0; i0 <= 0; i1 <= 0; i2 <= 0;
  end else begin
    x <= x_in;
    i0 <= i0 + x;
    i1 <= i1 + i0;
    i2 <= i2 + i1;
  end
end

```

```

end

always @(posedge clk or posedge reset)
begin : Comb          // 3 stage comb sections
  if (reset) begin // Asynchronous clear
    c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0;
    i2d1 <= 0; i2d2 <= 0; c1d1 <= 0; c1d2 <= 0;
    c2d1 <= 0; c2d2 <= 0;
  end else if (state == sample) begin
    c0  <= i2;
    i2d1 <= c0;
    i2d2 <= i2d1;
    c1  <= c0 - i2d2;
    c1d1 <= c1;
    c1d2 <= c1d1;
    c2  <= c1 - c1d2;
    c2d1 <= c2;
    c2d2 <= c2d1;
    c3  <= c2 - c2d2;
  end
end

assign y_out = c3[25:16];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cic3s32.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module cic3s32          //----> Interface
(input      clk,        // System clock
 input      reset,      // Asynchronous reset
 input signed [7:0] x_in, // System input
 output signed [9:0] y_out, // System output
 output reg clk2);      // Clock divider
// -----
  parameter hold=0, sample=1;
  reg [1:0] state;
  reg [4:0] count;
  reg signed [7:0] x;          // Registered input
  reg signed [25:0] i0;        // I section 0
  reg signed [20:0] i1;        // I section 1
  reg signed [15:0] i2;        // I section 2
  reg signed [13:0] i2d1, i2d2, c1, c0; // I+C0

```

```

reg signed [12:0] c1d1, c1d2, c2;          // COMB 1
reg signed [11:0] c2d1, c2d2, c3;        // COMB 2

```

```

always @(posedge clk or posedge reset)
begin : FSM
    if (reset) begin                // Asynchronous reset
        count <= 0;
        state <= hold;
        clk2 <= 0;
    end else begin
        if (count == 31) begin
            count <= 0;
            state <= sample;
            clk2 <= 1;
        end
        else begin
            count <= count + 1;
            state <= hold;
            clk2 <= 0;
        end
    end
end
end

```

```

always @(posedge clk or posedge reset)
begin : Int                // 3 stage integrator sections
    if (reset) begin // Asynchronous clear
        x <= 0; i0 <= 0; i1 <= 0; i2 <= 0;
    end else begin
        x <= x_in;
        i0 <= i0 + x;
        i1 <= i1 + i0[25:5];
        i2 <= i2 + i1[20:5];
    end
end
end

```

```

always @(posedge clk or posedge reset)

begin : Comb                // 3 stage comb sections
    if (reset) begin // Asynchronous clear
        c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0;
        i2d1 <= 0; i2d2 <= 0; c1d1 <= 0; c1d2 <= 0;
        c2d1 <= 0; c2d2 <= 0;
    end else if (state == sample) begin
        c0 <= i2[15:2];
    end
end

```



```

        i2d1 <= c0;
        i2d2 <= i2d1;
        c1   <= c0 - i2d2;
        c1d1 <= c1[13:1];
        c1d2 <= c1d1;
        c2   <= c1[13:1] - c1d2;
        c2d1 <= c2[12:1];
        c2d2 <= c2d1;
        c3   <= c2[12:1] - c2d2;
    end
end

    assign y_out = c3[11:2];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: rc_sinc.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module rc_sinc #(parameter OL = 2, //Output buffer length-1
                    IL = 3, //Input buffer length -1
                    L = 10) // Filter length -1
    (input clk,           // System clock
     input reset,         // Asynchronous reset
     input signed [7:0] x_in, // System input
     output [3:0] count_o, // Counter FSM
     output ena_in_o,      // Sample input enable
     output ena_out_o,     // Shift output enable
     output ena_io_o,      // Enable transfer2output
     output signed [8:0] f0_o, // First Sinc filter output
     output signed [8:0] f1_o, // Second Sinc filter output
     output signed [8:0] f2_o, // Third Sinc filter output
     output signed [8:0] y_out); // System output
// -----
    reg [3:0] count; // Cycle R_1*R_2
    reg ena_in, ena_out, ena_io; // FSM enables
    reg signed [7:0] x [0:10]; // TAP registers for 3 filters
    reg signed [7:0] ibuf [0:3]; // TAP in registers
    reg signed [7:0] obuf [0:2]; // TAP out registers
    reg signed [8:0] f0, f1, f2; // Filter outputs

    // Constant arrays for multiplier and taps:
    wire signed [8:0] c0 [0:10];
    wire signed [8:0] c2 [0:10];

```

```

// filter coefficients for filter c0
assign c0[0] = -19; assign c0[1] = 26; assign c0[2]=-42;
assign c0[3] = 106; assign c0[4] = 212; assign c0[5]=-53;
assign c0[6] = 29; assign c0[7] = -21; assign c0[8]=16;
assign c0[9] = -13; assign c0[10] = 11;

// filter coefficients for filter c2
assign c2[0] = 11; assign c2[1] = -13; assign c2[2] = 16;
assign c2[3] = -21; assign c2[4] = 29; assign c2[5] = -53;
assign c2[6] = 212; assign c2[7] = 106; assign c2[8] = -42;
assign c2[9] = 26; assign c2[10] = -19;

always @(posedge reset or posedge clk)
begin : FSM // Control the system and sample at clk rate
    if (reset) // Asynchronous reset
        count <= 0;
    else
        if (count == 11) count <= 0;
        else
            count <= count + 1;
end

always @(posedge clk)
begin // set the enable signal for the TAP lines
    case (count)
        2, 5, 8, 11 : ena_in <= 1;
        default : ena_in <= 0;
    endcase

    case (count)
        4, 8 : ena_out <= 1;
        default : ena_out <= 0;
    endcase

    if (count == 0) ena_io <= 1;
    else
        ena_io <= 0;
end

always @(posedge clk or posedge reset) // Input delay line
begin : INPUTMUX
    integer I; // loop variable
    if (reset) // Asynchronous clear
        for (I=0; I<=IL; I=I+1) ibuf[I] <= 0;
    else if (ena_in) begin

```

```

    for (I=IL; I>=1; I=I-1)
        ibuf[I] <= ibuf[I-1];          // shift one
    ibuf[0] <= x_in;                   // Input in register 0
end
end

```

```

always @(posedge clk or posedge reset)//Output delay line
begin : OUPUTMUX
    integer I;    // loop variable
    if (reset)    // Asynchronous clear
        for (I=0; I<=OL; I=I+1) obuf[I] <= 0;
    else begin
        if (ena_io) begin // store 3 samples in output buffer
            obuf[0] <= f0;
            obuf[1] <= f1;
            obuf[2] <= f2;
        end else if (ena_out) begin
            for (I=OL; I>=1; I=I-1)
                obuf[I] <= obuf[I-1];    // shift one
        end
    end
end
end

```

```

always @(posedge clk or posedge reset)
begin : TAP    // get 4 samples at one time
    integer I;    // loop variable
    if (reset)    // Asynchronous clear
        for (I=0; I<=10; I=I+1) x[I] <= 0;
    else if (ena_io) begin    // One tapped delay line
        for (I=0; I<=3; I=I+1)
            x[I] <= ibuf[I];    // take over input buffer

        for (I=4; I<=10; I=I+1) // 0->4; 4->8 etc.
            x[I] <= x[I-4];    // shift 4 taps
    end
end

```

```

always @(posedge clk or posedge reset)
begin : SOP0    // Compute sum-of-products for f0
    reg signed [16:0] sum; // temp sum
    reg signed [16:0] p [0:10]; // temp products
    integer I;

    for (I=0; I<=L; I=I+1) // Infer L+1 multiplier

```

```

    p[I] = c0[I] * x[I];

    sum = p[0];
    for (I=1; I<=L; I=I+1)          // Compute the direct
        sum = sum + p[I];           // filter adds
    if (reset) f0 <= 0;              // Asynchronous clear
    else f0 <= sum >>> 8;
end

always @(posedge clk or posedge reset)
begin : SOP1                        // Compute sum-of-products for f1
    if (reset) f1 <= 0;              // Asynchronous clear
    else f1 <= x[5]; // No scaling, i.e., unit impulse
end

always @(posedge clk) // Compute sum-of-products for f2
begin : SOP2
    reg signed[16:0] sum; // temp sum
    reg signed [16:0] p [0:10]; // temp products
    integer I;

    for (I=0; I<=L; I=I+1) // Infer L+1 multiplier
        p[I] = c2[I] * x[I];

    sum = p[0];
    for (I=1; I<=L; I=I+1)          // Compute the direct
        sum = sum + p[I];           // filter adds

    if (reset) f2 <= 0;              // Asynchronous clear
    else f2 <= sum >>> 8;
end

// Provide some test signals as outputs
assign f0_o = f0;
assign f1_o = f1;
assign f2_o = f2;
assign count_o = count;
assign ena_in_o = ena_in;
assign ena_out_o = ena_out;
assign ena_io_o = ena_io;

assign y_out = obuf[0L]; // Connect to output
endmodule

```

```

//*****
// IEEE STD 1364-2001 Verilog file: farrow.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module farrow #(parameter IL = 3) // Input buffer length -1
    (input clk,                // System clock
     input reset,              // Asynchronous reset
     input signed [7:0] x_in,  // System input
     output [3:0] count_o,     // Counter FSM
     output ena_in_o,          // Sample input enable
     output ena_out_o,         // Shift output enable
     output signed [8:0] c0_o,c1_o,c2_o,c3_o, // Phase delays
     output [8:0] d_out,       // Delay used
     output reg signed [8:0] y_out); // System output
// -----
    reg [3:0] count; // Cycle R_1*R_2
    wire [6:0] delta; // Increment d
    reg ena_in, ena_out; // FSM enables
    reg signed [7:0] x [0:3];
    reg signed [7:0] ibuf [0:3]; // TAP registers
    reg [8:0] d; // Fractional Delay scaled to 8 bits
    // Lagrange matrix outputs:
    reg signed [8:0] c0, c1, c2, c3;

    assign delta = 85;

    always @(posedge reset or posedge clk) // Control the
begin : FSM // system and sample at clk rate
    reg [8:0] dnew;
    if (reset) begin // Asynchronous reset
        count <= 0;
        d <= delta;
    end else begin
        if (count == 11)
            count <= 0;
        else
            count <= count + 1;
        if (ena_out) begin // Compute phase delay
            dnew = d + delta;
            if (dnew >= 255)
                d <= 0;
            else
                d <= dnew;
        end
    end
end

```

```

        end
    end
end

always @(posedge clk or posedge reset)
begin
    // Set the enable signals for the TAP lines
    case (count)
        2, 5, 8, 11 : ena_in <= 1;
        default      : ena_in <= 0;
    endcase

    case (count)
        3, 7, 11 : ena_out <= 1;
        default   : ena_out <= 0;
    endcase
end

always @(posedge clk or posedge reset)
begin : TAP
    //----> One tapped delay line
    integer I;    // loop variable
    if (reset)    // Asynchronous clear
        for (I=0; I<=IL; I=I+1) ibuf[I] <= 0;
    else if (ena_in) begin
        for (I=1; I<=IL; I=I+1)
            ibuf[I-1] <= ibuf[I];    // Shift one

        ibuf[IL] <= x_in;    // Input in register IL
    end
end

always @(posedge clk or posedge reset)
begin : GET
    // Get 4 samples at one time
    integer I;    // loop variable
    if (reset)    // Asynchronous clear
        for (I=0; I<=IL; I=I+1) x[I] <= 0;
    else if (ena_out) begin
        for (I=0; I<=IL; I=I+1)
            x[I] <= ibuf[I];    // take over input buffer
    end
end

// Compute sum-of-products:
always @(posedge clk or posedge reset)
begin : SOP

```

```

    reg signed [8:0] y1, y2, y3; // temp's

// Matrix multiplier iV=inv(Vandermonde) c=iV*x(n-1:n+2)'
//      x(0)   x(1)       x(2)   x(3)
// iV=    0    1.0000        0      0
//   -0.3333  -0.5000    1.0000  -0.1667
//    0.5000  -1.0000    0.5000      0
//   -0.1667   0.5000   -0.5000   0.1667
    if (reset) begin           // Asynchronous clear
        y_out <= 0;
        c0 <= 0; c1 <= 0; c2<= 0; c3 <= 0;
    end else if (ena_out) begin
        c0 <= x[1];
        c1 <= (-85 * x[0] >>> 8) - (x[1]/2) + x[2] -
                                   (43 * x[3] >>> 8);
        c2 <= ((x[0] + x[2]) >>> 1) - x[1] ;
        c3 <= ((x[1] - x[2]) >>> 1) +
                                   (43 * (x[3] - x[0]) >>> 8);

// Farrow structure = Lagrange with Horner schema
// for u=0:3, y=y+f(u)*d^u; end;
    y1 = c2 + ((c3 * d) >>> 8); // d is scale by 256
    y2 = ((y1 * d) >>> 8) + c1;
    y3 = ((y2 * d) >>> 8) + c0;

    y_out <= y3; // Connect to output + store in register
end
end

assign c0_o = c0; // Provide test signals as outputs
assign c1_o = c1;
assign c2_o = c2;
assign c3_o = c3;
assign count_o = count;
assign ena_in_o = ena_in;
assign ena_out_o = ena_out;
assign d_out = d;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cmoms.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****

```

```

module cmoms #(parameter IL = 3) // Input buffer length -1
(input clk,                // System clock
 input reset,              // Asynchron reset
 output [3:0] count_o,    // Counter FSM
 output ena_in_o,         // Sample input enable
 output ena_out_o,        // Shift output enable
 input signed [7:0] x_in, // System input
 output signed [8:0] xiir_o, // IIR filter output
 output signed [8:0] c0_o,c1_o,c2_o,c3_o, // C-MOMS matrix
 output signed [8:0] y_out); // System output
// -----
reg [3:0] count; // Cycle R_1*R_2
reg [1:0] t;
reg ena_in, ena_out; // FSM enables
reg signed [7:0] x [0:3];
reg signed [7:0] ibuf [0:3]; // TAP registers
reg signed [8:0] xiir; // iir filter output

reg signed [16:0] y, y0, y1, y2, y3, h0, h1; // temp's

// Spline matrix output:
reg signed [8:0] c0, c1, c2, c3;

// Precomputed value for d**k :
wire signed [8:0] d1 [0:2];
wire signed [8:0] d2 [0:2];
wire signed [8:0] d3 [0:2];

assign d1[0] = 0; assign d1[1] = 85; assign d1[2] = 171;
assign d2[0] = 0; assign d2[1] = 28; assign d2[2] = 114;
assign d3[0] = 0; assign d3[1] = 9; assign d3[2] = 76;

always @(posedge reset or posedge clk) // Control the
begin : FSM // system sample at clk rate
  if (reset) begin // Asynchronous reset
    count <= 0;
    t <= 1;
  end else begin
    if (count == 11)
      count <= 0;
    else
      count <= count + 1;
    if (ena_out)

```



```

        if (t>=2)      // Compute phase delay
            t <= 0;
        else
            t <= t + 1;
    end
end
assign t_out = t;

always @(posedge clk) // set the enable signal
begin                // for the TAP lines
    case (count)
        2, 5, 8, 11 : ena_in <= 1;
        default      : ena_in <= 0;
    endcase

    case (count)
        3, 7, 11     : ena_out <= 1;
        default      : ena_out <= 0;
    endcase
end

// Coeffs: H(z)=1.5/(1+0.5z^-1)
always @(posedge clk or posedge reset)
begin : IIR // Compute iir coefficients first
    reg signed [8:0] x1;    // x * 1
    if (reset) begin // Asynchronous clear
        xiir <= 0; x1 <= 0;
    end else
    if (ena_in) begin
        xiir <= (3 * x1 >>> 1) - (xiir >>> 1);
        x1 = x_in;
    end
end

always @(posedge clk or posedge reset)
begin : TAP //----> One tapped delay line
    integer I;    // Loop variable
    if (reset) begin // Asynchronous clear
        for (I=0; I<=IL; I=I+1) ibuf[I] <= 0;
    end else
    if (ena_in) begin
        for (I=1; I<=IL; I=I+1)
            ibuf[I-1] <= ibuf[I]; // Shift one
    end
end

```

```

        ibuf[IL] <= xiir;          // Input in register IL
    end
end

always @(posedge clk or posedge reset)
begin : GET                      // Get 4 samples at one time
    integer I;                  // Loop variable
    if (reset) begin // Asynchronous clear
        for (I=0; I<=IL; I=I+1) x[I] <= 0;
    end else
        if (ena_out) begin
            for (I=0; I<=IL; I=I+1)
                x[I] <= ibuf[I]; // Take over input buffer
        end
    end

    // Compute sum-of-products:
    always @(posedge clk or posedge reset)
    begin : SOP
        // Matrix multiplier C-MOMS matrix:
        //      x(0)      x(1)      x(2)      x(3)
        //      0.3333    0.6667    0         0
        //      -0.8333   0.6667    0.1667    0
        //      0.6667   -1.5       1.0       -0.1667
        //      -0.1667   0.5       -0.5       0.1667
        if (reset) begin // Asynchronous clear
            c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0;
            y0 <= 0; y1 <= 0; y2 <= 0; y3 <= 0;
            h0 <= 0; h1 <= 0; y <= 0;
        end else if (ena_out) begin
            c0 <= (85 * x[0] + 171 * x[1]) >>> 8;
            c1 <= (171 * x[1] - 213 * x[0] + 43 * x[2]) >>> 8;
            c2 <= (171 * x[0] - (43 * x[3]) >>> 8)
                    - (3 * x[1] >>> 1) + x[2];
            c3 <= (43 * (x[3] - x[0]) >>> 8)
                    + ((x[1] - x[2]) >>> 1);

            // No Farrow structure, parallel LUT for delays
            // for u=0:3, y=y+f(u)*d^u; end;
            y0 <= c0 * 256; // Use pipelined adder tree
            y1 <= c1 * d1[t];
            y2 <= c2 * d2[t];
            y3 <= c3 * d3[t];
            h0 <= y0 + y1;

```

```

        h1 <= y2 + y3;
        y  <= h0 + h1;
    end
end

assign y_out = y >>> 8; // Connect to output
assign c0_o = c0; // Provide some test signals as outputs
assign c1_o = c1;
assign c2_o = c2;
assign c3_o = c3;
assign count_o = count;
assign ena_in_o = ena_in;
assign ena_out_o = ena_out;
assign xiir_o = xiir;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: db4latti.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module db4latti
    (input clk,                // System clock
     input reset,              // Asynchron reset
     output clk2,              // Clock divider
     input signed [7:0] x_in,  // System input
     output signed [16:0] x_e, x_o, // Even/odd x input
     output reg signed [8:0] g, h); // g/h filter output
// -----
    reg signed [7:0] x_wait;
    reg signed [16:0] sx_up, sx_low;
    reg clk_div2;
    wire signed [16:0] sxa0_up, sxa0_low;
    wire signed [16:0] up0, up1, low1;
    reg signed [16:0] low0;

    always @(posedge clk or posedge reset) // Split into even
    begin : Multiplex // and odd samples at clk rate
        parameter even=0, odd=1;
        reg [0:0] state;

        if (reset) begin // Asynchronous reset
            state <= even;

```

```

    sx_up <= 0; sx_low <= 0;
    clk_div2 <= 0; x_wait <= 0;
end else
    case (state)
        even : begin
            // Multiply with 256*s=124
            sx_up    <= (x_in <<< 7) - (x_in <<< 2);
            sx_low   <= (x_wait <<< 7) - (x_wait <<< 2);
            clk_div2 <= 1;
            state <= odd;
        end
        odd : begin
            x_wait <= x_in;
            clk_div2 <= 0;
            state <= even;
        end
    endcase
end

//***** Multiply a[0] = 1.7321
// Compute: (2*sx_up - sx_up /4)-(sx_up /64 + sx_up /256)
assign sxa0_up = ((sx_up <<< 1) - (sx_up >>> 2))
               - ((sx_up >>> 6) + (sx_up >>> 8));
// Compute: (2*sx_low - sx_low/4)-(sx_low/64 + sx_low/256)
assign sxa0_low = ((sx_low <<< 1) - (sx_low >>> 2))
                 - ((sx_low >>> 6) + (sx_low >>> 8));

//***** First stage -- FF in lower tree
assign up0 = sxa0_low + sx_up;
always @(posedge clk or posedge reset)
begin: LowerTreeFF
    if (reset) begin                // Asynchronous clear
        low0 <= 0;
    end else if (clk_div2)
        low0 <= sx_low - sxa0_up;
end

//***** Second stage: a[1]=-0.2679
// Compute: (up0 - low0/4) - (low0/64 + low0/256);
assign up1 = (up0 - (low0 >>> 2))
            - ((low0 >>> 6) + (low0 >>> 8));
// Compute: (low0 + up0/4) + (up0/64 + up0/256)
assign low1 = (low0 + (up0 >>> 2))
              + ((up0 >>> 6) + (up0 >>> 8));

```

```

assign x_e = sx_up;          // Provide some extra
assign x_o = sx_low;         // test signals
assign clk2 = clk_div2;

always @(posedge clk or posedge reset)
begin: OutputScale
    if (reset) begin          // Asynchronous clear
        g <= 0; h <= 0;
    end else if (clk_div2) begin
        g <= up1 >>> 8;      // i.e. up1 / 256
        h <= low1 >>> 8;     // i.e. low1 / 256;
    end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: dwtdden.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module dwtdden
    #(parameter D1L = 28, //D1 buffer length
        D2L = 10) // D2 buffer length
    (input clk,              // System clock
     input reset,            // Asynchron reset
     input signed [15:0] x_in, // System input
     input signed [15:0] t4d1, t4d2, t4d3, t4a3, // Thresholds

     output signed [15:0] d1_out, // Level 1 detail
     output signed [15:0] a1_out, // Level 1 approximation
     output signed [15:0] d2_out, // Level 2 detail
     output signed [15:0] a2_out, // Level 2 approximation
     output signed [15:0] d3_out, // Level 3 detail
     output signed [15:0] a3_out, // Level 3 approximation
     // Debug signals:
     output signed [15:0] s3_out, a3up_out, d3up_out, // L3
     output signed [15:0] s2_out, s3up_out, d2up_out, // L2
     output signed [15:0] s1_out, s2up_out, d1up_out, // L1
     output signed [15:0] y_out); // System output
// -----

    reg [2:0] count; // Cycle 2**max level
    reg signed [15:0] x, xd; // Input delays
    reg signed [15:0] a1, a2 ; // Analysis filter

```

```

wire signed [15:0] d1, d2, a3, d3 ; // Analysis filter
reg signed [15:0] d1t, d2t, d3t, a3t ;
                                // Before thresholding
wire signed [15:0] abs_d1t, abs_d2t, abs_d3t, abs_a3t ;
                                // Absolute values
reg signed [15:0] a1up, a3up, d3up ;
reg signed [15:0] a1upd, s3upd, a3upd, d3upd ;
reg signed [15:0] a1d, a2d; // Delay filter output
reg ena1, ena2, ena3; // Clock enables
reg t1, t2, t3; // Toggle flip-flops
reg signed [15:0] s2, s3up, s3, d2syn ;
reg signed [15:0] s1, s2up, s2upd ;
// Delay lines for d1 and d2
reg signed [15:0] d2upd [0:11];
reg signed [15:0] d1upd [0:29];

always @(posedge reset or posedge clk) // Control the
begin : FSM                          // system sample at clk rate
  if (reset) begin                  // Asynchronous reset
    count <= 0; ena1 <= 0; ena2 <= 0; ena3 <= 0;
  end else begin
    if (count == 7) count <= 0;
    else count <= count + 1;
    case (count) // Level 1 enable
      3'b001 : ena1 <= 1;
      3'b011 : ena1 <= 1;
      3'b101 : ena1 <= 1;
      3'b111 : ena1 <= 1;
      default : ena1 <= 0;
    endcase
    case (count) // Level 2 enable
      3'b001 : ena2 <= 1;
      3'b101 : ena2 <= 1;
      default : ena2 <= 0;
    endcase
    case (count) // Level 3 enable
      3'b101 : ena3 <= 1;
      default : ena3 <= 0;
    endcase
  end
end

// Haar analysis filter bank
always @(posedge reset or posedge clk)

```

```

begin : Analysis
  if (reset) begin          // Asynchronous clear
    x <= 0; xd <= 0; d1t <= 0; a1 <= 0; a1d <= 0;
    d2t <= 0; a2 <= 0; a2d <= 0; d3t <= 0; a3t <= 0;
  end else begin
    x <= x_in;
    xd <= x;
    if (ena1) begin // Level 1 analysis
      d1t <= x - xd;
      a1  <= x + xd;
      a1d <= a1;
    end
    if (ena2) begin // Level 2 analysis
      d2t <= a1 - a1d;
      a2  <= a1 + a1d;
      a2d <= a2;
    end
    if (ena3) begin // Level 3 analysis
      d3t <= a2 - a2d;
      a3t <= a2 + a2d;
    end
  end
end

// Take absolute values first
assign abs_d1t = (d1t>=0)? d1t : -d1t;
assign abs_d2t = (d2t>=0)? d2t : -d2t;
assign abs_d3t = (d3t>=0)? d3t : -d3t;
assign abs_a3t = (a3t>=0)? a3t : -a3t;

// Thresholding of d1, d2, d3 and a3
assign d1 = (abs_d1t > t4d1)? d1t : 0;
assign d2 = (abs_d2t > t4d2)? d2t : 0;
assign d3 = (abs_d3t > t4d3)? d3t : 0;
assign a3 = (abs_a3t > t4a3)? a3t : 0;

// Down followed by up sampling is implemented by setting
// every 2. value to zero
always @(posedge reset or posedge clk)
begin : Synthesis
  integer k;      // Loop variable
  if (reset) begin // Asynchronous clear
    t1 <= 0; t2 <= 0; t3 <= 0;
    s3up <= 0; s3upd <= 0;
  end
end

```

```

    d3up <= 0; a3up <= 0; a3upd<=0; d3upd <= 0;
    s3 <= 0; s2 <= 0;
    s1 <= 0; s2up <= 0; s2upd <= 0;
    for (k=0; k<=D2L+1; k=k+1) // delay to match s3up
        d2upd[k] <= 0;
    for (k=0; k<=D1L+1; k=k+1) // delay to match s2up
        d1upd[k] <= 0;
end else begin
    t1 <= ~t1; // toggle FF level 1
    if (t1) begin
        d1upd[0] <= d1;
        s2up <= s2;
    end else begin
        d1upd[0] <= 0;
        s2up <= 0;
    end
    s2upd <= s2up;
    for (k=1; k<=D1L+1; k=k+1) // Delay to match s2up
        d1upd[k] <= d1upd[k-1];
    s1 <= (s2up + s2upd -d1upd[D1L] +d1upd[D1L+1]) >>> 1;
    if (ena1) begin
        t2 <= ~t2; // toggle FF level 2
        if (t2) begin
            d2upd[0] <= d2;
            s3up <= s3;
        end else begin
            d2upd[0] <= 0;
            s3up <= 0;
        end
        s3upd <= s3up;
        for (k=1; k<=D2L+1; k=k+1) // Delay to match s3up
            d2upd[k] <= d2upd[k-1];
        s2 <= (s3up +s3upd -d2upd[D2L] +d2upd[D2L+1])>>> 1;
    end

    if (ena2) begin // Synthesis level 3
        t3 <= ~t3; // toggle FF
        if (t3) begin
            d3up <= d3;
            a3up <= a3;
        end else begin
            d3up <= 0;
            a3up <= 0;
        end
    end
end

```



```

        a3upd <= a3up;
        d3upd <= d3up;
        s3 <= (a3up + a3upd - d3up + d3upd) >>> 1;
    end
end
end

assign a1_out = a1; // Provide some test signal as outputs
assign d1_out = d1;
assign a2_out = a2;
assign d2_out = d2;
assign a3_out = a3;
assign d3_out = d3;
assign a3up_out = a3up;
assign d3up_out = d3up;
assign s3_out = s3;
assign s3up_out = s3up;
assign d2up_out = d2upd[D2L];
assign s2_out = s2;
assign s1_out = s1;
assign s2up_out = s2up;
assign d1up_out = d1upd[D1L];
assign y_out = s1;
assign ena1_out = ena1;
assign ena2_out = ena2;
assign ena3_out = ena3;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: rader7.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module rader7                                //---> Interface
(input  clk,                                // System clock
 input  reset,                              // Asynchronous reset
 input  [7:0]  x_in,                        // Real system input
 output reg signed[10:0] y_real, //Real system output
 output reg signed[10:0] y_imag); //Imaginary system output
// -----
    reg signed [10:0]  accu;                // Signal for X[0]
// Direct bit access of 2D vector in Quartus Verilog 2001
// works; no auxiliary signal for this purpose necessary
    reg signed [18:0] im [0:5];
    reg signed [18:0] re [0:5];

```

```

// real is keyword in Verilog and can not be an identifier
// Tapped delay line array
reg signed [18:0] x57, x111, x160, x200, x231, x250 ;
// The filter coefficients
reg signed [18:0] x5, x25, x110, x125, x256;
// Auxiliary filter coefficients
reg signed [7:0] x, x_0; // Signals for x[0]
reg [1:0] state; // State variable
reg [4:0] count; // Clock cycle counter

always @(posedge clk or posedge reset) // State machine
begin : States // for RADER filter
    parameter Start=0, Load=1, Run=2;

    if (reset) begin // Asynchronous reset
        state <= Start; accu <= 0;
        count <= 0; y_real <= 0; y_imag <= 0;
    end else
        case (state)
            Start : begin // Initialization step
                state <= Load;
                count <= 1;
                x_0 <= x_in; // Save x[0]
                accu <= 0 ; // Reset accumulator for X[0]
                y_real <= 0;
                y_imag <= 0;
            end
            Load : begin // Apply x[5],x[4],x[6],x[2],x[3],x[1]
                if (count == 8) // Load phase done ?
                    state <= Run;
                else begin
                    state <= Load;
                    accu <= accu + x;
                end
                count <= count + 1;
            end
            Run : begin // Apply again x[5],x[4],x[6],x[2],x[3]
                if (count == 15) begin // Run phase done ?
                    y_real <= accu; // X[0]
                    y_imag <= 0; // Only re inputs => Im(X[0])=0
                    count <= 0; // Output of result
                    state <= Start; // and start again
                end else begin
                    y_real <= (re[0] >>> 8) + x_0;

```

```

                                // i.e. re[0]/256+x[0]
        y_imag <= (im[0] >>> 8);    // i.e. im[0]/256
        state <= Run;
        count <= count + 1;
    end
end
endcase
end
// -----
always @(posedge clk or posedge reset) // Structure of the
begin : Structure                    // two FIR filters
    integer k;    // loop variable
    if (reset) begin                // in transposed form
        for (k=0; k<=5; k=k+1) begin
            re[k] <= 0; im[k] <= 0;    // Asynchronous clear
        end
        x <= 0;
    end else begin
        x <= x_in;
        // Real part of FIR filter in transposed form
        re[0] <= re[1] + x160 ;    // W^1
        re[1] <= re[2] - x231 ;    // W^3
        re[2] <= re[3] - x57  ;    // W^2
        re[3] <= re[4] + x160 ;    // W^6
        re[4] <= re[5] - x231 ;    // W^4
        re[5] <= -x57;            // W^5

        // Imaginary part of FIR filter in transposed form
        im[0] <= im[1] - x200 ;    // W^1
        im[1] <= im[2] - x111 ;    // W^3
        im[2] <= im[3] - x250 ;    // W^2
        im[3] <= im[4] + x200 ;    // W^6
        im[4] <= im[5] + x111 ;    // W^4
        im[5] <= x250;            // W^5
    end
end
end
// -----
always @(posedge clk or posedge reset) // Note that all
begin : Coeffs                        // signals are globally defined
    // Compute the filter coefficients and use FFs
    if (reset) begin                // Asynchronous clear
        x160 <= 0; x200 <= 0; x250 <= 0;
        x57 <= 0; x111 <= 0; x231 <= 0;
    end else begin

```

```

        x160 <= x5 <<< 5;          // i.e. 160 = 5 * 32;
        x200 <= x25 <<< 3;         // i.e. 200 = 25 * 8;
        x250 <= x125 <<< 1;        // i.e. 250 = 125 * 2;
        x57  <= x25 + (x <<< 5);   // i.e. 57 = 25 + 32;
        x111 <= x110 + x;          // i.e. 111 = 110 + 1;
        x231 <= x256 - x25;        // i.e. 231 = 256 - 25;
    end
end

always @*          // Note that all signals
begin : Factors    // are globally defined
// Compute the auxiliary factor for RAG without an FF
    x5  = (x <<< 2) + x; // i.e. 5 = 4 + 1;
    x25 = (x5 <<< 2) + x5; // i.e. 25 = 5*4 + 5;
    x110 = (x25 <<< 2) + (x5 <<< 2); // i.e. 110 = 25*4+5*4;
    x125 = (x25 <<< 2) + x25; // i.e. 125 = 25*4+25;
    x256 = x <<< 8; // i.e. 256 = 2 ** 8;
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fft256.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module fft256          //----> Interface
(input  clk,           // System clock
 input  reset,         // Asynchronous reset
 input signed [15:0] xr_in, xi_in, // Real and imag. input
 output reg fft_valid, // FFT output is valid
 output reg signed [15:0] fftr, ffti, // Real/imag. output
 output [8:0] rcount_o, // Bitreverse index counter
 output [15:0] xr_out0, // Real first in reg. file
 output [15:0] xi_out0, // Imag. first in reg. file
 output [15:0] xr_out1, // Real second in reg. file
 output [15:0] xi_out1, // Imag. second in reg. file
 output [15:0] xr_out255, // Real last in reg. file
 output [15:0] xi_out255, // Imag. last in reg. file
 output [8:0] stage_o, gcount_o, // Stage and group count
 output [8:0] i1_o, i2_o, // (Dual) data index
 output [8:0] k1_o, k2_o, // Index offset
 output [8:0] w_o, dw_o, // Cos/Sin (increment) angle
 output reg [8:0] wo); // Decision tree location loop FSM
// -----

```

```

reg[2:0] s; // State machine variable
parameter start=0, load=1, calc=2, update=3,
reverse=4, done=5;

reg [8:0] w;
reg signed [15:0] sin, cos;
reg signed [15:0] tr, ti;
// Double length product
reg signed [31:0] cos_tr, sin_ti, cos_ti, sin_tr;
reg [15:0] cos_rom[127:0];
reg [15:0] sin_rom[127:0];
reg [8:0] i1, i2, gcount, k1, k2;
reg [8:0] stage, dw;
reg [7:0] rcount;

reg [7:0] slv, rslv;
wire [8:0] N, ldN;
assign N = 256; // Number of points
assign ldN = 8; // Log_2 number of points
// Register array for 16 bit precision:
reg [15:0] xr[255:0];
reg [15:0] xi[255:0];

initial
begin
    $readmemh("cos128x16.txt", cos_rom);
    $readmemh("sin128x16.txt", sin_rom);
end

always @ (negedge clk or posedge reset)
    if (reset == 1) begin
        cos <= 0; sin <= 0;
    end else begin
        cos <= cos_rom[w[6:0]];
        sin <= sin_rom[w[6:0]];
    end

always @(posedge reset or posedge clk)
begin : States // FFT in behavioral style
    integer k;
    reg [8:0] count;
    if (reset) begin // Asynchronous reset
        s <= start; count = 0;
        gcount = 0; stage= 1; i1 = 0; i2 = N/2; k1=N;
    end
end

```

```

        k2=N/2; dw = 1; fft_valid <= 0;
        fftr <= 0; ffti <= 0; wo <= 0;
end else
    case (s)                                // Next State assignments
    start : begin
        s <= load; count <= 0; w <= 0;
        gcount = 0; stage= 1; i1 = 0; i2 = N/2; k1=N;
        k2=N/2; dw = 1; fft_valid <= 0; rcount <= 0;
    end
    load : begin                            // Read in all data from I/O ports
        xr[count] <= xr_in; xi[count] <= xi_in;
        count <= count + 1;
        if (count == N) s <= calc;
        else s <= load;
    end
    calc : begin                            // Do the butterfly computation
        tr = xr[i1] - xr[i2];
        xr[i1] <= xr[i1] + xr[i2];
        ti = xi[i1] - xi[i2];
        xi[i1] <= xi[i1] + xi[i2];
        cos_tr = cos * tr; sin_ti = sin * ti;
        xr[i2] <= (cos_tr >>> 14) + (sin_ti >>> 14);
        cos_ti = cos * ti; sin_tr = sin * tr;
        xi[i2] <= (cos_ti >>> 14) - (sin_tr >>> 14);
        s <= update;
    end
    update : begin                          // all counters and pointers
        s <= calc;                          // by default do next butterfly
        i1 = i1 + k1;                        // next butterfly in group
        i2 = i1 + k2;
        wo <= 1;
        if ( i1 >= N-1 ) begin // all butterfly
            gcount = gcount + 1; // done in group?
            i1 = gcount;
            i2 = i1 + k2;
            wo <= 2;
            if ( gcount >= k2 ) begin // all groups done
                gcount = 0; i1 = 0; i2 = k2; // in stages?
                dw = dw * 2;
                stage = stage + 1;
                wo <= 3;
                if (stage > ldN) begin // all stages done
                    s <= reverse;
                    count = 0;

```

```

        wo <= 4;
    end else begin // start new stage
        k1 = k2; k2 = k2/2;
        i1 = 0; i2 = k2;
        w <= 0;
        wo <= 5;
    end
    end else begin // start new group
        i1 = gcount; i2 = i1 + k2;
        w <= w + dw;
        wo <= 6;
    end
end
end
reverse : begin // Apply Bit Reverse
    fft_valid <= 1;
    for (k=0;k<=7;k=k+1) rcount[k] = count[7-k];
    fftr <= xr[rcount]; ffti <= xi[rcount];
    count = count + 1;
    if (count >= N) s <= done;
    else
        s <= reverse;
    end
done : begin // Output of results
    s <= start; // start next cycle
end
endcase
end

assign xr_out0 = xr[0];
assign xi_out0 = xi[0];
assign xr_out1 = xr[1];
assign xi_out1 = xi[1];
assign xr_out255 = xr[255];
assign xi_out255 = xi[255];
assign i1_o = i1; // Provide some test signals as outputs
assign i2_o = i2;
assign stage_o = stage;
assign gcount_o = gcount;
assign k1_o = k1;
assign k2_o = k2;
assign w_o = w;
assign dw_o = dw;
assign rcount_o = rcount;
assign w_out = w;

```

```

    assign cos_out = cos;
    assign sin_out = sin;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: lfsr.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module lfsr          //----> Interface
    (input clk,      // System clock
     input reset,    // Asynchronous reset
     output [6:1] y); // System output
// -----
    reg [6:1] ff; // Note that reg is keyword in Verilog and
                  // can not be variable name
    integer i;    // loop variable

    always @(posedge clk or posedge reset)
    begin // Length 6 LFSR with xnor
        if (reset)          // Asynchronous clear
            ff <= 0;
        else begin
            ff[1] <= ff[5] ^^ ff[6]; //Use non-blocking assignment
            for (i=6; i>=2 ; i=i-1) //Tapped delay line: shift one
                ff[i] <= ff[i-1];
        end
    end

    assign    y = ff;          // Connect to I/O pins

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: lfsr6s3.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module lfsr6s3       //----> Interface
    (input clk,      // System clock
     input reset,    // Asynchronous reset
     output [6:1] y); // System output
// -----
    reg [6:1] ff; // Note that reg is keyword in Verilog and

```



```

// can not be variable name

always @(posedge clk or posedge reset)
begin
    if (reset)                // Asynchronous clear
        ff <= 0;
    else begin                // Implement three-step
        ff[6] <= ff[3];        // length-6 LFSR with xnor;
        ff[5] <= ff[2];        // use non-blocking assignments
        ff[4] <= ff[1];
        ff[3] <= ff[5] ^^ ff[6];
        ff[2] <= ff[4] ^^ ff[5];
        ff[1] <= ff[3] ^^ ff[4];
    end
end

assign y = ff;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: ammod.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module ammod #(parameter W = 8) // Bit width - 1
(input      clk,                // System clock
 input      reset,              // Asynchronous reset
 input signed [W:0] r_in,       // Radius input
 input signed [W:0] phi_in,     // Phase input
 output reg signed [W:0] x_out, // x or real part output
 output reg signed [W:0] y_out, // y or imaginary part
 output reg signed [W:0] eps); // Error of results
// -----
    reg signed [W:0] x [0:3]; // There is bit access in 2D
    reg signed [W:0] y [0:3]; // array types in
    reg signed [W:0] z [0:3]; // Quartus Verilog 2001

always @(posedge clk or posedge reset)
begin: Pipeline
    integer k; // Loop variable
    if (reset) begin
        for (k=0; k<=3; k=k+1) begin // Asynchronous clear
            x[k] <= 0; y[k] <= 0; z[k] <= 0;
        end
    end
end

```

```

        x_out <= 0; eps <= 0; y_out <= 0;
    end
else begin

    if (phi_in > 90) begin // Test for |phi_in| > 90
        x[0] <= 0; // Rotate 90 degrees
        y[0] <= r_in; // Input in register 0
        z[0] <= phi_in - 'sd90;
    end else if (phi_in < - 90) begin
        x[0] <= 0;
        y[0] <= - r_in;
        z[0] <= phi_in + 'sd90;
    end else begin
        x[0] <= r_in;
        y[0] <= 0;
        z[0] <= phi_in;
    end

    if (z[0] >= 0) begin // Rotate 45 degrees
        x[1] <= x[0] - y[0];
        y[1] <= y[0] + x[0];
        z[1] <= z[0] - 'sd45;
    end else begin
        x[1] <= x[0] + y[0];
        y[1] <= y[0] - x[0];
        z[1] <= z[0] + 'sd45;
    end

    if (z[1] >= 0) begin // Rotate 26 degrees
        x[2] <= x[1] - (y[1] >>> 1); // i.e. x[1] - y[1] /2
        y[2] <= y[1] + (x[1] >>> 1); // i.e. y[1] + x[1] /2
        z[2] <= z[1] - 'sd26;
    end else begin
        x[2] <= x[1] + (y[1] >>> 1); // i.e. x[1] + y[1] /2
        y[2] <= y[1] - (x[1] >>> 1); // i.e. y[1] - x[1] /2
        z[2] <= z[1] + 'sd26;
    end

    if (z[2] >= 0) begin // Rotate 14 degrees
        x[3] <= x[2] - (y[2] >>> 2); // i.e. x[2] - y[2] /4
        y[3] <= y[2] + (x[2] >>> 2); // i.e. y[2] + x[2] /4
        z[3] <= z[2] - 'sd14;
    end else begin
        x[3] <= x[2] + (y[2] >>> 2); // i.e. x[2] + y[2] /4

```

```

        y[3] <= y[2] - (x[2] >>> 2); // i.e.  $y[2] - x[2]/4$ 
        z[3] <= z[2] + 'sd14;
    end

    x_out <= x[3];
    eps  <= z[3];
    y_out <= y[3];
end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fir_lms.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// This is a generic LMS FIR filter generator
// It uses W1 bit data/coefficients bits
module fir_lms //----> Interface
    #(parameter W1 = 8, // Input bit width
        W2 = 16, // Multiplier bit width 2*W1
        L = 2, // Filter length
        Delay = 3) // Pipeline steps of multiplier
    (input clk, // System clock
    input reset, // Asynchronous reset
    input signed [W1-1:0] x_in, //System input
    input signed [W1-1:0] d_in, // Reference input
    output signed [W1-1:0] f0_out, // 1. filter coefficient
    output signed [W1-1:0] f1_out, // 2. filter coefficient
    output signed [W2-1:0] y_out, // System output
    output signed [W2-1:0] e_out); // Error signal
// -----
// Signed data types are supported in 2001
// Verilog, and used whenever possible
    reg signed [W1-1:0] x [0:1]; // Data array
    reg signed [W1-1:0] f [0:1]; // Coefficient array
    reg signed [W1-1:0] d;
    wire signed [W1-1:0] emu;
    reg signed [W2-1:0] p [0:1]; // 1. Product array
    reg signed [W2-1:0] xemu [0:1]; // 2. Product array
    wire signed [W2-1:0] y, sxtty, e, sxtdd;
    wire signed [W2-1:0] sum; // Auxilary signals

    always @(posedge clk or posedge reset)

```

```

begin: Store          // Store these data or coefficients
  if (reset) begin    // Asynchronous clear
    d <= 0; x[0] <= 0; x[1] <= 0; f[0] <= 0; f[1] <= 0;
  end else begin
    d <= d_in; // Store desired signal in register
    x[0] <= x_in; // Get one data sample at a time
    x[1] <= x[0]; // shift 1
    f[0] <= f[0] + xemu[0][15:8]; // implicit divide by 2
    f[1] <= f[1] + xemu[1][15:8];
  end
end

// Instantiate L multiplier
always @(*)
begin : MulGen1
  integer I; // loop variable
  for (I=0; I<L; I=I+1) p[I] <= x[I] * f[I];
end

assign y = p[0] + p[1]; // Compute ADF output

// Scale y by 128 because x is fraction
assign e = d - (y >>> 7) ;
assign emu = e >>> 1; // e*mu divide by 2 and
                      // 2 from xemu makes mu=1/4

// Instantiate L multipliers
always @(*)
begin : MulGen2
  integer I; // loop variable
  for (I=0; I<=L-1; I=I+1) xemu[I] <= emu * x[I];
end

assign y_out = y >>> 7; // Monitor some test signals
assign e_out = e;
assign f0_out = f[0];
assign f1_out = f[1];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fir4dlms.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****

```

```

// This is a generic DLMS FIR filter generator
// It uses W1 bit data/coefficients bits
module fir4dlms          //----> Interface
#(parameter W1 = 8,      // Input bit width
      W2 = 16,           // Multiplier bit width 2*W1
      L = 2)             // Filter length
  (input clk,              // System clock
   input reset,            // Asynchronous reset
   input signed [W1-1:0] x_in, //System input
   input signed [W1-1:0] d_in,  // Reference input
   output signed [W1-1:0] f0_out, // 1. filter coefficient
   output signed [W1-1:0] f1_out, // 2. filter coefficient
   output signed [W2-1:0] y_out,  // System output
   output signed [W2-1:0] e_out); // Error signal
// -----
// 2D array types memories are supported by Quartus II
// in Verilog, use therefore single vectors
  reg signed [W1-1:0] x[0:4];
  reg signed [W1-1:0] f[0:1];
  reg signed [W1-1:0] d[0:2]; // Desired signal array
  wire signed [W1-1:0] emu;
  reg signed [W2-1:0] xemu[0:1]; // Product array
  reg signed [W2-1:0] p[0:1];    // double bit width
  reg signed [W2-1:0] y, e, sxted;

  always @(posedge clk or posedge reset) // Store these data
  begin: Store                          // or coefficients
    integer k;      // loop variable
    if (reset) begin // Asynchronous clear
      for (k=0; k<=2; k=k+1) d[k] <= 0;
      for (k=0; k<=4; k=k+1) x[k] <= 0;
      for (k=0; k<=1; k=k+1) f[k] <= 0;
    end else begin
      d[0] <= d_in; // Shift register for desired data
      d[1] <= d[0];
      d[2] <= d[1];
      x[0] <= x_in; // Shift register for data
      x[1] <= x[0];
      x[2] <= x[1];
      x[3] <= x[2];
      x[4] <= x[3];
      f[0] <= f[0] + xemu[0][15:8]; // implicit divide by 2
      f[1] <= f[1] + xemu[1][15:8];
    end
  end

```

```

end

// Instantiate L pipelined multiplier
always @(posedge clk or posedge reset)
begin : Mul
    integer k, I;    // loop variable
    if (reset) begin    // Asynchronous clear
        for (k=0; k<=L-1; k=k+1) begin
            p[k] <= 0;
            xemu[k] <= 0;
        end
        y <= 0; e <= 0;
    end else begin
        for (I=0; I<L; I=I+1) begin
            p[I] <= x[I] * f[I];
            xemu[I] <= emu * x[I+3];
        end
        y <= p[0] + p[1]; // Compute ADF output
        // Scale y by 128 because x is fraction
        e <= d[2] - (y >>> 7);
    end
end

assign emu = e >>> 1; // e*mu divide by 2 and
                    // 2 from xemu makes mu=1/4

assign y_out = y >>> 7; // Monitor some test signals
assign e_out = e;
assign f0_out = f[0];
assign f1_out = f[1];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: pca.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module pca // -----> Interface
    (input clk,                // System clock
     input reset,              // Asynchron reset
     input signed [31:0] s1_in, // 1. signal input
     input signed [31:0] s2_in, // 2. signal input
     input signed [31:0] mu1_in, // Learning rate 1. PC

```

```

input  signed [31:0] mu2_in, // Learning rate 2. PC
output signed [31:0] x1_out, // Mixing 1. output
output signed [31:0] x2_out, // Mixing 2. output
output signed [31:0] w11_out, // Eigenvector [1,1]
output signed [31:0] w12_out, // Eigenvector [1,2]
output signed [31:0] w21_out, // Eigenvector [2,1]
output signed [31:0] w22_out, // Eigenvector [2,2]
output reg signed [31:0] y1_out, // 1. PC output
output reg signed [31:0] y2_out); // 2. PC output
// -----
// All data and coefficients are in 16.16 format
reg signed [31:0] s, s1, s2, x1, x2, w11, w12, w21, w22;
reg signed [31:0] h11, h12, y1, y2, mu1, mu2;
// Product double bit width
reg signed [63:0] a11s1, a12s2, a21s1, a22s2;
reg signed [63:0] x1w11, x2w12, w11y1, mu1y1, p12;
reg signed [63:0] x1w21, x2w22, w21y2, p21, w22y2;
reg signed [63:0] mu2y2, p22, p11;

wire signed [31:0] a11, a12, a21, a22, ini;
assign a11 = 32'h0000C000; // 0.75
assign a12 = 32'h00018000; // 1.5
assign a21 = 32'h00008000; // 0.5
assign a22 = 32'h00005555; // 0.333333
assign ini = 32'h00008000; // 0.5

always @(posedge clk or posedge reset)
begin : P1 // PCA using Sanger GHA
  if (reset) begin // reset/initialize all registers
    x1 <= 0; x2 <= 0; y1_out <= 0; y2_out <= 0;
    w11 <= ini; w12 <= ini; s1 <= 0; mu1 <= 0;
    w21 <= ini; w22 <= ini; s2 <= 0; mu2 <= 0;
  end else begin
    s1 <= s1_in; // place inputs in registers
    s2 <= s2_in;
    mu1 <= mu1_in;
    mu2 <= mu2_in;

    // Mixing matrix
    a11s1 = a11 * s1; a12s2 = a12 * s2;
    x1 <= (a11s1 >>> 16) + (a12s2 >>> 16);
    a21s1 = a21 * s1; a22s2 = a22 * s2;
    x2 <= (a21s1 >>> 16) - (a22s2 >>> 16);

```

```

// first PC and eigenvector
x1w11 = x1 * w11;
x2w12 = x2 * w12;
y1 = (x1w11 >>> 16) + (x2w12 >>> 16);
w11y1 = w11 * y1;
mul1y1 = mul1 * y1;
h11 = w11y1 >>> 16;
p11 = (mul1y1 >>> 16) * (x1 - h11);
w11 <= w11 + (p11 >>> 16);

h12 = (w12 * y1) >>> 16;
p12 = (x2 - h12) * (mul1y1 >>> 16);
w12 <= w12 + (p12 >>> 16);

// second PC and eigenvector
x1w21 = x1 * w21; x2w22 = x2 * w22;
y2 = (x1w21 >>> 16) + (x2w22 >>> 16);
w21y2 = w21 * y2;
mu2y2 = mu2 * y2;
p21 = (mu2y2 >>> 16) * (x1 - h11 - (w21y2 >>> 16));
w21 <= w21 + (p21 >>> 16);

w22y2 = w22 * y2;
p22 = (mu2y2 >>> 16) * (x2 - h12 - (w22y2 >>> 16));
w22 <= w22 + (p22 >>> 16);
// registers y output
y1_out <= y1;
y2_out <= y2;
end
end

// Redefine bits as 32 bit SLV
assign x1_out = x1;
assign x2_out = x2;
assign w11_out = w11;
assign w12_out = w12;
assign w21_out = w21;
assign w22_out = w22;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: ica.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org

```



```

//*****
module ica // -----> Interface
    (input clk,                // System clock
     input reset,              // Asynchron reset
     input  signed [31:0] s1_in, // 1. signal input
     input  signed [31:0] s2_in, // 2. signal input
     input  signed [31:0] mu_in, // Learning rate
     output signed [31:0] x1_out, // Mixing 1. output
     output signed [31:0] x2_out, // Mixing 2. output
     output signed [31:0] B11_out, // Demixing 1,1
     output signed [31:0] B12_out, // Demixing 1,2
     output signed [31:0] B21_out, // Demixing 2,1
     output signed [31:0] B22_out, // Demixing 2,2
     output reg signed [31:0] y1_out, // 1. component output
     output reg signed [31:0] y2_out); // 2. component output
// -----
// All data and coefficients are in 16.16 format

    reg signed [31:0] s, s1, s2, x1, x2, B11, B12, B21, B22;
    reg signed [31:0] y1, y2, f1, f2, H11, H12, H21, H22, mu;
    reg signed [31:0] DB11, DB12, DB21, DB22;
    // Product double bit width
    reg signed [63:0] a11s1, a12s2, a21s1, a22s2;
    reg signed [63:0] x1B11, x2B12, x1B21, x2B22;
    reg signed [63:0] y1y1, y2f1, y1y2, y1f2, y2y2;
    reg signed [63:0] muDB11, muDB12, muDB21, muDB22;
    reg signed [63:0] B11H11, H12B21, B12H11, H12B22;
    reg signed [63:0] B11H21, H22B21, B12H21, H22B22;

    wire signed [31:0] a11, a12, a21, a22, one, negone;
    assign a11 = 32'h0000C000; // 0.75
    assign a12 = 32'h00018000; // 1.5
    assign a21 = 32'h00008000; // 0.5
    assign a22 = 32'h00005555; // 0.333333
    assign one = 32'h00010000; // 1.0
    assign negone = 32'hFFFF0000; // -1.0

    always @(posedge clk or posedge reset)
    begin : P1 // ICA using EASI
        if (reset) begin // reset/initialize all registers
            x1 <= 0; x2 <= 0; y1_out <= 0; y2_out <= 0;
            B11 <= one; B12 <= 0; s1 <= 0; mu <= 0;
            B21 <= 0; B22 <= one; s2 <= 0;
        end else begin

```

```

    s1 <= s1_in; // place inputs in registers
    s2 <= s2_in;
    mu <= mu_in;

    // Mixing matrix
    a11s1 = a11 * s1; a12s2 = a12 * s2;
    x1 <= (a11s1 >>> 16) + (a12s2 >>> 16);
    a21s1 = a21 * s1; a22s2 = a22 * s2;
    x2 <= (a21s1 >>> 16) - (a22s2 >>> 16);
    // New y values first
    x1B11 = x1 * B11; x2B12 = x2 * B12;
    y1 = (x1B11 >>> 16) + (x2B12 >>> 16);
    x1B21 = x1 * B21; x2B22 = x2 * B22;
    y2 = (x1B21 >>> 16) + (x2B22 >>> 16);
    // compute the H matrix
    // Build tanh approximation function for f1
    if (y1 > one) f1 = one;
    else if (y1 < negone) f1 = negone;
    else f1 = y1;
    // Build tanh approximation function for f2
    if (y2 > one) f2 = one;
    else if (y2 < negone) f2 = negone;
    else f2 = y2;

    y1y1 = y1 * y1;
    H11 = one - (y1y1 >>> 16) ;
    y2f1 = f1 * y2; y1y2 = y1 * y2; y1f2 = y1 * f2;
    H12 = (y2f1 >>> 16) - (y1y2 >>> 16) - (y1f2 >>> 16);
    H21 = (y1f2 >>> 16) - (y1y2 >>> 16) - (y2f1 >>> 16);
    y2y2 = y2 * y2;
    H22 = one - (y2y2 >>> 16);
    // update matrix Delta B
    B11H11 = B11 * H11; H12B21 = H12 * B21;
    DB11 = (B11H11 >>> 16) + (H12B21 >>> 16);
    B12H11 = B12 * H11; H12B22 = H12 * B22;
    DB12 = (B12H11 >>> 16) + (H12B22 >>> 16);
    B11H21 = B11 * H21; H22B21 = H22 * B21;
    DB21 = (B11H21 >>> 16) + (H22B21 >>> 16);
    B12H21 = B12 * H21; H22B22 = H22 * B22;
    DB22 = (B12H21 >>> 16) + (H22B22 >>> 16);
    // Store update matrix B in registers
    muDB11 = mu * DB11; muDB12 = mu * DB12;
    muDB21 = mu * DB21; muDB22 = mu * DB22;
    B11 <= B11 + (muDB11 >>> 16) ;

```

```

        B12 <= B12 + (muDB12 >>> 16);
        B21 <= B21 + (muDB21 >>> 16);
        B22 <= B22 + (muDB22 >>> 16);
    // register y output
    y1_out <= y1;
    y2_out <= y2;
end
end

assign x1_out = x1; // Redefine bits as 32 bit SLV
assign x2_out = x2;
assign B11_out = B11;
assign B12_out = B12;
assign B21_out = B21;
assign B22_out = B22;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: g711alaw.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// G711 includes A and mu-law coding for speech signals:
// A ~ = 87.56; |x| <= 4095, i.e., 12 bit plus sign
// mu ~ = 255; |x| <= 8160, i.e., 14 bit
// -----
module g711alaw #(parameter WIDTH = 13) // Bit width
    (input clk, // System clock
    input reset, // Asynchron reset
    input signed [12:0] x_in, // System input
    output reg signed [7:0] enc, // Encoder output
    output reg signed [12:0] dec, // Decoder output
    output signed [13:0] err); // Error of results
// -----
    wire s;
    wire signed [12:0] x; // Auxiliary vectors
    wire signed [12:0] dif;
// -----
    assign s = x_in[WIDTH -1]; // sign magnitude not 2C!
    assign x = {1'b0,x_in[WIDTH-2:0]};
    assign dif = dec - x_in; // Difference
    assign err = (dif>0)? dif : -dif; // Absolute error
// -----
    always @*

```

```

begin : Encode          // Mini floating-point format encoder
  if ((x>=0) && (x<=63))
    enc <= {s,2'b00,x[5:1]}; // segment 1
  else if ((x>=64) && (x<=127))
    enc <= {s,3'b010,x[5:2]}; // segment 2
  else if ((x>=128) && (x<=255))
    enc <= {s,3'b011,x[6:3]}; // segment 3
  else if ((x>=256) && (x<=511))
    enc <= {s,3'b100,x[7:4]}; // segment 4
  else if ((x>=512) && (x<=1023))
    enc <= {s,3'b101,x[8:5]}; // segment 5
  else if ((x>=1024) && (x<=2047))
    enc <= {s,3'b110,x[9:6]}; // segment 6
  else if ((x>=2048) && (x<=4095))
    enc <= {s,3'b111,x[10:7]}; // segment 7
  else enc <= {s,7'b0000000}; // + or - 0
end
// -----
always @*
begin : Decode // Mini floating point format decoder
  case (enc[6:4])
    3'b000 : dec <= {s,6'b000000,enc[4:0],1'b1};
    3'b010 : dec <= {s,6'b000001,enc[3:0],2'b10};
    3'b011 : dec <= {s,5'b00001,enc[3:0],3'b100};
    3'b100 : dec <= {s,4'b0001,enc[3:0],4'b1000};
    3'b101 : dec <= {s,3'b001,enc[3:0],5'b10000};
    3'b110 : dec <= {s,2'b01,enc[3:0],6'b100000};
    default : dec <= {s,1'b1,enc[3:0],7'b1000000};
  endcase
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: adpcm.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module adpcm          //----> Interface
(input clk,           // System clock
 input reset,         // Asynchron reset
 input signed [15:0] x_in, // Input to encoder
 output [3:0] y_out,   // 4 bit ADPCM coding word
 output signed [15:0] p_out, // Predictor/decoder output
 output reg p_underflow, p_overflow, // Predictor flags

```

```

output signed [7:0] i_out,           // Index to table
output reg i_underflow, i_overflow, // Index flags
output signed [15:0] err,           // Error of system
output [14:0] sz_out,               // Step size
output s_out);                     // Sign bit
// -----
reg signed [15:0] va, va_d; // Current signed adpcm input
reg sign ; // Current adpcm sign bit
reg [3:0] sdelta; // Current signed adpcm output
reg [14:0] step; // Stepsize
reg signed [15:0] sstep; // Stepsize including sign
reg signed [15:0] valpred; // Predicted output value
reg signed [7:0] index; // Current step change index

reg signed [16:0] diff0, diff1, diff2, diff3;
// Difference val - valprev
reg signed [16:0] p1, p2, p3; // Next valpred
reg signed [7:0] i1, i2, i3; // Next index
reg [3:0] delta2, delta3, delta4;
// Current absolute adpcm output
reg [14:0] tStep;
reg signed [15:0] vpdiff2, vpdiff3, vpdiff4 ;
// Current change to valpred

// Quantization lookup table has 89 entries
wire [14:0] t [0:88];

// ADPCM step variation table
wire signed [4:0] indexTable [0:15];

assign indexTable[0]=-1; assign indexTable[1]=-1;
assign indexTable[2]=-1; assign indexTable[3]=-1;
assign indexTable[4]=2; assign indexTable[5]=4;
assign indexTable[6]=6; assign indexTable[7]=8;
assign indexTable[8]=-1; assign indexTable[9]=-1;
assign indexTable[10]=-1; assign indexTable[11]=-1;
assign indexTable[12]=2; assign indexTable[13]=4;
assign indexTable[14]=6; assign indexTable[15]=8;
// -----
assign t[0]=7; assign t[1]=8; assign t[2]=9;
assign t[3]=10; assign t[4]=11; assign t[5]=12;
assign t[6]= 13; assign t[7]= 14; assign t[8]= 16;
assign t[9]= 17; assign t[10]= 19; assign t[11]= 21;
assign t[12]= 23; assign t[13]= 25; assign t[14]= 28;

```

```

assign t[15]= 31; assign t[16]= 34; assign t[17]= 37;
assign t[18]= 41; assign t[19]= 45; assign t[20]= 50;
assign t[21]= 55; assign t[22]= 60; assign t[23]= 66;
assign t[24]= 73; assign t[25]= 80; assign t[26]= 88;
assign t[27]= 97; assign t[28]= 107; assign t[29]= 118;
assign t[30]= 130; assign t[31]= 143; assign t[32]= 157;
assign t[33]= 173; assign t[34]= 190; assign t[35]= 209;
assign t[36]= 230; assign t[37]= 253; assign t[38]= 279;
assign t[39]= 307; assign t[40]= 337; assign t[41]= 371;
assign t[42]= 408; assign t[43]= 449; assign t[44]= 494;
assign t[45]= 544; assign t[46]= 598; assign t[47]= 658;
assign t[48]= 724; assign t[49]= 796; assign t[50]= 876;
assign t[51]= 963; assign t[52]= 1060; assign t[53]= 1166;
assign t[54]= 1282; assign t[55]= 1411; assign t[56]= 1552;
assign t[57]= 1707; assign t[58]= 1878; assign t[59]= 2066;
assign t[60]= 2272; assign t[61]= 2499; assign t[62]= 2749;
assign t[63]= 3024; assign t[64]= 3327; assign t[65]= 3660;
assign t[66]= 4026; assign t[67]= 4428; assign t[68]= 4871;
assign t[69]= 5358; assign t[70]= 5894; assign t[71]= 6484;
assign t[72]= 7132; assign t[73]= 7845; assign t[74]= 8630;
assign t[75]= 9493; assign t[76]= 10442; assign t[77]= 11487;
assign t[78]= 12635; assign t[79]= 13899;
assign t[80]= 15289; assign t[81]= 16818;
assign t[82]= 18500; assign t[83]= 20350;
assign t[84]= 22385; assign t[85]= 24623;
assign t[86]= 27086; assign t[87]= 29794;
assign t[88]= 32767;
// -----
always @(posedge clk or posedge reset)
begin : Encode
    if (reset) begin // Asynchronous clear
        va <= 0; va_d <= 0;
        step <= 0; index <= 0;
        valpred <= 0;
    end else begin // Store in register
        va_d <= va; // Delay signal for error comparison
        va <= x_in;
        step <= t[i3];
        index <= i3;
        valpred <= p3; // Store predicted in register
    end
end

always @(va, va_d, step, index, valpred) begin

```

```

// ----- State 1: Compute difference from predicted sample
diff0 = va - valpred;
if (diff0 < 0) begin
    sign = 1;          // Set sign bit if negative
    diff1 = -diff0;    // Use absolute value for quantization
end else begin
    sign = 0;
    diff1 = diff0;
end
// State 2: Quantize by division and
// State 3: compute inverse quantization
// Compute: delta=floor(diff(k)*4./step(k)); and
// vpdiff(k)=floor((delta(k)+.5).*step(k)/4);
if (diff1 >= step) begin // bit 2
    delta2 = 4;
    diff2 = diff1 - step;
    vpdiff2 = step/8 + step;
end else begin
    delta2 = 0;
    diff2 = diff1;
    vpdiff2 = step/8;
end
if (diff2 >= step/2) begin //// bit3
    delta3 = delta2 + 2 ;
    diff3 = diff2 - step/2;
    vpdiff3 = vpdiff2 + step/2;
end else begin
    delta3 = delta2;
    diff3 = diff2;
    vpdiff3 = vpdiff2;
end
if (diff3 >= step/4) begin
    delta4 = delta3 + 1;
    vpdiff4 = vpdiff3 + step/4;
end else begin
    delta4 = delta3;
    vpdiff4 = vpdiff3;
end
// State 4: Adjust predicted sample based on inverse
if (sign)                // quantized
    p1 = valpred - vpdiff4;
else
    p1 = valpred + vpdiff4;
//----- State 5: Threshold to maximum and minimum -----

```

```

    if (p1 > 32767) begin // Check for 16 bit range
        p2 = 32767; p_overflow <= 1; // 215-1 two's complement
    end else begin
        p2 = p1; p_overflow <= 0;
    end
    if (p2 < -32768) begin // -215
        p3 = -32768; p_underflow <= 1;
    end else begin
        p3 = p2; p_underflow <= 0;
    end

// State 6: Update the stepsize and index for stepsize LUT
    i1 = index + indexTable[delta4];
    if (i1 < 0) begin // Check index range [0...88]
        i2 = 0; i_underflow <= 1;
    end else begin
        i2 = i1; i_underflow <= 0;
    end
    if (i2 > 88) begin
        i3 = 88; i_overflow <= 1;
    end else begin
        i3 = i2; i_overflow <= 0;
    end
    if (sign)
        sdelta = delta4 + 8;
    else
        sdelta = delta4;
end

assign y_out = sdelta; // Monitor some test signals
assign p_out = valpred;
assign i_out = index;
assign sz_out = step;
assign s_out = sign;
assign err = va_d - valpred;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: reg_file.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Description: This is a W x L bit register file.
module reg_file #(parameter W = 7, // Bit width - 1

```



```

                                N = 15) // Number of registers - 1
(input clk,                    // System clock
 input reset,                  // Asynchronous reset
 input reg_ena,                // Write enable active 1
 input [W:0] data,             // System input
 input [3:0] rd,               // Address for write
 input [3:0] rs,               // 1. read address
 input [3:0] rt,               // 2. read address
 output reg [W:0] s,           // 1. data
 output reg [W:0] t);          // 2. data
// -----
reg [W:0] r [0:N];

always @(posedge clk or posedge reset)
begin : MUX                    // Input mux inferring registers
    integer k;                // loop variable
    if (reset)                 // Asynchronous clear
        for (k=0; k<=N; k=k+1) r[k] <= 0;
    else if ((reg_ena == 1) && (rd > 0))
        r[rd] <= data;
end

// 2 output demux without registers
always @*
begin : DEMUX
    if (rs > 0) // First source
        s = r[rs];
    else
        s = 0;
    if (rt > 0) // Second source
        t = r[rt];
    else
        t = 0;
end

endmodule

```

```

//*****
// IEEE STD 1364-2001 Verilog file: trisc0.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Title: T-RISC stack machine
// Description: This is the top control path/FSM of the
// T-RISC, with a single 3 phase clock cycle design

```

```

// It has a stack machine/0-address type instruction word
// The stack has only 4 words.
// -----
module trisc0 #(parameter WA = 7, // Address bit width -1
                WD = 7) // Data bit width -1
(input  clk,           // System clock
 input  reset,         // Asynchronous reset
 output jc_out,        // Jump condition flag
 output me_ena,        // Memory enable
 input  [WD:0] iport,  // Input port
 output reg [WD:0] oport, // Output port
 output signed [WD:0] s0_out, // Stack register 0
 output signed [WD:0] s1_out, // Stack register 1
 output [WD:0] dmd_in,   // Data memory data read
 output [WD:0] dmd_out,  // Data memory data read
 output [WD:0] pc_out,   // Program counter
 output [WD:0] dma_out,  // Data memory address write
 output [WD:0] dma_in,   // Data memory address read
 output [7:0] ir_imm,    // Immediate value
 output [3:0] op_code); // Operation code
// -----
//parameter ifetch=0, load=1, store=2, incpc=3;
reg [1:0] state;

wire [3:0] op;
wire [WD:0] imm, dmd;
reg signed [WD:0] s0, s1, s2, s3;
reg [WA:0] pc;
wire [WA:0] dma;
wire [11:0] pmd, ir;
wire eq, ne, not_clk;
reg mem_ena, jc;

// OP Code of instructions:
parameter
add  = 0, neg  = 1, sub  = 2, opand = 3, opor = 4,
inv  = 5, mul  = 6, pop  = 7, pushi = 8, push = 9,
scan = 10, print = 11, cne = 12, ceq  = 13, cjp = 14,
jmp  = 15;

always @(*) // sequential FSM of processor
    // Check store in register ?
    case (op) // always store except Branch
        pop      : mem_ena <= 1;

```

```

        default : mem_ena <= 0;
    endcase

always @(negedge clk or posedge reset)
    if (reset == 1) // update the program counter
        pc <= 0;
    else begin // use falling edge
        if (((op==cjp) & (jc==0)) | (op==jmp))
            pc <= imm;
        else
            pc <= pc + 1;
    end

always @(posedge clk or posedge reset)
    if (reset) // compute jump flag and store in FF
        jc <= 0;
    else
        jc <= ((op == ceq) & (s0 == s1)) |
                ((op == cne) & (s0 != s1));

// Mapping of the instruction, i.e., decode instruction
assign op = ir[11:8]; // Operation code
assign dma = ir[7:0]; // Data memory address
assign imm = ir[7:0]; // Immediate operand

prog_rom brom
( .clk(clk), .reset(reset), .address(pc), .q(pmd));
assign ir = pmd;

assign not_clk = ~clk;

data_ram bram
( .clk(not_clk), .address(dma), .q(dmd),
  .data(s0), .we(mem_ena));

always @(posedge clk or posedge reset)
begin : P3
    integer temp;
    if (reset) begin // Asynchronous clear
        s0 <= 0; s1 <= 0; s2 <= 0; s3 <= 0;
        oport <= 0;
    end else begin
        case (op)
            add : s0 <= s0 + s1;

```

```

        neg    :    s0  <= -s0;
        sub    :    s0  <= s1 - s0;
        opand  :    s0  <= s0 & s1;
        opor   :    s0  <= s0 | s1;
        inv    :    s0  <= ~ s0;
        mul    :    begin temp = s0 * s1;  // double width
                     s0  <= temp[WD:0]; end  // product
        pop    :    s0  <= s1;
        push   :    s0  <= dmd;
        pushi  :    s0  <= imm;
        scan   :    s0  <= iport;
        print  :    begin oport <= s0; s0<=s1; end
        default:    s0  <= 0;
    endcase
    case (op) // Specify the stack operations
        pushi, push, scan : begin s3<=s2;
                               s2<=s1; s1<=s0; end           // Push type
        cjp, jmp,  inv | neg : ;    // Do nothing for branch
        default :    begin s1<=s2; s2<=s3; s3<=0; end
                               // Pop all others

    endcase
end
end

// Extra test pins:
assign dmd_out = dmd; assign dma_out = dma; //Data memory
assign dma_in = dma; assign dmd_in  = s0;
assign pc_out = pc;  assign ir_imm = imm;
assign op_code = op;  // Program control
// Control signals:
assign jc_out = jc; assign me_ena = mem_ena;
// Two top stack elements:
assign s0_out = s0; assign s1_out = s1;

endmodule

```

Appendix B. Design Examples Synthesis Results

The synthesis results for all examples can be easily reproduced for the Quartus version installed on your computer by using the script `qvhdl.tcl` for VHDL or `qv.tcl` for Verilog available in the source code directories of the CD-ROM. Run the VHDL TCL script with

```
quartus_sh -t qvhdl.tcl
```

to compile all designs. The next step is to run the resource and timing analysis with

```
quartus_sta -t fmax4all.tcl
```

The script produces four parameters for each design. For the `trisc0.vhd`, for instance, we get the following:

```
....
-----
trisc0 (Clock clk) : Fmax = 92.66 (Restricted Fmax = 92.66)
trisc0 LEs: 171 / 114,480 ( < 1 % )
trisc0 M9K bits: 256 / 3,981,312 ( < 1 % )
trisc0 9-bit DSP blocks: 1 / 532 ( < 1 % )
-----
....
```

then use a utility like `grep` through the report `qvhdl.txt` file using `Fmax`, `LEs`: etc.

From the script you will notice that the following special options of Quartus II web edition 12.1 were used:

- Device set Family to Cyclone IV E and then under Available devices select EP4CE115F29C7.
- For Timing Analysis Settings set Default required fmax: to 1ns.
- For Analysis & Synthesis Settings from the Assignments menu
 - set Optimization Technique to Speed
 - Deselect Power-Up Don't Care
- In the Fitter Settings select as Fitter effort Standard Fit (highest effort)

The table below displays the results for all VHDL and Verilog examples given in this book. The table is structured as follows. The first column shows the entity or module name of the design. Columns 2 to 6 are data for the VHDL designs: the number of LEs shown in the report file; the number of 9×9 -bit multipliers; the number of M9K memory blocks; the registered performance **Fmax** using the **TimeQuest** slow 85C model; and the page with the source code. The same data are provided for the Verilog design examples, shown in columns 7 to 9. Note that VHDL and Verilog produce the same data for a number of 9×9 -bit multipliers most of the time, except for the four designs **ica** (Verilog 184 multiplier), **pca** (Verilog 138 multiplier), **iir5para** (Verilog 58 multiplier), and **iir5lwdf** (Verilog 18 multiplier). LEs and registered performance never match. The number of M9K memory blocks do not match for the three designs **fft256**, **fun_text**, and **trisc0**. In Verilog the ROM LUTs are synthesized to M9K blocks, while in VHDL LEs are used. An **fpu** design is not available in Verilog. A few designs don't use registers and a registered performance cannot be measured.

Design	LEs	9 × 9 Mult. vhd/v	VHDL M9Ks vhd/v	f_{MAX} MHz	Page	LEs	Verilog f_{MAX} MHz	Page
add1p	125	0	0	350.63	83	77	336.25	797
add2p	233	0	0	243.43	83	143	318.17	798
add3p	372	0	0	231.43	83	228	278.47	799
adpcm	531	0	0	49.5	618	510	56.0	870
ammod	264	0	0	197.39	512	222	298.78	859
arctan	106	3	0	32.71	145	105	33.05	806
cic3r32	341	0	0	282.49	321	339	280.11	831
cic3s32	209	0	0	290.02	330	206	294.2	833
cmoms	549	3	0	95.27	369	421	102.81	841
cmul7p8	48	0	0	-	63	48	-	797
cordic	276	0	0	209.6	137	172	317.97	804
dapara	39	0	0	205.17	212	39	205.17	819
dasign	52	0	0	258.4	204	39	331.56	817
db4latti	420	0	0	58.11	390	248	99.02	845
db4poly	167	0	0	618.43	310	156	554.32	829
div_aegp	45	4	0	124.91	103	44	129.28	801
div_res	106	0	0	263.5	96	89	269.25	803
dwtdden	879	0	1	120.93	406	889	164.28	847
example	33	0	0	267.24	17	32	457.67	795
farrow	363	3	0	39.82	358	268	58.25	839
fft256	34,340	8	0/2	31.12	442	33,926	31.16	854
fir4dlms	106	4	0	261.57	568	105	260.62	862
fir_gen	93	4	0	157.38	182	93	153.66	813
fir_lms	51	4	0	69.26	561	51	70.2	861
fir_srg	109	0	0	88.35	193	79	99.81	814
fpu	8112	7	0	-	120	-	-	-
fun_text	180	0	0/1	250.63	33	32	306.65	796
g711alaw	70	0	0	-	358	97	-	869
ica	2275	172/184	0	17.87	605	2091	17.84	866
iir	62	0	0	147.3	227	30	224.82	820
iir5lwdf	764	12/18	0	55.97	296	611	52.46	828
iir5para	624	51/58	0	87.69	267	513	86.72	825
iir5sfix	2474	128	0	46.99	255	2474	47.08	824
iir_par	236	0	0	479.39	247	185	430.29	822
iir_pipe	123	0	0	215.05	241	75	350.14	821
lfsr	6	0	0	944.29	495	6	944.29	858
lfsr6s3	6	0	0	931.97	497	6	931.97	858
ln	88	10	0	29.2	156	88	29.2	807
magnitude	96	0	0	119.59	167	145	107.34	812
pca	2447	180/138	0	18.46	596	1609	23.82	864
rader7	428	0	0	138.45	429	403	151.56	851
rc_sinc	880	0	0	59.53	350	847	78.52	835
reg_file	226	0	0	-	653	226	-	874
sqrtd	261	2	0	86.23	161	244	112.1	809
trisc0	171	1	1/2	92.66	701	140	85.59	875

Appendix C. VHDL and Verilog Coding Keywords

Unfortunately, today we find *two* HDL languages are popular. The US west coast and Asia prefer Verilog, while the US east coast and Europe more frequently use VHDL. For digital signal processing with FPGAs, both languages seem to be well suited, but some VHDL examples were in the past a little easier to read because of the supported signed arithmetic and multiply/divide operations in the IEEE VHDL 1076-1987 and 1076-1993 standards. This gap has disappeared with the introduction of the Verilog IEEE standard 1364-2001, as it also includes signed arithmetic. Other constraints may include personal preferences, EDA library and tool availability, data types, readability, capability, and language extensions using PLIs, as well as commercial, business and marketing issues, to name just a few. A detailed comparison can be found in the book by Smith [3]. Tool providers acknowledge today that both languages need to be supported.

It is therefore a good idea to use an HDL code style that can easily be translated into either language. An important rule is to avoid any keyword in *both* languages in the HDL code when naming variables, labels, constants, user types, etc. The IEEE standard VHDL 1076-1993 uses 97 keywords (see VHDL 1076-1993 Language Reference Manual (LRM) on p. 179) and an extra 18 keywords are used in VHDL 1076-2008 (see VHDL 1076-2008 Language Reference Manual (LRM) on p. 236). New in VHDL 1076-2008 are:

ASSUME, ASSUME_GUARANTEE, CONTEXT, COVER, DEFAULT, FAIRNESS,
FORCE, PARAMETER, PROPERTY, PROTECTED, RELEASE, RESTRICT,
RESTRICT_GUARANTEE, SEQUENCE, STRONG, VMODE, VPROP, VUNIT

which are in version Quartus 12.1 *not* highlighted in blue in the editor but may be recognized in later Quartus versions. The IEEE standard Verilog 1364-1995, on the other hand, has 102 keywords (see LRM, p. 604). New in Verilog 1076-2001 are:

automatic, cell, config, design, endconfig, endgenerate,
generate, genvar, incdir, include, instance, liblist,
library, localparam, noshowcancelled, pulsestyle_onevent,
pulsestyle_ondetect, showcancelled, signed, unsigned, use

Together, both HDL languages (Verilog 1364-2001 and VHDL 1076-2008) have 215 keywords, including 23 in common. The Table below shows VHDL

1076-2008 keywords in capital letters, Verilog 1364-2001 keywords in small letters, and the common keywords with a capital first letter.

Table Appendix C:1. VHDL 1076-1993 and Verilog 1364-2001 keywords

ABS ACCESS AFTER ALIAS ALL always And ARCHITECTURE ARRAY ASSERT
 assign ASSUME ASSUME_GUARANTEE ATTRIBUTE automatic Begin BLOCK
 BODY buf BUFFER bufif0 bufif1 BUS Case casex casez cell cmos
 config COMPONENT CONFIGURATION CONSTANT CONTEXT COVER deassign
 Default defparam design disable DISCONNECT DOWNT0 edge Else
 ELSIF End endcase endconfig endfunction endgenerate endmodule
 endprimitive endspecify endtable endtask ENTITY event EXIT
 FAIRNESS FILE For Force forever fork Function Generate GENERIC
 genvar GROUP GUARDED highz0 highz1 If ifnone IMPURE IN incdir
 include INERTIAL initial Inout input instance integer IS join
 LABEL large liblist Library LINKAGE LITERAL LOOP localparam
 macromodule MAP medium MOD module Nand negedge NEW NEXT nmos Nor
 noshowcancelled Not notif0 notif1 NULL OF ON OPEN Or OTHERS OUT
 output PACKAGE Parameter pmos PORT posedge POSTPONED primitive
 PROCEDURE PROCESS PROPERTY PROTECTED pull0 pull1 pulldown pullup
 pulsestyle_onevent pulsestyle_ondetect PURE RANGE rcmos real
 realtime RECORD reg REGISTER REJECT Release REM repeat REPORT
 RESTRICT RESTRICT_GUARANTEE RETURN rnmos ROL ROR rpmos rtran
 rtranif0 rtranif1 scaled SELECT SEQUENCE SEVERITY SHARED
 showcancelled SIGNAL signed OF SLA SLL small specify specparam
 SRA SRL STRONG strong0 strong1 SUBTYPE supply0 supply1 table task
 THEN time TO tran tranif0 tranif1 TRANSPORT tri tri0 tri1 triand
 trior trireg TYPE UNAFFECTED UNITS unsigned UNTIL Use VARIABLE
 VMODE VPROP VUNIT vectored Wait wand weak0 weak1 WHEN While wire
 WITH wor Xnor Xor

Appendix D. CD-ROM Content

The accompanying CD-ROM includes:

- All VHDL/Verilog design examples and scripts to compile
- Utility programs and files

To install the Quartus II 12.1 web edition software first go to Altera's webpage www.altera.com and click on **Design Tools & Services** and select **Design Software**. Select the web edition unless you have a full subscription. Download the software including the free ModelSim-Altera package.

Altera frequently update the Quartus II software to support new devices and other improvements and you may consider downloading the latest Quartus II version from the Altera webpage directly, but keep in mind that the files are large and that the synthesis results will differ slightly for another version than 12.1 used for the book.

The design examples for the book are located in the directories **vhd1** and **verilog** for the VHDL and Verilog examples, respectively. These directories contain, for each example, the following files:

- The VHDL or Verilog source code (*.vhd and *.v)
- The Quartus project files (*.qpf)
- The Quartus setting files (*.qsf)
- The ModelSim simulator stimuli script (*.do)
- The files for timing simulation (*.vho and *.vo)

To simplify the compilation and postprocessing, the source code directories include the additional (*.bat) files and Tcl scripts shown below:

File	Comment
<code>qvhdl.tcl</code> or <code>qv.tcl</code>	Tcl script to compile all design examples. Note that the device can be changed from Cyclone IV to Flex, Apex or Stratix just by changing the comment sign # in column 1 of the script.
<code>fmax4all.bat</code>	Script to compute the used resources and the maximum performance of the designs.
<code>qclean.bat</code>	Cleans all temporary Quartus II compiler files, but not the report files (<code>*.map.rpt</code>), and the project files <code>*.qpf</code> and <code>*.qsf</code> .
<code>qveryclean.bat</code>	Cleans all temporary compiler files, <i>including</i> all report files (<code>*.rpt</code>) and project files.

Use the DOS prompt and type

```
quartus_sh -t qvhdl.tcl
```

to compile all design examples and then

```
quartus_sta -t fmax4all.tcl
```

to determine the performance and resources. Then run the `qclean.bat` to remove the unnecessary files. The Tcl script language developed by the Berkeley Professor John Ousterhout [442–444] (used by most modern CAD tools: Altera Quartus, Xilinx ISE, ModelTech, etc.) allows a comfortable scripting language to define setting, specify functions, etc. Given the fact that many tools also use the graphic toolbox Tcl/Tk we have witnessed that many tools now also look almost the same.

The script includes all settings and also alternative device definitions. The script produces four parameters for each design. For the `trisc0.vhd`, for instance, we get:

```
....
-----
trisc0 (Clock clk) : Fmax = 92.66 (Restricted Fmax = 92.66)
trisc0 LEs: 171 / 114,480 ( < 1 % )
trisc0 M9K bits: 256 / 3,981,312 ( < 1 % )
trisc0 9-bit DSP blocks: 1 / 532 ( < 1 % )
-----
....
```

The results for all examples are summarized in Appendix B Table p. 881.

Other devices are specified in the script and include:

- EPF10K20RC240-4 from the UP1 University board
- EPF10K70RC240-4 from the UP2 University board
- EP20K200EFC484-2X from the Nios development board

- EP2C35F672C6 from the DE2 University board
- EP4SGX230 from the DE4 University board
- EP1S10F484C5, EP1S25F780C5, and EP2S60F1020C4ES from other DSP boards available from Altera

For the simulation stimuli `*.do` files are provided for the `ModelSim` simulator that are almost identical for the VHDL and Verilog projects. A simulation file example is shown in Chap. 1, p. 38. Both use a `tb_ini.do` initialization file that compiles the source code and provides an `add_local` function that allows one to add additional signals in the functional simulation that may not be available in timing simulation. For the `fun_text` project, for instance, we use `do fun_text.do 0` for RTL and `do fun_text.do 1` for timing simulation in the `ModelSim` simulator transcript window. Timing simulation requires a full compilation first and the output files `*.vo` or `*.vho` must be placed in the source code directory.

Utility Programs and Files

A couple of extra utility programs are also included on the CD-ROM¹ and can be found in the directory `util`:

File	Description
<code>sine.exe</code>	Program to generate the VHDL and Verilog sine for the function generator in Chap. 1
<code>csd.exe</code>	Program to find the canonical signed digit representation of integers or fractions as used in Chap. 2
<code>fp_ops.exe</code>	Program to compute the floating-point test data used in Chap. 2
<code>dagen.exe</code>	Program to generate the VHDL code for the distributed arithmetic files used in Chap. 3
<code>ragopt.exe</code>	Program to compute the reduced adder graph for constant-coefficient filters as used in Chap. 3. It has ten predefined lowpass and half-band filters. The program uses a MAG cost table stored in the file <code>mag14.dat</code>
<code>cic.exe</code>	Program to compute the parameters for a CIC filter as used in Chap. 5

The programs are compiled using the author's MS Visual C++ standard edition software (available for \$50–100 at all major retailers) for DOS window

¹ You need to copy the programs to your hard-drive or memory stick first; you cannot start them from the CD directly since the programs write out the results in text files.

applications and should therefore run on Windows 95 or higher. The DOS script `Testall.bat` produces the examples used in the book.

Also under `util` we find the following utility files:

File	Description
<code>quickver.pdf</code>	Quick reference card for Verilog HDL from QUALIS
<code>quickvhdl.pdf</code>	Quick reference card for VHDL from QUALIS
<code>quicklog.pdf</code>	Quick reference card for the IEEE 1164 logic package from QUALIS
<code>93vhdl.vhd</code>	The IEEE VHDL 1076-1993 keywords
<code>2008vhdl.vhd</code>	The IEEE VHDL 1076-2008 keywords
<code>95key.v</code>	The IEEE Verilog 1364-1995 keywords
<code>01key.v</code>	The IEEE Verilog 1364-2001 keywords
<code>95direct.v</code>	The IEEE Verilog 1364-1995 compiler directives
<code>95tasks.v</code>	The IEEE Verilog 1364-1995 system tasks and functions

In addition, the CD-ROM includes a collection of useful Internet links (see file `dsp4fpga.htm` under `util`), such as device vendors, software tools, VHDL and Verilog resources, and links to online available HDL introductions, e.g., the “Verilog Handbook” by Dr. D. Hyde and “The VHDL Handbook Cookbook” by Dr. P. Ashenden.

(L)WDF Filter Toolbox

The (L)WDF toolbox written by Lincklaen Arriens can be found in the `lwdf` folder. There are also two PDF manuals that help you get started:

- `WDF_toolbox_UG_v1_0.pdf` is the (L)WDF Toolbox MATLAB Users Guide that includes tutorial to design the (L)WDF filters
- `WDF_toolbox_RG_v1_0.pdf` is the (L)WDF Toolbox MATLAB reference guide that includes a description of the available functions

These files are used in Chap. 4 to design WDF and LWDF narrow band IIR filters.

Compressed Sound Data

Under `sound` we find the following speech data files used in Chap. 8:

File	Description
<code>Speech_PCM16bit.wav</code>	Original speech data in 16-bit precision (no compression)
<code>Speech_PCM8bit.wav</code>	Original speech data in 8-bit precision (no compression)
<code>Speech_PCM4bit.wav</code>	Original speech data in 4-bit precision (no compression)
<code>Speech_A_LAW8bit.wav</code>	Compressed speech data using 8 bits per sample A-law compression
<code>Speech_PCM4Lloyd.wav</code>	Compressed speech data using 4 bits per sample Lloyd optimal quantizer
<code>Speech_ADPCM4bit.wav</code>	Compressed speech data using 4 bits per sample ADPCM method

Microprocessor Project Files and Programs

All microprocessor-related tools and documents can be found in the `uP` folder. Six software Flex/Bison projects along with their compiler scripts are included:

- `build1.bat` and `simple.l` are used for a simple Flex example.
- `build2.bat`, `d_ff.vhd`, and `vhdlcheck.l` are a basic VHDL lexical analysis.
- `build3.bat`, `asm2mif.l`, and `add2.txt` are used for a simple Flex example.
- `build4.bat`, `add2.y`, and `add2.txt` are used for a simple Bison example.
- `build5.bat`, `calc.l`, `calc.y`, and `calc.txt` are infix calculators and are used to demonstrate the Bison/Flex communication.
- `build6.bat`, `c2asm.h`, `c2asm.h`, `c2asm.c`, `lc2asm.c`, `yc2asm.c`, and `factorial.c` are used for a C-to-assembler compiler for a stack computer.

The `*.txt` files are used as input files for the programs. The `buildx.bat` can be used to compile each project separately; alternatively you can use the `uPrunall.bat` under Unix to compile and run all files in one step. The compiled files that run under SunOS UNIX end with `*.exe` while the DOS programs end with `*.com`.

Here is a short description of the other supporting files in the `uP` directory: **Bison.pdf** contains the Bison compiler, i.e., the YACC-compatible parser generator, written by Charles Donnelly and Richard Stallman; **Flex.pdf** is the description of the fast scanner generator written by Vern Paxson.

VGA Project Files

To get started with the VGA project, connect the DE2 board to your PC via USB cable, add a power supply for the board and the VGA cable to a VGA monitor. After you turn on the power you should see the test picture of the DE2 board on the VGA monitor. This is the startup configuration that is factory programmed in the DE2's E²ROM. Now copy the whole directory `DE2_115_ImageVGA` from the CD to your PC, memory stick, or network drive, and you are ready to start the Quartus software and load the project or double click `DE2_115_ImageVGA.qpf`. You can download the project `DE2_115_ImageVGA.sof` to the board to become familiar with the project and the switches. Observe the VGA display, LCD and the eight seven segment displays. Try to change the MSB and the LSB of the slider switches and observe the changes in the VGA display.

If you want to modify the project here is a brief description of the major files of the projects:

- `DE2_115_ImageVGA.v` is the top level design file that includes the edge detection filter, instantiation of the image memory, and connections to the I/O pins. Since the project does not use the Nios II processor no `Qsys` file is required. All design files including the driver for the I/O are already included in the source code directory.
- `VGA_wave.do` is the script to run the `ModelSim` simulation. Remember that we use the timing simulation since the loading of the large MIF file in functional simulation has excessive memory requirements. Make a full compile first before you start the simulation. Start `ModelSim` and then run the script with `do VGA_wave.do 1`. The parameter "1" means timing simulation.
- If you like to test other images you need to do two things. First you need to convert your 640×480 BMP image using the MATLAB script `bm2txt.m` or the factory provided `PrintNum.exe` to an MIF file. You can find these files in the `VGA_DATA` directory of the project. There are also several other test picture you can try. For the second step, in Quartus start the MegaWizard, load/edit the megafunction file `img_data.v` and browse to the new MIF file. Then recompile the whole project and download the SOF file to the board.
- `qclean.bat` cleans the temporary files, but not the SOF file. In addition you should delete the `db` directory before moving the project to another location.

After you make any changes to the project make sure to recompile the whole project before downloading the SOF file to the board. This project has no `Qsys` files and a Nios II system generation is therefore unnecessary.

Image Processing Project Files

To get started with the median filter Nios II software project, connect the DE2 to your PC via USB cable, add a power supply for the board and the VGA cable to a VGA monitor. After you turn on the power you should see the test picture of the DE2 board on the VGA monitor. This is the startup configuration that is factory programmed in the DE2's E²ROM. Now copy the whole directory `DE2_115_ImageProcessing` from the CD to your PC, memory stick, or network drive, and you are ready to start the Quartus software and load the project or double click `DE2_115_ImageProcessing.qpf`. You should then download the project `DE2_115_ImageProcessing.sof` to the board. The VGA display will show random black-and-white pixels. Next start the **Nios II Software Build Tools for Eclipse**. You may try to import the whole software project from the software/median folder; however we found that most often paths or files could not be found in the right location. It is usually more successful if you start with a "Hello World" project and then copy the required files in this hello world project. Therefore, we recommend that within Eclipse you generate a "Hello World" project. If you run it as Nios hardware it will give out in the terminal window the message "Hello from Nios II!" After you have successfully run this program replace the `hello_world.c` file with the `median.c` file and copy the file `Qpicture.mif` into the same directory in which you have the hello world project files. You find the required new source files in the `c-source` subdirectory. Now you need to enable the host file system support. Right click `median_bsp` in **Project Explorer** and Start then **Nios II → BSP Editor . . .**. In **BSP Editor** select **Software Packages** and Enable `altera_hostfs`. Finally click the **Generate** button. Then right click the project in the **Project Explore** window and select **Debug As → Nios II Hardware**. Then the project will be downloaded to the board and the debug window opens. Then press the **Run** button or F8. The image is transferred to the board, noise is added, and horizontal and vertical filters are applied. The LEDs indicate the status of the steps. The slider switches (SWs) are used as follows: SW7..SW0 is used as threshold value, SW17 as on/off to save the file with the current image to a text file on the host system, and SW16-SW14 are used to specify the median filter length. The minimum length is 3. The edge detection run in a forever loop so that you can try different thresholds and filter lengths without the need to transfer the image again. Since the image transfer goes over the JTAG cable the time taken is substantial.

If you want to modify the project here is a brief description of the major files of the projects:

- If you like to test other images you need to convert your 320×240 BMP image using the **MatLab** script `bm2txt.m` or the factory provided `PrintNum.exe` to an MIF file. You find these files in the `c-source` subdi-

rectory. You do not need to recompile the Quartus design if you just change the image; you can use the SOF file provided.

- **median.c** is the program that includes all median filtering, adding of S&P noise, and file I/O. SW and LEDs are used too. Make sure the correct base address is used if you make changes to the hardware files. You do not need to recompile the Quartus design if you only change the software and you can use the SOF file provided.
- **DE2_115_ImageProcessing.qpf** is the top level project file that includes the Qsys Nios system as well as all connections to the I/O pins. Making modification to the Qsys design should only be done by an experienced Qsys user. As a minimum you should have completed the Qsys tutorial: “Introduction to the Altera Qsys System Integration Tool” from the University Program tutorials. You also need to download and install the University program IP cores that come with the free “University Program Installer.” The IP version must match the Quartus version you are using. After successful installation the IP blocks should appear in the Qsys component library as shown in Fig. 10.19, p. 778 on the left. Only after successful installation of the IP blocks will you be able to make modifications to the Qsys file. If you want to use another board make sure you include the correct pin file and make the required correction to the top level VHDL file **DE2_115_ImageProcessing.vhd**.

After you make any changes to the Qsys project make sure to generate the new Nios II system and recompile the whole project before downloading the SOF file to the board.

Custom Instruction Computer Project Files

The custom instruction computer project supports the CI for the bit swap operation found in Chap. 9 and the CI to improve the motion estimation in Chap. 10. The Quartus design is based on the Basic Computer system provided by Altera’s University Program. To get started with the CI project connect the DE2 to your PC via USB cable and add a power supply for the board. No VGA cable or VGA monitor is required for this project. Now copy the whole directory **DE2_115_CI_Computer** from the CD to your PC, memory stick, or network drive, and you are ready to start the Quartus software and load the project or double click **DE2_115_CI_Computer.qpf**. You should then download the project **DE2_115_CI_Computer.sof** to the board. You may try to import the whole software project from the **software/Motion** folder; however we found that most often pathes or files could not be found in the right location. It is usually more successful if you start with a “Hello World” project and then copy the required files in this hello world project. Therefore, we recommend that you within Eclipse generate a “Hello World” source code project, but name it project **Motion**. Right click the project and select **Run**

As \rightarrow Nios II Hardware and it will produce in the terminal window the message “Hello from Nios II!” After you have successfully run this program replace the `hello_world.c` file with the project file you want to run. You can find these files in the `c-source` subdirectory. If you want to modify the project here is a brief description of the major files of the project:

- The software programs in the `c-source` subdirectory are: the `my_swap.c` file used in Chap. 9 for the three versions of the bit swap operation; the `madtest.c` file that has a brief check of byte access used for the MAD computation; the `motion.c` program that generates two test images and computes the motion vectors and measures the run time. You do not need to recompile the Quartus II or generate a Qsys system if you change only the software and you can use the provided SOF file.
- `DE2_115_CIProcessor.qpf` is the top level project file that includes the Qsys Nios system as well as all connections to the I/O pins. Making modification to the Qsys design should only be done by an experienced Qsys user. As a minimum you should have completed the Qsys tutorial: “Introduction to the Altera Qsys System Integration Tool” and “Making Qsys Components” from the University Program tutorials. You also need to download and install the University program IP cores that come with the free “University Program Installer.” The IP version must match the Quartus version you are using. After successful installation the IP blocks should appear in the Qsys component library as shown in Fig. 10.19, p. 778 on the left. Only after successful installation of the IP blocks will you be able to make modifications to the Qsys file. If you want to use another board make sure you include the correct pin file and make the required correction to the top level VHDL file `DE2_115_ImageProcessing.vhd`. The TCL scripts and the VHDL source code for the CI files can be found under `nios_system` \rightarrow `synthesis` \rightarrow `submodules`.

After you make any changes to the Qsys project make sure to generate the new Nios II system and recompile the whole project before downloading the SOF file to the board.

Appendix E. Glossary

ACC	Accumulator
ACT	Actel FPGA family
ADC	Analog-to-digital converter
ADCL	All-digital CL
ADF	Adaptive digital filter
ADPCM	Adaptive differential pulse code modulation
ADPLL	All-digital PLL
ADSP	Analog Devices digital signal processor family
AES	Advanced encryption standard
AFT	Arithmetic Fourier transform
AHDL	Altera HDL
AHSM	Additive half square multiplier
ALM	Adaptive logic module
ALU	Arithmetic logic unit
AM	Amplitude modulation
AMBA	Advanced microprocessor bus architecture
AMD	Advanced Micro Devices, Inc.
AMUSE	Algorithm for multiple unknown signals extraction
APEX	Adaptive principal component extraction
ASCII	American Standard Code for Information Interchange
ASIC	Application specific IC
AWGN	Additive white Gaussian noise
BCD	Binary coded decimal
BDD	Binary decision diagram
BIT	Binary digit
BLMS	Block LMS
BMP	Bitmap
BP	Bandpass
BRAM	Block RAM
BRS	Base removal scaling
BS	Barrelshifter
BSS	Blind source separation

CAD	Computer-aided design
CAE	Computer-aided engineering
CAM	Content addressable memory
CAST	Carlisle Adams and Stafford Tavares
CBC	Cipher block chaining
CBIC	Cell-based IC
CCD	Charge-coupled device
CCITT	Comité consultatif international téléphonique et télégraphique
CD	Compact disc
CFA	Common factor algorithm
CFB	Cipher feedback
CHF	Swiss franc
CIC	Cascaded integrator comb
CIF	Common intermediate format
CISC	Complex instruction set computer
CL	Costas loop
CLB	Configurable logic block
C-MOMS	Causal MOMS
CMOS	Complementary metal oxide semiconductor
CODEC	Coder/decoder
CORDIC	Coordinate rotation digital computer
COTS	Commercial off-the-shelf technology
CPLD	Complex PLD
CPU	Central processing unit
CQF	Conjugate quadrature filter
CRNS	Complex RNS
CRT	Chinese remainder theorem
CRT	Cathode ray tube
CSE	Common sub-expression
CSOC	Canonical self-orthogonal code
CSD	Canonical signed digit
CWT	Continuous wavelet transform
CZT	Chirp- z transform
DA	Distributed arithmetic
DAC	Digital-to-analog converter
DAT	Digital audio tap
DB	Daubechies filter
DC	Direct current
DCO	Digital controlled oscillator
DCT	Discrete cosine transform
DCU	Data cache unit
DDRAM	Double data rate RAM
DES	Data encryption standard
DFT	Discrete Fourier transform

DHT	Discrete Hartley transform
DIF	Decimation in frequency
DIT	Decimation in time
DLMS	Delayed LMS
DM	Delta modulation
DMA	Direct memory access
DMIPS	Dhrystone MIPS
DMT	Discrete Morlet transform
DOD	Department of defence
DPCM	Differential PCM
DPLL	Digital PLL
DSP	Digital signal processing
DST	Discrete sine transform
DWT	Discrete wavelet transform
EAB	Embedded array block
EASI	Equivariant adaptive separation via independence
ECB	Electronic code book
ECG	Electrocardiography
ECL	Emitter coupled logic
EDA	Electronic design automation
EDIF	Electronic design interchange format
EFF	Electronic Frontier Foundation
EOB	End of block
EPF	Altera FPGA family
EPROM	Electrically programmable ROM
ERA	Plessey FPGA family
ERNS	Eisenstein RNS
ESA	European Space Agency
ESB	Embedded system block
EVR	Eigenvalue ratio
EXU	Execution unit
FAEST	Fast a posteriori error sequential technique
FCT	Fast Cosine transform
FC2	FPGA compiler II
FF	Flip-flop
FFT	Fast Fourier transform
FIFO	First-in first-out
FIR	Finite impulse response
FIT	Fused internal timer
FLEX	Altera FPGA family
FM	Frequency modulation
FNT	Fermat NTT
FPGA	Field-programmable gate array

FPL	Field-programmable logic (combines CPLD and FPGA)
FPLD	FPL device
FPMAC	Floating-point MAC
FPS	Frames per second
FSF	Frequency sampling filter
FSK	Frequency shift keying
FSM	Finite state machine
GAL	Generic array logic
GF	Galois field
GFPMACS	Giga FPMAC
GIF	Graphic interchange format
GNU	GNU's not Unix
GPP	General purpose processor
GPR	General purpose register
HB	Half-band filter
HDL	Hardware description language
HDMI	High definition multimedia interface
HDTV	High-definition television
HI	High frequency
HP	Hewlett Packard
HSP	Harris Semiconductor DSP ICs
HW	Hardware
IBM	International Business Machines (corporation)
IC	Integrated circuit
ICA	Independent component analysis
ICU	Instruction cache unit
IDCT	Inverse DCT
IDEA	International data encryption algorithm
IDFT	Inverse discrete Fourier transform
IEC	International electrotechnical commission
IEEE	Institute of Electrical and Electronics Engineers
IF	Inter frequency
IFFT	Inverse fast Fourier transform
IIR	Infinite impulse response
IMA	Interactive multimedia association
I-MOMS	Interpolating MOMS
INTT	Inverse NTT
IP	Intellectual property
I/Q	In-/Quadrature phase
ISA	Instruction set architecture
ISDN	Integrated services digital network
ISO	International standardization organization

ITU	International Telecommunication Union
JPEG	Joint photographic experts group
JTAG	Joint test action group
KCPSM	Ken Chapman PSM
KLT	Karhunen–Loeve transform
LAB	Logic array block
LAN	Local area network
LC	Logic cell
LCD	Liquid-crystal display
LE	Logic element
LIFO	Last in first out
LISA	Language for instruction set architecture
LF	Low frequency
LFSR	Linear feedback shift register
LMS	Least-mean-square
LNS	Logarithmic number system
LO	Low frequency
LP	Low pass
LPC	Linear predictive coding
LPM	Library of parameterized modules
LRS	serial left right shifter
LS	Least-square
LSB	Least significant bit
LSI	Large scale integration
LTI	Linear time-invariant
LUT	Look-up table
LWDF	Lattice WDF
LZW	Lempel-Ziv-Welch
MAC	Multiplication and accumulate
MACH	AMD/Vantis FPGA family
MAG	Multiplier adder graph
MAX	Altera CPLD family
MIF	Memory initialization file
MIPS	Microprocessor without interlocked pipeline
MIPS	Million instructions per second
MLSE	Maximum likelihood sequence estimator
MMU	Memory management unit
MMX	Multimedia extension
MNT	Mersenne NTT
MOMS	Maximum order minimum support
μ P	Microprocessor

MPEG	Moving picture experts group
MPX	Multiplexer
MSPS	Millions of sample per second
MRC	Mixed radix conversion
MSB	Most significant bit
MUL	Multiplication
NCO	Numeric controlled oscillators
NLMS	Normalized LMS
NOF	Non-output fundamental
NP	Nonpolynomial complex problem
NRE	Nonrecurring engineering costs
NTSC	National television system committee
NTT	Number theoretic transform
OFB	Open feedback (mode)
O-MOMS	Optimal MOMS
OPAST	Orthogonal PAST
PAL	Phase alternating line
PAM	Pulse-amplitude-modulated
PAST	Projection approximation subspace tracking
PC	Personal computer
PC	Principle component
PCA	Principle component analysis
PCI	Peripheral component interconnect
PCM	Pulse-code modulation
PD	Phase detector
PDF	Probability density function
PDSP	Programmable digital signal processor
PFA	Prime factor algorithm
PIT	Programmable interval timer
PLA	Programmable logic array
PLD	Programmable logic device
PLL	Phase-locked loop
PM	Phase modulation
PNG	Portable network graphic
PPC	Power PC
PREP	Programmable Electronic Performance (cooperation)
PRNS	Polynomial RNS
PROM	Programmable ROM
PSK	Phase shift keying
PSM	Programmable state machine
QCIF	Quarter CIF

QDFT	Quantized DFT
QLI	Quick look-in
QFFT	Quantized FFT
QMF	Quadrature mirror filter
QRNS	Quadratic RNS
QSM	Quarter square multiplier
QVGA	Quarter VGA
RAG	Reduced adder graph
RAM	Random access memory
RC	Resistor/capacity
RF	Radio frequency
RGB	Red, green and blue
RISC	Reduced instruction set computer
RLS	Recursive least square
RNS	Residue number system
ROM	Read only memory
RPFA	Rader prime factor algorithm
RS	serial right shifter
RSA	Rivest, Shamir, and Adelman
SD	Signed digit
SDRAM	Synchronous dynamic RAM
SECAM	Sequential color with memory
SG	Stochastic gradient
SIMD	Single instruction multiple data
SLMS	Signed LMS
SM	Signed magnitude
SNR	Signal-to-noise ratio
SOBI	Second order blind identification
SPEC	System performance evaluation cooperation
SPLD	Simple PLD
SPT	Signed power of two
SR	Shift register
SRAM	Static random access memory
SSE	Streaming SIMD extension
STFT	Short term Fourier transform
SVGA	Super VGA
SW	Software
SXGA	Super extended graphics array
TDLMS	Transform domain LMS
TLB	Translation look-aside buffer
TLU	Table look-up
TMS	Texas Instruments DSP family

TI	Texas Instruments
TOS	Top of stack
TSMC	Taiwan semiconductor manufacturing company
TTL	Transistor transistor logic
TVP	True vector processor
UART	Universal asynchronous receiver/transmitter
USB	Universal serial bus
VCO	Voltage-control oscillator
VGA	Video graphics array
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit
VLC	Variable run-length coding
VLIW	Very long instruction word
VLSI	Very large integrated ICs
WDF	Wave digital filter
WDT	Watchdog timer
WFTA	Winograd Fourier transform algorithm
WSS	Wide sense stationary
WWW	World wide web
XC	Xilinx FPGA family
XNOR	exclusive NOR gate
YACC	Yet another compiler-compiler

References

1. B. Dipert: "EDN's first annual PLD directory," *EDN* pp. 54–84 (2000)
2. S. Brown, Z. Vranesic: *Fundamentals of Digital Logic with VHDL Design* (McGraw-Hill, New York, 1999)
3. D. Smith: *HDL Chip Design* (Doone Publications, Madison, Alabama, USA, 1996)
4. U. Meyer-Bäse: *The Use of Complex Algorithm in the Realization of Universal Sampling Receiver using FPGAs (in German)* (VDI/Springer, Düsseldorf, 1995), vol. 10, No. 404, 215 pages
5. U. Meyer-Bäse: *Fast Digital Signal Processing (in German)* (Springer, Heidelberg, 1999), 370 pages
6. P. Lapsley, J. Bier, A. Shoham, E. Lee: *DSP Processor Fundamentals* (IEEE Press, New York, 1997)
7. D. Shear: "EDN's DSP Benchmarks," *EDN* **33**, pp. 126–148 (1988)
8. V. Betz, S. Brown: "FPGA Challenges and Opportunities at 40 nm and Beyond," in *International Conference on Field Programmable Logic and Applications* Prague (2009), p. 4, fPL
9. Plessey: (1990), "Data sheet," ERA60100
10. J. Greene, E. Hamdy, S. Beal: "Antifuse Field Programmable Gate Arrays," *Proceedings of the IEEE* pp. 1042–56 (1993)
11. Lattice: (1997), "Data sheet," GAL 16V8
12. J. Rose, A. Gamal, A. Sangiovanni-Vincentelli: "Architecture of Field-Programmable Gate Arrays," *Proceedings of the IEEE* pp. 1013–29 (1993)
13. Xilinx: "PREP Benchmark Observations," in *Xilinx-Seminar* San Jose (1993)
14. Altera: "PREP Benchmarks Reveal FLEX 8000 is Biggest, MAX 7000 is Fastest," in *Altera News & Views* San Jose (1993)
15. Actel: "PREP Benchmarks Confirm Cost Effectiveness of Field Programmable Gate Arrays," in *Actel-Seminar* (1993)
16. E. Lee: "Programmable DSP Architectures: Part I," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* pp. 4–19 (1988)
17. E. Lee: "Programmable DSP Architectures: Part II," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* pp. 4–14 (1989)
18. R. Petersen, B. Hutchings: "An Assessment of the Suitability of FPGA-Based Systems for Use in Digital Signal Processing," *Lecture Notes in Computer Science* **975**, 293–302 (1995)
19. J. Villasenor, B. Hutchings: "The Flexibility of Configurable Computing," *IEEE Signal Processing Magazine* pp. 67–84 (1998)
20. Altera: (2011), "Floating-Point Megafunctions User Guide," ver. 11.1
21. Texas Instruments: (2008), "TMS320C6727B, TMS320C6726B, TMS320C6722B, TMS320C6720 Floating-Point Digital Signal Processors"
22. Xilinx: (1993), "Data book," XC2000, XC3000 and XC4000
23. Altera: (1996), "Data sheet," FLEX 10K CPLD Family

24. Altera: (2013), "Cyclone IV Device Handbook," volume 1-3
25. U. Meyer-Bäse: *Digital Signal Processing with Field Programmable Gate Arrays*, 1st edn. (Springer, Heidelberg, 2001), 422 pages
26. F. Vahid, T. Givargis: *Embedded System Design* (John Wiley & Sons, New York, 2001)
27. J. Hakewill: "Gainin Control over Silicon IP," *Communication Design* online (2000)
28. E. Castillo, U. Meyer-Baese, L. Parrilla, A. Garcia, A. Lloris: "Watermarking Strategies for RNS-Based System Intellectual Property Protection," in *Proc. of 2005 IEEE Workshop on Signal Processing Systems SiPS'05 Athens* (2005), pp. 160–165
29. O. Spaniol: *Computer Arithmetic: Logic and Design* (John Wiley & Sons, New York, 1981)
30. I. Koren: *Computer Arithmetic Algorithms* (Prentice Hall, Englewood Cliffs, New Jersey, 1993)
31. E.E. Swartzlander: *Computer Arithmetic, Vol. I* (Dowden, Hutchinson and Ross, Inc., Stroudsburg, Pennsylvania, 1980), also reprinted by IEEE Computer Society Press 1990
32. E. Swartzlander: *Computer Arithmetic, Vol. II* (IEEE Computer Society Press, Stroudsburg, Pennsylvania, 1990)
33. K. Hwang: *Computer Arithmetic: Principles, Architecture and Design* (John Wiley & Sons, New York, 1979)
34. U. Meyer-Baese: *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd edn. (Springer-Verlag, Berlin, 2007), 774 pages
35. N. Takagi, H. Yasuura, S. Yajima: "High Speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Transactions on Computers* **34**(2) (1985)
36. D. Bull, D. Horrocks: "Reduced-Complexity Digital Filtering Structures using Primitive Operations," *Electronics Letters* pp. 769–771 (1987)
37. D. Bull, D. Horrocks: "Primitive operator digital filters," *IEE Proceedings-G* **138**, 401–411 (1991)
38. A. Dempster, M. Macleod: "Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters," *IEEE Transactions on Circuits and Systems II* **42**, 569–577 (1995)
39. A. Dempster, M. Macleod: "Comments on "Minimum Number of Adders for Implementing a Multiplier and Its Application to the Design of Multiplierless Digital Filters"," *IEEE Transactions on Circuits and Systems II* **45**, 242–243 (1998)
40. F. Taylor, R. Gill, J. Joseph, J. Radke: "A 20 Bit Logarithmic Number System Processor," *IEEE Transactions on Computers* **37**(2) (1988)
41. P. Lee: "An FPGA Prototype for a Multiplierless FIR Filter Built Using the Logarithmic Number System," *Lecture Notes in Computer Science* **975**, 303–310 (1995)
42. J. Mitchell: "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers* **EC-11**, 512–517 (1962)
43. N. Szabo, R. Tanaka: *Residue Arithmetic and its Applications to Computer Technology* (McGraw-Hill, New York, 1967)
44. M. Soderstrand, W. Jenkins, G. Jullien, F. Taylor: *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*, IEEE Press Reprint Series (IEEE Press, New York, 1986)
45. U. Meyer-Bäse, A. Meyer-Bäse, J. Mellott, F. Taylor: "A Fast Modified CORDIC-Implementation of Radial Basis Neural Networks," *Journal of VLSI Signal Processing* pp. 211–218 (1998)

46. V. Hamann, M. Sprachmann: "Fast Residual Arithmetics with FPGAs," in *Proceedings of the Workshop on Design Methodologies for Microelectronics* Smolenice Castle, Slovakia (1995), pp. 253–255
47. G. Jullien: "Residue Number Scaling and Other Operations Using ROM Arrays," *IEEE Transactions on Communications* **27**, 325–336 (1978)
48. M. Griffin, M. Sousa, F. Taylor: "Efficient Scaling in the Residue Number System," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1989), pp. 1075–1078
49. G. Zelniker, F. Taylor: "A Reduced-Complexity Finite Field ALU," *IEEE Transactions on Circuits and Systems* **38**(12), 1571–1573 (1991)
50. IEEE: "Standard for Binary Floating-Point Arithmetic," *IEEE Std 754-1985* pp. 1–14 (1985)
51. IEEE: "Standard for Binary Floating-Point Arithmetic," *IEEE Std 754-2008* pp. 1–70 (2008)
52. N. Shirazi, P. Athanas, A. Abbott: "Implementation of a 2-D Fast Fourier Transform on an FPGA-Based Custom Computing Machine," *Lecture Notes in Computer Science* **975**, 282–292 (1995)
53. Xilinx: "Using the Dedicated Carry Logic in XC4000E," in *Xilinx Application Note XAPP 013* San Jose (1996)
54. M. Bayoumi, G. Jullien, W. Miller: "A VLSI Implementation of Residue Adders," *IEEE Transactions on Circuits and Systems* pp. 284–288 (1987)
55. A. Garcia, U. Meyer-Bäse, F. Taylor: "Pipelined Hogenauer CIC Filters using Field-Programmable Logic and Residue Number System," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* Vol. 5 (1998), pp. 3085–3088
56. L. Turner, P. Graumann, S. Gibb: "Bit-serial FIR Filters with CSD Coefficients for FPGAs," *Lecture Notes in Computer Science* **975**, 311–320 (1995)
57. J. Logan: "A Square-Summing, High-Speed Multiplier," *Computer Design* pp. 67–70 (1971)
58. Leibowitz: "A Simplified Binary Arithmetic for the Fermat Number Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* **24**, 356–359 (1976)
59. T. Chen: "A Binary Multiplication Scheme Based on Squaring," *IEEE Transactions on Computers* pp. 678–680 (1971)
60. E. Johnson: "A Digital Quarter Square Multiplier," *IEEE Transactions on Computers* pp. 258–260 (1980)
61. Altera: (2004), "Implementing Multipliers in FPGA Devices," application note 306, Ver. 3.0
62. D. Anderson, J. Earle, R. Goldschmidt, D. Powers: "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Journal of Research and Development* **11**, 34–53 (1967)
63. V. Pedroni: *Circuit Design and Simulation with VHDL* (The MIT Press, Cambridge, Massachusetts, 2010)
64. A. Rushton: *VHDL for logic Synthesis*, 3rd edn. (John Wiley & Sons, New York, 2011)
65. P. Ashenden: *The Designer's Guide to VHDL*, 3rd edn. (Morgan Kaufman Publishers, Inc., San Mateo, CA, 2008)
66. A. Croisier, D. Esteban, M. Levilion, V. Rizo: (1973), "Digital Filter for PCM Encoded Signals," US patent no. 3777130
67. A. Peled, B. Liu: "A New Realization of Digital Filters," *IEEE Transactions on Acoustics, Speech and Signal Processing* **22**(6), 456–462 (1974)
68. K. Yiu: "On Sign-Bit Assignment for a Vector Multiplier," *Proceedings of the IEEE* **64**, 372–373 (1976)

69. K. Kammeyer: "Quantization Error on the Distributed Arithmetic," *IEEE Transactions on Circuits and Systems* **24**(12), 681–689 (1981)
70. F. Taylor: "An Analysis of the Distributed-Arithmetic Digital Filter," *IEEE Transactions on Acoustics, Speech and Signal Processing* **35**(5), 1165–1170 (1986)
71. S. White: "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* pp. 4–19 (1989)
72. K. Kammeyer: "Digital Filter Realization in Distributed Arithmetic," in *Proc. European Conf. on Circuit Theory and Design* (1976), Genoa, Italy
73. F. Taylor: *Digital Filter Design Handbook* (Marcel Dekker, New York, 1983)
74. H. Nussbaumer: *Fast Fourier Transform and Convolution Algorithms* (Springer, Heidelberg, 1990)
75. H. Schmid: *Decimal Computation* (John Wiley & Sons, New York, 1974)
76. Y. Hu: "CORDIC-Based VLSI Architectures for Digital Signal Processing," *IEEE Signal Processing Magazine* pp. 16–35 (1992)
77. U. Meyer-Bäse, A. Meyer-Bäse, W. Hilberg: "**CO**ordinate **R**otation **DI**gital Computer (CORDIC) Synthesis for FPGA," *Lecture Notes in Computer Science* **849**, 397–408 (1994)
78. J.E. Volder: "The CORDIC Trigonometric computing technique," *IRE Transactions on Electronics Computers* **8**(3), 330–4 (1959)
79. J. Walther: "A Unified algorithm for elementary functions," *Spring Joint Computer Conference* pp. 379–385 (1971)
80. X. Hu, R. Huber, S. Bass: "Expanding the Range of Convergence of the CORDIC Algorithm," *IEEE Transactions on Computers* **40**(1), 13–21 (1991)
81. D. Timmermann (1990): "CORDIC-Algorithmen, Architekturen und monolithische Realisierungen mit Anwendungen in der Bildverarbeitung," Ph.D. thesis, VDI Press, Serie 10, No. 152
82. H. Hahn (1991): "Untersuchung und Integration von Berechnungsverfahren elementarer Funktionen auf CORDIC-Basis mit Anwendungen in der adaptiven Signalverarbeitung," Ph.D. thesis, VDI Press, Serie 9, No. 125
83. G. Ma (1989): "A Systolic Distributed Arithmetic Computing Machine for Digital Signal Processing and Linear Algebra Applications," Ph.D. thesis, University of Florida, Gainesville
84. Y.H. Hu: "The Quantization Effects of the CORDIC-Algorithm," *IEEE Transactions on signal processing* pp. 834–844 (1992)
85. M. Abramowitz, A. Stegun: *Handbook of Mathematical Functions*, 9th edn. (Dover Publications, Inc., New York, 1970)
86. W. Press, W. Teukolsky, W. Vetterling, B. Flannery: *Numerical Recipes in C*, 2nd edn. (Cambridge University Press, Cambridge, 1992)
87. Intersil: (2001), "Data sheet," HSP50110
88. A.V. Oppenheim, R.W. Schaffer: *Discrete-Time Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1992)
89. D.J. Goodman, M.J. Carey: "Nine Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing* pp. 121–126 (1977)
90. U. Meyer-Bäse, J. Chen, C. Chang, A. Dempster: "A Comparison of Pipelined RAG-n and DA FPGA-Based Multiplierless Filters," in *IEEE Asia Pacific Conference on Circuits and Systems* (2006), pp. 1555–1558
91. O. Gustafsson, A. Dempster, L. Wanhammar: "Extended Results for Minimum-Adder Constant Integer Multipliers," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* Phoenix (2002), pp. 73–76

92. Y. Wang, K. Roy: "CSDC: A New Complexity Reduction Technique for Multiplierless Implementation of Digital FIR Filters," *IEEE Transactions on Circuits and Systems I* **52**(0), 1845–1852 (2005)
93. H. Samueli: "An Improved Search Algorithm for the Design of Multiplierless FIR Filters with Powers-of-Two Coefficients," *IEEE Transactions on Circuits and Systems* **36**(7), 10441047 (1989)
94. Y. Lim, S. Parker: "Discrete Coefficient FIR Digital Filter Design Based Upon an LMS Criteria," *IEEE Transactions on Circuits and Systems* **36**(10), 723–739 (1983)
95. Altera: (2013), "FIR Compiler: MegaCore Function User Guide," ver. 12.1
96. U. Meyer-Baese, G. Botella, D. Romero, M. Kumm: "Optimization of high speed pipelining in FPGA-based FIR filter design using genetic algorithm," in *Proc. SPIE Int. Soc. Opt. Eng., Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering X* (2012), pp. 84010R1–12, vol. 8401
97. R. Hartley: "Subexpression Sharing in Filters Using Canonic Signed Digital Multiplier," *IEEE Transactions on Circuits and Systems II* **30**(10), 677–688 (1996)
98. S. Mirzaei, R. Kastner, A. Hosangadi: "Layout aware optimization of high speed fixed coefficient FIR filters for FPGAs," *International Journal of Reconfigurable Computing* **2010**(3), 1–17 (2010)
99. R. Saal: *Handbook of filter design* (AEG-Telefunken, Frankfurt, Germany, 1979)
100. C. Barnes, A. Fam: "Minimum Norm Recursive Digital Filters that Are Free of Overflow Limit Cycles," *IEEE Transactions on Circuits and Systems* pp. 569–574 (1977)
101. A. Fettweis: "Wave Digital Filters: Theorie and Practice," *Proceedings of the IEEE* pp. 270–327 (1986)
102. R. Crochiere, A. Oppenheim: "Analysis of Linear Digital Networks," *Proceedings of the IEEE* **63**(4), 581–595 (1975)
103. A. Dempster, M. Macleod: "Multiplier blocks and complexity of IIR structures," *Electronics Letters* **30**(22), 1841–1842 (1994)
104. A. Dempster, M. Macleod: "IIR Digital Filter Design Using Minimum Adder Multiplier Blocks," *IEEE Transactions on Circuits and Systems II* **45**, 761–763 (1998)
105. A. Dempster, M. Macleod: "Constant Integer Multiplication using Minimum Adders," *IEE Proceedings - Circuits, Devices & Systems* **141**, 407–413 (1994)
106. K. Parhi, D. Messerschmidt: "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part I: Pipelining Using Scattered Look-Ahead and Decomposition," *IEEE Transactions on Acoustics, Speech and Signal Processing* **37**(7), 1099–1117 (1989)
107. H. Loomis, B. Sinha: "High Speed Recursive Digital Filter Realization," *Circuits, Systems, Signal Processing* **3**(3), 267–294 (1984)
108. M. Soderstrand, A. de la Serna, H. Loomis: "New Approach to Clustered Look-ahead Pipelined IIR Digital Filters," *IEEE Transactions on Circuits and Systems II* **42**(4), 269–274 (1995)
109. J. Living, B. Al-Hashimi: "Mixed Arithmetic Architecture: A Solution to the Iteration Bound for Resource Efficient FPGA and CPLD Recursive Digital Filters," in *IEEE International Symposium on Circuits and Systems* Vol. I (1999), pp. 478–481
110. H. Martinez, T. Parks: "A Class of Infinite-Duration Impulse Response Digital Filters for Sampling Rate Reduction," *IEEE Transactions on Acoustics, Speech and Signal Processing* **26**(4), 154–162 (1979)

111. K. Parhi, D. Messerschmidt: "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part II: Pipelined Incremental Block Filtering," *IEEE Transactions on Acoustics, Speech and Signal Processing* **37**(7), 1118–1134 (1989)
112. A. Gray, J. Markel: "Digital Lattice and Ladder Filter Synthesis," *IEEE Transactions on Audio and Electroacoustics* **21**(6), 491–500 (1973)
113. L. Gazsi: "Explicit Formulas for Lattice Wave Digital Filters," *IEEE Transactions on Circuits and Systems* pp. 68–88 (1985)
114. J. Xu, U. Meyer-Baese, K. Huang: "FPGA-based solution for real-time tracking of time-varying harmonics and power disturbances," *International Journal of Power Electronics (IJEPEC)* **4**(2), 134–159 (2012)
115. J. Xu (2009): "FPGA-based Real Time Processing of Time-Varying Waveform Distortions and Power Disturbances in Power Systems," Ph.D. thesis, Florida State University
116. L. Jackson: "Roundoff-Noise Analysis for Fixed-point Digital Filters Realized in Cascade or Parallel Form," *IEEE Transactions on Audio and Electroacoustics* **18**(2), 107–123 (1970)
117. W. Hess: *Digitale Filter* (Teubner Studienbücher, Stuttgart, 1989)
118. H.L. Arriens: (2013), "(L)WDF Toolbox for MATLAB," personal communication
URL <http://ens.ewi.tudelft.nl/~huib/mtbx/>
119. T. Saramaki: "On the Design of Digital Filters as a Sum of Two All-pass Filters," *IEEE Transactions on Circuits and Systems* pp. 1191–1193 (1985)
120. P. Vaidyanathan, P. Regalia, S. Mitra: "Design of doubly complementary IIR digital filters using a single complex allpass filter, with multirate applications," *IEEE Transactions on Circuits and Systems* **34**, 378–389 (1987)
121. M. Anderson, S. Summerfield, S. Lawson: "Realisation of lattice wave digital filters using three-port adaptors," *IEE Electronics Letters* pp. 628–629 (1995)
122. M. Shajaan, J. Sorensen: "Time-Area Efficient Multiplier-Free Recursive Filter Architectures for FPGA Implementation," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1996), pp. 3269–3272
123. P. Vaidyanathan: *Multirate Systems and Filter Banks* (Prentice Hall, Englewood Cliffs, New Jersey, 1993)
124. S. Winograd: "On Computing the Discrete Fourier Transform," *Mathematics of Computation* **32**, 175–199 (1978)
125. Z. Mou, P. Duhamel: "Short-Length FIR Filters and Their Use in Fast Non-recursive Filtering," *IEEE Transactions on Signal Processing* **39**, 1322–1332 (1991)
126. P. Balla, A. Antoniou, S. Morgera: "Higher Radix Aperiodic-Convolution Algorithms," *IEEE Transactions on Acoustics, Speech and Signal Processing* **34**(1), 60–68 (1986)
127. E.B. Hogenauer: "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing* **29**(2), 155–162 (1981)
128. Harris: (1992), "Data sheet," HSP43220 Decimating Digital Filter
129. Motorola: (1989), "Datasheet," DSP56ADC16 16-Bit Sigma-Delta Analog-to-Digital Converter
130. Intersil: (2000), "Data sheet," HSP50214 Programmable Downconverter
131. Texas Instruments: (2000), "Data sheet," GC4114 Quad Transmit Chip
132. Altera: (2007), "Understanding CIC Compensation Filters," application note 455, Ver. 1.0

133. O. Six (1996): "Design and Implementation of a Xilinx universal XC-4000 FPGAs board," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
134. S. Dworak (1996): "Design and Realization of a new Class of Frequency Sampling Filters for Speech Processing using FPGAs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
135. L. Wang, W. Hsieh, T. Truong: "A Fast Computation of 2-D Cubic-spline Interpolation," *IEEE Signal Processing Letters* **11**(9), 768–771 (2004)
136. T. Laakso, V. Valimäki, M. Karjalainen, U. Laine: "Splitting the Unit Delay," *IEEE Signal Processing Magazine* **13**(1), 30–60 (1996)
137. M. Unser: "Splines: a Perfect Fit for Signal and Image Processing," *IEEE Signal Processing Magazine* **16**(6), 22–38 (1999)
138. S. Cucchi, F. Desinan, G. Parladori, G. Sicuranza: "DSP Implementation of Arbitrary Sampling Frequency Conversion for High Quality Sound Application," in *IEEE International Symposium on Circuits and Systems* Vol. 5 (1991), pp. 3609–3612
139. C. Farrow: "A Continuously Variable Digital Delay Element," in *IEEE International Symposium on Circuits and Systems* Vol. 3 (1988), pp. 2641–2645
140. S. Mitra: *Digital Signal Processing: A Computer-Based Approach*, 3rd edn. (McGraw Hill, Boston, 2006)
141. S. Dooley, R. Stewart, T. Durrani: "Fast On-line B-spline Interpolation," *IEE Electronics Letters* **35**(14), 1130–1131 (1999)
142. Altera: "Farrow-Based Decimating Sample Rate Converter," in *Altera application note AN-347* San Jose (2004)
143. F. Harris: "Performance and Design Considerations of the Farrow Filter when used for Arbitrary Resampling of Sampled Time Series," in *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems & Computers* Vol. 2 (1997), pp. 1745–1749
144. M. Unser, A. Aldroubi, M. Eden: "B-spline Signal Processing: I– Theory," *IEEE Transactions on Signal Processing* **41**(2), 821–833 (1993)
145. P. Vaidyanathan: "Generalizations of the Sampling Theorem: Seven Decades after Nyquist," *Circuits and Systems I: Fundamental Theory and Applications* **48**(9), 1094–1109 (2001)
146. Z. Mihajlovic, A. Goluban, M. Zagar: "Frequency Domain Analysis of B-spline Interpolation," in *Proceedings of the IEEE International Symposium on Industrial Electronics* Vol. 1 (1999), pp. 193–198
147. M. Unser, A. Aldroubi, M. Eden: "Fast B-spline Transforms for Continuous Image Representation and Interpolation," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **13**(3), 277–285 (1991)
148. M. Unser, A. Aldroubi, M. Eden: "B-spline Signal Processing: II– Efficiency Design and Applications," *IEEE Transactions on Signal Processing* **41**(2), 834–848 (1993)
149. M. Unser, M. Eden: "FIR Approximations of Inverse Filters and Perfect Reconstruction Filter Banks," *Signal Processing* **36**(2), 163–174 (1994)
150. T. Blu, P. Thévenaz, M. Unser: "MOMS: Maximal-Order Interpolation of Minimal Support," *IEEE Transactions on Image Processing* **10**(7), 1069–1080 (2001)
151. T. Blu, P. Thévenaz, M. Unser: "High-Quality Causal Interpolation for On-line Unidimensional Signal Processing," in *Proceedings of the Twelfth European Signal Processing Conference (EUSIPCO'04)* (2004), pp. 1417–1420
152. A. Gotchev, J. Vesma, T. Saramäki, K. Egiazarian: "Modified B-Spline Functions for Efficient Image Interpolation," in *First IEEE Balkan Conference on*

- Signal Processing, Communications, Circuits, and Systems* (2000), pp. 241–244
153. W. Hawkins: “FFT Interpolation for Arbitrary Factors: a Comparison to Cubic Spline Interpolation and Linear Interpolation,” in *Proceedings IEEE Nuclear Science Symposium and Medical Imaging Conference* Vol. 3 (1994), pp. 1433–1437
 154. A. Haar: “Zur Theorie der orthogonalen Funktionensysteme,” *Mathematische Annalen* **69**, 331–371 (1910). Dissertation Göttingen 1909
 155. W. Sweldens: “The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Constructions,” in *SPIE, Wavelet Applications in Signal and Image Processing III* (1995), pp. 68–79
 156. C. Herley, M. Vetterli: “Wavelets and Recursive Filter Banks,” *IEEE Transactions on Signal Processing* **41**, 2536–2556 (1993)
 157. I. Daubechies: *Ten Lectures on Wavelets* (Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992)
 158. I. Daubechies, W. Sweldens: “Factoring Wavelet Transforms into Lifting Steps,” *The Journal of Fourier Analysis and Applications* **4**, 365–374 (1998)
 159. G. Strang, T. Nguyen: *Wavelets and Filter Banks* (Wellesley-Cambridge Press, Wellesley MA, 1996)
 160. D. Esteban, C. Galand: “Applications of Quadrature Mirror Filters to Split Band Voice Coding Schemes,” in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1977), pp. 191–195
 161. M. Smith, T. Barnwell: “Exact Reconstruction Techniques for Tree-Structured Subband Coders,” *IEEE Transactions on Acoustics, Speech and Signal Processing* pp. 434–441 (1986)
 162. M. Vetterli, J. Kovacevic: *Wavelets and Subband Coding* (Prentice Hall, Englewood Cliffs, New Jersey, 1995)
 163. R. Crochiere, L. Rabiner: *Multirate Digital Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1983)
 164. M. Achero, J.M. Mangen, Y. Buhler.: “Progressive Wavelet Algorithm versus JPEG for the Compression of METEOSAT Data,” in *SPIE, San Diego* (1995)
 165. T. Ebrahimi, M. Kunt: “Image Compression by Gabor Expansion,” *Optical Engineering* **30**, 873–880 (1991)
 166. D. Gabor: “Theory of communication,” *J. Inst. Elect. Eng (London)* **93**, 429–457 (1946)
 167. A. Grossmann, J. Morlet: “Decomposition of Hardy Functions into Square Integrable Wavelets of Constant Shape,” *SIAM J. Math. Anal.* **15**, 723–736 (1984)
 168. U. Meyer-Bäse: “High Speed Implementation of Gabor and Morlet Wavelet Filterbanks using RNS Frequency Sampling Filters,” in *Aerosense 98 *SPIE*, Orlando (1998), pp. 522–533*
 169. U. Meyer-Bäse: “Die Hutlets – eine biorthogonale Wavelet-Familie: Effiziente Realisierung durch multiplizierfreie, perfekt rekonstruierende Quadratur Mirror Filter,” *Frequenz* pp. 39–49 (1997)
 170. U. Meyer-Bäse, F. Taylor: “The Hutlets - a Biorthogonal Wavelet Family and their High Speed Implementation with RNS, Multiplier-free, Perfect Reconstruction QMF,” in *Aerosense 97 SPIE, Orlando* (1997), pp. 670–681
 171. D. Donoho, I. Johnstone: “Ideal Spatial Adatation by Wavelet Shrinkage,” *Biometrika* **81**(3), 425–545 (1994)
 172. S. Mallat: *A Wavelet Tour of Signal Processing* (Academic Press, San Diego, USA, 1998)
 173. D. Donoho, I. Johnstone, G. Kerkyacharian, D. Picard: “Wavelet Shrinkage: Asymptopia?,” *J. Roy. Statist. Soc.* **57**(2), 301–369 (1995)

174. M. Heideman, D. Johnson, C. Burrus: "Gauss and the History of the Fast Fourier Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* **34**, 265–267 (1985)
175. C. Burrus: "Index Mappings for Multidimensional Formulation of the DFT and Convolution," *IEEE Transactions on Acoustics, Speech and Signal Processing* **25**, 239–242 (1977)
176. B. Baas (1997): "An Approach to Low-power, High-performance, Fast Fourier Transform Processor Design," Ph.D. thesis, Stanford University
177. G. Sunada, J. Jin, M. Berzins, T. Chen: "COBRA: An 1.2 Million Transistor Exandable Column FFT Chip," in *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors* (IEEE Computer Society Press, Los Alamitos, CA, USA, 1994), pp. 546–550
178. TMS: (1996), "TM-66 swiFFT Chip," Texas Memory Systems
179. SHARP: (1997), "BDSP9124," digital signal processor
180. J. Mellott (1997): "Long Instruction Word Computer," Ph.D. thesis, University of Florida, Gainesville
181. P. Lavoie: "A High-Speed CMOS Implementation of the Winograd Fourier Transform Algorithm," *IEEE Transactions on Signal Processing* **44**(8), 2121–2126 (1996)
182. G. Panneerselvam, P. Graumann, L. Turner: "Implementation of Fast Fourier Transforms and Discrete Cosine Transforms in FPGAs," in *Lecture Notes in Computer Science* Vol. 1142 (1996), pp. 1142:272–281
183. Altera: "Fast Fourier Transform," in *Solution Brief 12, Altera Corporation* (1997)
184. G. Goslin: "Using Xilinx FPGAs to Design Custom Digital Signal Processing Devices," in *Proceedings of the DSP^X* (1995), pp. 595–604
185. C. Dick: "Computing 2-D DFTs Using FPGAs," *Lecture Notes in Computer Science: Field-Programmable Logic* pp. 96–105 (1996)
186. S.D. Stearns, D.R. Hush: *Digital Signal Analysis* (Prentice Hall, Englewood Cliffs, New Jersey, 1990)
187. K. Kammeyer, K. Kroschel: *Digitale Signalverarbeitung* (Teubner Studienbücher, Stuttgart, 1989)
188. E. Brigham: *FFT*, 3rd edn. (Oldenbourg Verlag, München Wien, 1987)
189. R. Ramirez: *The FFT: Fundamentals and Concepts* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
190. R.E. Blahut: *Theory and practice of error control codes* (Addison-Wesley, Melo Park, California, 1984)
191. C. Burrus, T. Parks: *DFT/FFT and Convolution Algorithms* (John Wiley & Sons, New York, 1985)
192. D. Elliott, K. Rao: *Fast Transforms: Algorithms, Analyses, Applications* (Academic Press, New York, 1982)
193. A. Nuttall: "Some Windows with Very Good Sidelobe Behavior," *IEEE Transactions on Acoustics, Speech and Signal Processing* **ASSP-29**(1), 84–91 (1981)
194. U. Meyer-Bäse, K. Damm (1988): "Fast Fourier Transform using Signal Processor," Master's thesis, Department of Information Science, Darmstadt University of Technology
195. M. Narasimha, K. Shenoi, A. Peterson: "Quadratic Residues: Application to Chirp Filters and Discrete Fourier Transforms," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1976), pp. 12–14
196. C. Rader: "Discrete Fourier Transform when the Number of Data Samples is Prime," *Proceedings of the IEEE* **56**, 1107–8 (1968)

197. J. McClellan, C. Rader: *Number Theory in Digital Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1979)
198. I. Good: "The Relationship between Two Fast Fourier Transforms," *IEEE Transactions on Computers* **20**, 310–317 (1971)
199. L. Thomas: "Using a Computer to Solve Problems in Physics," in *Applications of Digital Computers* (Ginn, Dordrecht, 1963)
200. A. Dandalis, V. Prasanna: "Fast Parallel Implementation of DFT Using Configurable Devices," *Lecture Notes in Computer Science* **1304**, 314–323 (1997)
201. U. Meyer-Bäse, S. Wolf, J. Mellott, F. Taylor: "High Performance Implementation of Convolution on a Multi FPGA Board using NTT's defined over the Eisenstein Residuen Number System," in *Aerosense 97 SPIE, Orlando* (1997), pp. 431–442
202. Xilinx: (2000), "High-Performance 256-Point Complex FFT/IFFT," product specification
203. Altera: (2012), "FFT MegaCore Function: User Guide," UG-FFT-12.0
204. Z. Wang: "Fast Algorithms for the Discrete W transform and for the discrete Fourier Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* pp. 803–816 (1984)
205. M. Narasimha, A. Peterson: "On the Computation of the Discrete Cosine Transform," *IEEE Transaction on Communications* **26**(6), 934–936 (1978)
206. K. Rao, P. Yip: *Discrete Cosine Transform* (Academic Press, San Diego, CA, 1990)
207. B. Lee: "A New Algorithm to Compute the Discrete Cosine Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* **32**(6), 1243–1245 (1984)
208. S. Ramachandran, S. Srinivasan, R. Chen: "EPLD-Based Architecture of Real Time 2D-discrete Cosine Transform and Quantization for Image Compression," in *IEEE International Symposium on Circuits and Systems* Vol. III (1999), pp. 375–378
209. C. Burrus, P. Eschenbacher: "An In-Place, In-Order Prime Factor FFT Algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing* **29**(4), 806–817 (1981)
210. H. Lüke: *Signalübertragung* (Springer, Heidelberg, 1988)
211. D. Herold, R. Huthmann (1990): "Decoder for the Radio Data System (RDS) using Signal Processor TMS320C25," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
212. U. Meyer-Bäse, R. Watzel: "A comparison of DES and LFSR based FPGA Implementable Cryptography Algorithms," in *3rd International Symposium on Communication Theory & Applications* (1995), pp. 291–298
213. U. Meyer-Bäse, R. Watzel: "An Optimized Format for Long Frequency Paging Systems," in *3rd International Symposium on Communication Theory & Applications* (1995), pp. 78–79
214. U. Meyer-Bäse: "Convolutional Error Decoding with FPGAs," *Lecture Notes in Computer Science* **1142**, 376–175 (1996)
215. R. Watzel (1993): "Design of Paging Scheme and Implementation of the Suitable Crypto-Controller using FPGAs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
216. J. Maier, T. Schubert (1993): "Design of Convolutional Decoders using FPGAs for Error Correction in a Paging System," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
217. Y. Gao, D. Herold, U. Meyer-Bäse: "Zum bestehenden Übertragungsprotokoll kompatible Fehlerkorrektur," in *Funkuhren Zeitsignale Normalfrequenzen* (1993), pp. 99–112

218. D. Herold (1991): "Investigation of Error Corrections Steps for DCF77 Signals using Programmable Gate Arrays," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
219. P. Sweeney: *Error Control Coding* (Prentice Hall, New York, 1991)
220. D. Wiggert: *Error-Control Coding and Applications* (Artech House, Dedham, Mass., 1988)
221. G. Clark, J. Cain: *Error-Correction Coding for Digital Communications* (Plenum Press, New York, 1988)
222. W. Stahnke: "Primitive Binary Polynomials," *Mathematics of Computation* pp. 977–980 (1973)
223. W. Fumy, H. Riess: *Kryptographie* (R. Oldenbourg Verlag, München, 1988)
224. B. Schneier: *Applied Cryptography* (John Wiley & Sons, New York, 1996)
225. M. Langhammer: "Reed-Solomon Codec Design in Programmable Logic," *Communication System Design* (www.csdmag.com) pp. 31–37 (1998)
226. B. Akers: "Binary Decusion Diagrams," *IEEE Transactions on Computers* pp. 509–516 (1978)
227. R. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers* pp. 677–691 (1986)
228. A. Sangiovanni-Vincentelli, A. Gamal, J. Rose: "Synthesis Methods for Field Programmable Gate Arrays," *Proceedings of the IEEE* pp. 1057–83 (1993)
229. R. del Rio (1993): "Synthesis of boolean Functions for Field Programmable Gate Arrays," Master's thesis, Univerity of Frankfurt, FB Informatik
230. U. Meyer-Bäse: "Optimal Strategies for Incoherent Demodulation of Narrow Band FM Signals," in *3rd International Symposium on Communication Theory & Applications* (1995), pp. 30–31
231. J. Proakis: *Digital Communications* (McGraw-Hill, New York, 1983)
232. R. Johannesson: "Robustly Optimal One-Half Binary Convolutional Codes," *IEEE Transactions on Information Theory* pp. 464–8 (1975)
233. J. Massey, D. Costello: "Nonsystematic Convolutional Codes for Sequential Decoding in Space Applications," *IEEE Transactions on Communications* pp. 806–813 (1971)
234. F. MacWilliams, J. Sloane: "Pseudo-Random Sequences and Arrays," *Proceedings of the IEEE* pp. 1715–29 (1976)
235. T. Lewis, W. Payne: "Generalized Feedback Shift Register Pseudorandom Number Algorithm," *Journal of the Association for Computing Machinery* pp. 456–458 (1973)
236. P. Bratley, B. Fox, L. Schrage: *A Guide to Simulation* (Springer-Lehrbuch, Heidelberg, 1983), pp. 186–190
237. M. Schroeder: *Number Theory in Science and Communication* (Springer, Heidelberg, 1990)
238. P. Kocher, J. Jaffe, B. Jun: "Differential Power Analysis," in *Lecture Note in Computer Science* (1999), pp. 388–397
239. EFF: *Cracking DES* (O'Reilly & Associates, Sebastopol, 1998), Electronic Frontier Foundation
240. W. Stallings: "Encryption Choices Beyond DES," *Communication System Design* (www.csdmag.com) pp. 37–43 (1998)
241. W. Carter: "FPGAs: Go reconfigure," *Communication System Design* (www.csdmag.com) p. 56 (1998)
242. J. Anderson, T. Aulin, C.E. Sundberg: *Digital Phase Modulation* (Plenum Press, New York, 1986)

243. U. Meyer-Bäse (1989): "Investigation of Thresholdimproving Limiter/Discriminator Demodulator for FM Signals through Computer simulations," Master's thesis, Department of Information Science, Darmstadt University of Technology
244. E. Allmann, T. Wolf (1991): "Design and Implementation of a full digital zero IF Receiver using programmable Gate Arrays and Floatingpoint DSPs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
245. O. Herrmann: "Quadraturfilter mit rationalem Übertragungsfaktor," *Archiv der elektrischen Übertragung (AEÜ)* pp. 77–84 (1969)
246. O. Herrmann: "Transversalfilter zur Hilbert-Transformation," *Archiv der elektrischen Übertragung (AEÜ)* pp. 581–587 (1969)
247. V. Considine: "Digital Complex Sampling," *Electronics Letters* pp. 608–609 (1983)
248. T.E. Thiel, G.J. Saulnier: "Simplified Complex Digital Sampling Demodulator," *Electronics Letters* pp. 419–421 (1990)
249. U. Meyer-Bäse, W. Hilberg: (1992), "Schmalbandempfänger für Digitalsignale," German patent no. 4219417.2-31
250. B. Schlanske (1992): "Design and Implementation of a Universal Hilbert Sampling Receiver with CORDIC Demodulation for LF FAX Signals using Digital Signal Processor," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
251. A. Dietrich (1992): "Realisation of a Hilbert Sampling Receiver with CORDIC Demodulation for DCF77 Signals using Floatingpoint Signal Processors," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
252. A. Viterbi: *Principles of Coherent Communication* (McGraw-Hill, New York, 1966)
253. F. Gardner: *Phaselock Techniques* (John Wiley & Sons, New York, 1979)
254. H. Geschwinde: *Einführung in die PLL-Technik* (Vieweg, Braunschweig, 1984)
255. R. Best: *Theorie und Anwendung des Phase-locked Loops* (AT Press, Schwtizerland, 1987)
256. W. Lindsey, C. Chie: "A Survey of Digital Phase-Locked Loops," *Proceedings of the IEEE* pp. 410–431 (1981)
257. R. Sanneman, J. Rowbotham: "Unlock Characteristics of the Optimum Type II Phase-Locked Loop," *IEEE Transactions on Aerospace and Navigational Electronics* pp. 15–24 (1964)
258. J. Stensby: "False Lock in Costas Loops," *Proceedings of the 20th Southeastern Symposium on System Theory* pp. 75–79 (1988)
259. A. Mararios, T. Tozer: "False-Lock Performance Improvement in Costas Loops," *IEEE Transactions on Communications* pp. 2285–88 (1982)
260. A. Makarios, T. Tozer: "False-Look Avoidance Scheme for Costas Loops," *Electronics Letters* pp. 490–2 (1981)
261. U. Meyer-Bäse: "Coherent Demodulation with FPGAs," *Lecture Notes in Computer Science* **1142**, 166–175 (1996)
262. J. Guyot, H. Schmitt (1993): "Design of a full digital Costas Loop using programmable Gate Arrays for coherent Demodulation of Low Frequency Signals," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
263. R. Resch, P. Schreiner (1993): "Design of Full Digital Phase Locked Loops using programmable Gate Arrys for a low Frequency Reciever," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
264. D. McCarty: "Digital PLL Suits FPGAs," *Elektronik Design* p. 81 (1992)

265. J. Holmes: "Tracking-Loop Bias Due to Costas Loop Arm Filter Imbalance," *IEEE Transactions on Communications* pp. 2271–3 (1982)
266. H. Choi: "Effect of Gain and Phase Imbalance on the Performance of Lock Detector of Costas Loop," *IEEE International Conference on Communications, Seattle* pp. 218–222 (1987)
267. N. Wiener: *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (John Wiley & Sons, New York, 1949)
268. S. Haykin: *Adaptive Filter Theory* (Prentice Hall, Englewood Cliffs, New Jersey, 1986)
269. B. Widrow, S. Stearns: *Adaptive Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
270. C. Cowan, P. Grant: *Adaptive Filters* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
271. A. Papoulis: *Probability, Random Variables, and Stochastic Processes* (McGraw-Hill, Singapore, 1986)
272. M. Honig, D. Messerschmitt: *Adaptive Filters: Structures, Algorithms, and Applications* (Kluwer Academic Publishers, Norwell, 1984)
273. S. Alexander: *Adaptive Signal Processing: Theory and Application* (Springer, Heidelberg, 1986)
274. N. Shanbhag, K. Parhi: *Pipelined Adaptive Digital Filters* (Kluwer Academic Publishers, Norwell, 1994)
275. B. Mulgrew, C. Cowan: *Adaptive Filters and Equalisers* (Kluwer Academic Publishers, Norwell, 1988)
276. J. Treichler, C. Johnson, M. Larimore: *Theory and Design of Adaptive Filters* (Prentice Hall, Upper Saddle River, New Jersey, 2001)
277. B. Widrow, J. Glover, J. McCool, J. Kaunitz, C. Williams, R. Hearn, J. Zeidler, E. Dong, R. Goodlin: "Adaptive Noise Cancelling: Principles and Applications," *Proceedings of the IEEE* **63**, 1692–1716 (1975)
278. B. Widrow, J. McCool, M. Larimore, C. Johnson: "Stationary and Nonstationary Learning Characteristics of the LMS Adaptive Filter," *Proceedings of the IEEE* **64**, 1151–1162 (1976)
279. T. Kummura, M. Ikekawa, M. Yoshida, I. Kuroda: "VLIW DSP for Mobile Applications," *IEEE Signal Processing Magazine* **19**, 10–21 (2002)
280. Analog Device: "Application Handbook," 1987
281. L. Horowitz, K. Senne: "Performance Advantage of Complex LMS for Controlling Narrow-Band Adaptive Arrays," *IEEE Transactions on Acoustics, Speech and Signal Processing* **29**, 722–736 (1981)
282. A. Feuer, E. Weinstein: "Convergence Analysis of LMS Filters with Uncorrelated Gaussian Data," *IEEE Transactions on Acoustics, Speech and Signal Processing* **33**, 222–230 (1985)
283. S. Narayan, A. Peterson, M. Narasimha: "Transform Domain LMS Algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing* **31**, 609–615 (1983)
284. G. Clark, S. Parker, S. Mitra: "A Unified Approach to Time- and Frequency-Domain Realization of FIR Adaptive Digital Filters," *IEEE Transactions on Acoustics, Speech and Signal Processing* **31**, 1073–1083 (1983)
285. F. Beaufays (1995): "Two-Layer Structures for Fast Adaptive Filtering," Ph.D. thesis, Stanford University
286. A. Feuer: "Performance Analysis of Block Least Mean Square Algorithm," *IEEE Transactions on Circuits and Systems* **32**, 960–963 (1985)
287. D. Marshall, W. Jenkins, J. Murphy: "The use of Orthogonal Transforms for Improving Performance of Adaptive Filters," *IEEE Transactions on Circuits and Systems* **36**(4), 499–510 (1989)

288. J. Lee, C. Un: "Performance of Transform-Domain LMS Adaptive Digital Filters," *IEEE Transactions on Acoustics, Speech and Signal Processing* **34**(3), 499–510 (1986)
289. G. Long, F. Ling, J. Proakis: "The LMS Algorithm with Delayed Coefficient Adaption," *IEEE Transactions on Acoustics, Speech and Signal Processing* **37**, 1397–1405 (1989)
290. G. Long, F. Ling, J. Proakis: "Corrections to "The LMS Algorithm with Delayed Coefficient Adaption",," *IEEE Transactions on Signal Processing* **40**, 230–232 (1992)
291. R. Poltmann: "Conversion of the Delayed LMS Algorithm into the LMS Algorithm," *IEEE Signal Processing Letters* **2**, 223 (1995)
292. T. Kimijima, K. Nishikawa, H. Kiya: "An Effective Architecture of Pipelined LMS Adaptive Filters," *IEICE Transactions Fundamentals* **E82-A**, 1428–1434 (1999)
293. D. Jones: "Learning Characteristics of Transpose-Form LMS Adaptive Filters," *IEEE Transactions on Circuits and Systems II* **39**(10), 745–749 (1992)
294. M. Rupp, R. Frenzel: "Analysis of LMS and NLMS Algorithms with Delayed Coefficient Update Under the Presence of Spherically Invariant Processes," *IEEE Transactions on Signal Processing* **42**, 668–672 (1994)
295. M. Rupp: "Saving Complexity of Modified Filtered-X-LMS and Delayed Update LMS," *IEEE Transactions on Circuits and Systems II* **44**, 57–60 (1997)
296. M. Rupp, A. Sayed: "Robust FxLMS Algorithms with Improved Convergence Performance," *IEEE Transactions on Speech and Audio Processing* **6**, 78–85 (1998)
297. L. Ljung, M. Morf, D. Falconer: "Fast Calculation of Gain Matrices for Recursive Estimation Schemes," *International Journal of Control* **27**, 1–19 (1978)
298. G. Carayannis, D. Manolakis, N. Kalouptsidis: "A Fast Sequential Algorithm for Least-Squares Filtering and Prediction," *IEEE Transactions on Acoustics, Speech and Signal Processing* **31**, 1394–1402 (1983)
299. F. Albu, J. Kadlec, C. Softley, R. Matousek, A. Hermanek, N. Coleman, A. Fagan: "Implementation of (Normalised RLS Lattice on Virtex," *Lecture Notes in Computer Science* **2147**, 91–100 (2001)
300. D. Morales, A. Garcia, E. Castillo, U. Meyer-Baese, A. Palma: "Wavelets for full reconfigurable ECG acquisition system)," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2011), pp. 805 817–1–8
301. M. Keralapura, M. Pourfathi, B. Sikeci-Mergen: "Impact of Contrast Functions in Fast-ICA on Twin ECG Separation," *IAENG International Journal of Computer Science* **38**(1), 1–10 (2011)
302. D. Watkins: *Fundamentals of Matrix Computations* (John Wiley & Sons, New York, 1991)
303. G. Engeln-Müllges, F. Reutter: *Numerisch Mathematik für Ingenieure* (BI Wissenschaftsverlag, Mannheim, 1987)
304. K.K. Shyu, M.H. Li: "FPGA Implementation of FastICA Based on Floating-Point Arithmetic Design for Real-Time Blind Source Separation," in *International Joint Conference on Neural Networks* Vancouver, BC, Canada (2006), pp. 2785–2792
305. Altera: (2008), "QR Matrix Decomposition," application note 506, Ver. 2.0
306. A. Cichocki, R. Unbehauen: *Neural Networks for Optimization and Signal Processing* (John Wiley & Sons, New York, 1993)
307. T. Sanger (1993): "Optimal Unsupervised Learning in Feedforward Neural Networks," Master's thesis, MIT, Dept. of E&C Science

308. Y. Hirai, K. Nishizawa: "Hardware Implementation of the PCA Learning Network by Asynchronous PDM Digital Circuit," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* (2000), pp. 65–70
309. S. Kung, K. Diamantaras, J. Taur: "Adaptive Principal Component EXtraction (APEX) and Applications," *IEEE Transactions on Signal Processing* **42**(5), 1202–1216 (1994)
310. B. Yang: "Projection Approximation Subspace Tracking," *IEEE Transactions on Signal Processing* **43**(1), 95–107 (1995)
311. K. Abed-Meraim, A. Chkeif, Y. Hua: "Fast Orthonormal PAST Algorithm," *IEEE Signal Processing Letters* **7**(3), 60–62 (2000)
312. C. Jutten, J. Herault: "Blind Separation of Source, Part I: An Adaptive Algorithm Based on Neuromimetic Architecture," *Signal Processing* **24**(1), 1–10 (1991)
313. A. Hyvaerinen, J. Karhunen, E. Oja: *Independent Component Analysis* (John Wiley & Sons, New York, 2001)
314. A. Cichocki, S. Amari: *Adaptive blind signal and image processing* (John Wiley & Sons, New York, 2002)
315. S. Choi, A. Cichocki, H. Park, S. Lee: "Blind Source Separation and Independent Component Analysis: A Review," *Neural Information Processing - Letters and Reviews* **6**(1), 1–57 (2005)
316. S. Makino: "Blind source separation of convolutive mixtures," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2006), pp. 624709–1–15
317. J. Cardoso, B. Laheld: "Equivariant Adaptive Source Separation," *IEEE Transactions on Signal Processing* **44**(12), 3017–3029 (1996)
318. S. Kim, K. Umeno, R. Takahashi: "FPGA implementation of EASI algorithm," *IEICE Electronics Express* **22**(4), 707–711 (2007)
319. L. Yuan, Z. Sun: "A Survey of Using Sign Function to Improve The Performance of EASI Algorithm," in *Proceedings of the 2007 IEEE International Conference on Mechatronic and Automation* Harbin, China (2007), pp. 2456–2460
320. C. Odom (2013): "Independent Component Analysis Algorithm Fpga Design To Perform Real-Time Blind Source Separation," Master's thesis, Florida State University
321. J. Karhunen, E. Oja, L. Wang, R. Vigarío, J. Joutsensalo: "A Class of Neural Networks for Independent Component Analysis," *IEEE Transactions on Neural Networks* **8**(3), 486–504 (1997)
322. A. Hyvarinen, E. Oja: "Independent Component Analysis: Algorithms and Applications," *Neural Networks* **13**(4), 411–430 (2000)
323. A. Hyvarinen, E. Oja: "Independent Component Analysis by general nonlinear Hebbian-like learning rules," *Signal Processing* **64**(1), 301–313 (1998)
324. H. Fastl, E. Zwicker: *Psychoacoustics: Facts and Models* (Springer, Berlin, 2010)
325. ITU-T: (1972), "General Aspects of Digital Transmission Systems," pulse code modulation (PCM) of Voice Frequencies, ITU-T Recommendation G.711
326. W. Chu: *Speech Coding Algorithms: Foundation and Evolution of Standardized Coders* (John Wiley & Sons, New York, 2003)
327. L. Rabiner, R. Schafer: *Theory and Applications of Digital Speech Processing* (Pearson, Upper Saddle River, 2011)
328. IMA: (1992), "Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems," IMA Digital Audio Focus and Technical Working Groups

329. D. Huang: "Lossless Compression for μ -Law (A-Law) and IMA ADPCM on the Basis of a Fast RLS Algorithm," in *IEEE International Conference on Multimedia and Expo* New York (2000), pp. 1775–1778
330. S. Lloyd: "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory* **28**(2), 129137 (1982)
331. D. Wong, B. Juang, A. Gray: "An 800 bit/s Vector Quantization LPC Vocoder," *IEEE Transactions on Acoustics, Speech and Signal Processing* **30**(5), 770–780 (1982)
332. Analog Device: *Digital Signal Processing Applications using the ADSP-2100 family* (Prentice Hall, Englewood Cliffs, New Jersey, 1995), vol. 2
333. S. Aramvith, M. Sun: *Handbook of Image and Video Processing* (Academic Press, New York, 2005), Chap. MPEG-1 and MPEG-2 Video Standards, editor: Al Bovik
334. D. Schulz (1997): "Compression of High Quality Digital Audio Signals using Noiseextraction," Ph.D. thesis, Department of Information Science, Darmstadt University of Technology
335. Xilinx: (2005), "PicoBlaze 8-bit Embedded Microcontroller User Guide," www.xilinx.com
336. V. Heuring, H. Jordan: *Computer Systems Design and Architecture*, 2nd edn. (Prentice Hall, Upper Saddle River, New Jersey, 2004), contribution by M. Murdocca
337. D. Patterson, J. Hennessy: *Computer Organization & Design: The Hardware/Software Interface*, 2nd edn. (Morgan Kaufman Publishers, Inc., San Mateo, CA, 1998)
338. J. Hennessy, D. Patterson: *Computer Architecture: A Quantitative Approach*, 3rd edn. (Morgan Kaufman Publishers, Inc., San Mateo, CA, 2003)
339. M. Murdocca, V. Heuring: *Principles of Computer Architecture*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2000)
340. W. Stallings: *Computer Organization & Architecture*, 6th edn. (Prentice Hall, Upper Saddle River, NJ, 2002)
341. R. Bryant, D. O'Hallaron: *Computer Systems: A Programmer's Perspective*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2003)
342. C. Rowen: *Engineering the Complex SOC*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2004)
343. S. Mazor: "The History of the Microcomputer – Invention and Evolution," *Proceedings of the IEEE* **83**(12), 1601–8 (1995)
344. H. Faggin, M. Hoff, S. Mazor, M. Shima: "The History of the 4004," *IEEE Micro Magazine* **16**, 10–20 (1996)
345. Intel: (2006), "Microprocessor Hall of Fame," <http://www.intel.com/museum>
346. Intel: (1980), "2920 Analog Signal Processor," design handbook
347. TI: (2000), "Technology Innovation," www.ti.com/sc/technovations
348. TI: (1983), "TMS3210 Assembly Language Programmer's Guide," digital signal processor products
349. TI: (1993), "TMS320C5x User's Guide," digital signal processor products
350. A. Device: (1993), "ADSP-2103," 3-Volt DSP Microcomputer
351. P. Koopman: *Stack Computers: The New Wave*, 1st edn. (Mountain View Press, La Honda, CA, 1989)
352. Xilinx: (2002), "Creating Embedded Microcontrollers," www.xilinx.com, Part 1-5
353. Altera: (2003), "Nios-32 Bit Programmer's Reference Manual," Nios embedded processor, Ver. 3.1
354. Xilinx: (2002), "Virtex-II Pro," documentation

355. Xilinx: (2005), "MicroBlaze – The Low-Cost and Flexible Processing Solution," www.xilinx.com
356. Altera: (2003), "Nios II Processor Reference Handbook," NII5V-1-5.0
357. B. Parhami: *Computer Architecture: From Microprocessor to Supercomputers*, 1st edn. (Oxford University Press, New York, 2005)
358. Altera: (2004), "Netseminar Nios processor," <http://www.altera.com>
359. A. Hoffmann, H. Meyr, R. Leupers: *Architecture Exploration for Embedded Processors with LISA*, 1st edn. (Kluwer Academic Publishers, Boston, 2002)
360. A. Aho, R. Sethi, J. Ullman: *Compilers: Principles, Techniques, and Tools*, 1st edn. (Addison Wesley Longman, Reading, Massachusetts, 1988)
361. R. Leupers: *Code Optimization Techniques for Embedded Processors*, 2nd edn. (Kluwer Academic Publishers, Boston, 2002)
362. R. Leupers, P. Marwedel: *Retargetable Compiler Technology for Embedded Systems*, 1st edn. (Kluwer Academic Publishers, Boston, 2001)
363. V. Paxson: (1995), "Flex, Version 2.5: A Fast Scanner Generator," <http://www.gnu.org>
364. C. Donnelly, R. Stallman: (2002), "Bison: The YACC-compatible Parser Generator," <http://www.gnu.org>
365. S. Johnson: (1975), "YACC – Yet Another Compiler-Compiler," technical report no. 32, AT&T
366. R. Stallman: (1990), "Using and Porting GNU CC," <http://www.gnu.org>
367. W. Lesk, E. Schmidt: (1975), "LEX – a Lexical Analyzer Generator," technical report no. 39, AT&T
368. T. Niemann: (2004), "A Compact Guide to LEX & YACC," <http://www.epaperpress.com>
369. J. Levine, T. Mason, D. Brown: *lex & yacc*, 2nd edn. (O'Reilly Media Inc., Beijing, 1995)
370. T. Parsons: *Intorduction to Compiler Construction*, 1st edn. (Computer Science Press, New York, 1992)
371. A. Schreiner, H. Friedman: *Introduction to Compiler Construction with UNIX*, 1st edn. (Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1985)
372. C. Fraser, D. Hanson: *A Retargetable C Compilers: Design and Implementation*, 1st edn. (Addison-Wesley, Boston, 2003)
373. V. Zivojnovic, J. Velarde, C. Schläger, H. Meyr: "DSPSTONE: A DSP-oriented Benchmarking Methodology," in *International Conference* (19), pp. 1–6
374. Institute for Integrated Systems for Signal Processing: (1994), "DSPstone," final report
375. W. Strauss: "Digital Signal Processing: The New Semiconductor Industry Technology Driver," *IEEE Signal Processing Magazine* pp. 52–56 (2000)
376. Xilinx: (2002), "Virtex-II Pro Platform FPGA," handbook
377. Xilinx: (2005), "Accelerated System Performance with APU-Enhanced Processing," Xcell Journal
378. ARM: (2001), "ARM922T with AHB: Product Overview," <http://www.arm.com>
379. ARM: (2000), "ARM9TDMI Technical Reference Manual," <http://www.arm.com>
380. ARM: (2011), "Cortex-A series processors," <http://www.arm.com>
381. Altera: (2004), "Nios Software Development Reference Manual," <http://www.altera.com>
382. Altera: (2004), "Nios Development Kit, APEX Edition," Getting Started User Guide
383. Altera: (2004), "Nios Development Board Document," <http://www.altera.com>

384. Altera: (2004), "Nios Software Development Tutorial," <http://www.altera.com>
385. Altera: (2004), "Custom Instruction Tutorial," <http://www.altera.com>
386. B. Fletcher: "FPGA Embedded Processors," in *Embedded Systems Conference* San Francisco, CA (2005), p. 18
387. U. Meyer-Baese, A. Vera, S. Rao, K. Lenk, M. Pattichis: "FPGA Wavelet Processor Design using Language for Instruction-set Architectures (LISA)," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2007), pp. 6576U1–U12
388. D. Sunkara (2004): "Design of Custom Instruction Set for FFT using FPGA-Based Nios processors," Master's thesis, Florida State University
389. U. Meyer-Baese, D. Sunkara, E. Castillo, E.A. Garcia: "Custom Instruction Set NIOS-Based OFDM Processor for FPGAs," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2006), pp. 6248o01–15
390. J. Ramirez, U. Meyer-Baese, A. Garcia: "Efficient Wavelet Architectures using Field- Programmable Logic and Residue Number System Arithmetic," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2004), pp. 222–232
391. D. Bailey: *Design for Embedded Image Processing on FPGAs*, 1st edn. (John Wiley & Sons, Asia, 2011)
392. W. Pratt: *Digital Image Processing*, 4th edn. (John Wiley & Sons, New York, 2007)
393. R. Gonzalez, R. Woods: *Digital Image Processing*, 2nd edn. (Prentice Hall, New Jersey, 2001)
394. L. Shapiro, G. Stockman: *Computer Vision*, 1st edn. (Prentice Hall, New Jersey, 2001)
395. Y. Wang, J. Ostermann, Y. Zhang: *Video Processing and Communications*, 1st edn. (Prentice Hall, New Jersey, 2001)
396. A. Weeks: *Fundamentals of Electronic Image Processing* (SPIE, US, 1996)
397. J. Bradley: (1994), "XV Interactive Image Display for the X Window System," version 3.10a
398. D. Ziou, S. Tabbone: "Edge Detection Techniques: An Overview," *International Journal Of Pattern Recognition And Image Analysis* **8**(4), 537–559 (1998)
399. ITU: (1992), "T.81 : Information Technology Digital Compression And Coding Of Continuous-Tone Still Images Requirements and Guidelines," CCITT Recommendation T.81
URL <http://www.itu.int/rec/T-REC-T.800-200208-I/en>
400. ITU: (2002), "T.800 : Information technology - JPEG 2000 image coding system: Core coding system," recommendation T.800
URL <http://www.itu.int/rec/T-REC-T.800-200208-I/en>
401. M. Marcellin, M. Gormish, A. Bilgin, M. Boliek: "An overview of JPEG-2000," in *Proceedings Data Compression Conference* (2000), pp. 523–541
402. C. Christopoulos, A. Skodras, T. Ebrahimi: "The JPEG2000 Still Image Coding System: An Overview," *IEEE Transactions on Consumer Electronics* **46**(4), 1103–1127 (2000)
403. T. Acharya, P. Tsai: *JPEG2000 Standard for Image Compression* (John Wiley & Sons, Inc., New Jersey, 2005)
404. B. Fornberg: "Generation of Finite Difference Formulas on Arbitrarily Spaced Grids," *Mathematics Of Computation* **51**(184), 699–706 (1988)
405. J. Prewitt: *Object Enhancement and Extraction* (Academic Press, New York, 1970), Chap. Picture Processing and Psychopictorics, editor: B. Lipkin
406. I. Sobel (1970): "Camera Models and Machine Perception," Ph.D. thesis, Stanford University, Palo Alto, CA

407. I. Abdou, W. Pratt: "Quantitative Design and Evaluation of Enhancement/Thresholding Edge Detectors," *Proceedings of the IEEE* **67**(5), 753–763 (1979)
408. J. Canny: "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(6), 679–698 (1986)
409. H. Neoh, A. Hazanchuk: "Adaptive Edge Detection for Real-Time Video Processing using FPGAs," in *Global Signal Processing* Santa Clara Convention Center (2004), pp. 1–6
410. Altera: (2004), "Edge Detection Reference Design," application Note 364, Ver. 1.0
411. Altera: (2007), "Video and Image Processing Design Using FPGAs," white Paper
412. Altera: (2012), "Video IP Cores for Altera DE-Series Boards," ver. 12.0
413. J. Scott, M. Pusateri, M. Mushtaq: "Comparison of 2D median filter hardware implementations for real-time stereo video," in *Applied Imagery Pattern Recognition Workshop* (2008), pp. 1–6
414. H. Eng, K. Ma: "Noise Adaptive Soft-Switching Median Filter," *IEEE Transactions on Image Processing* **10**(2), 242–251 (2001)
415. T. Huang, G. Yang, G. Tang: "A Fast Two-Dimensional Median Filtering Algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing* **27**(1), 13–18 (1979)
416. H. Hwang, A. Haddad: "Adaptive Median Filters; New Algorithms and Results," *IEEE Transactions on Image Processing* **4**(4), 499–502 (1995)
417. C. Thompson: "The VLSI Complexity of Sorting," *IEEE Transactions on Computers* **32**(12), 1171–1184 (1983)
418. Xilinx: "Two-Dimensional Rank Order Filter," in *Xilinx application note XAPP 953* San Jose (2006)
419. K. Batcher: "Sorting networks and their applications," in *Proceedings of the spring joint computer conference* (ACM, New York, NY, USA, 1968), pp. 307–314
420. G. Bates, S. Nooshabadi: "FPGA implementation of a median filter," in *Proceedings of IEEE Speech and Image Technologies for Computing and Telecommunications IEEE Region 10 Annual Conference* (1997), pp. 437–440
421. K. Benkrid, D. Crookes, A. Benkrid: "Design and implementation of a novel algorithm for general purpose median filtering on FPGAs," in *IEEE International Symposium on Circuits and Systems* Vol. IV (2002), pp. 425–428
422. S. Fahmy, P. Cheung, W. Luk: "Novel Fpga-Based Implementation Of Median And Weighted Median Filters For Image Processing," in *International Conference on Field Programmable Logic and Applications* (2005), pp. 142–147
423. Altera: (2010), "Media Computer System for the Altera DE2-115 Board," ver. 11.0
424. P. Pirsch, N. Demassieux, W. Gehrke: "VLSI Architectures for Video Compression-A Survey," *Proceedings of the IEEE* **83**(2), 220–246 (1995)
425. Y.M.C. Hsueh-Ming Hang, S.C. Cheng: "Motion Estimation for Video Coding Standards," *Journal of VLSI Signal Processing* **17**, 113–136 (1997)
426. M. Ghanbari: "The Cross-Search Algorithm for Motion Estimation," *IEEE Transactions On Communications* **38**(7), 1301–1308 (1990)
427. D. Gonzalez, G. Botella, S. Mookherjee, U. Meyer-Baese, A. Meyer-Baese: "NIOS II processor-based acceleration of motion compensation techniques," in *Proc. SPIE Int. Soc. Opt. Eng., Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering IX* (2011), pp. 80581C1–12, vol. 8058

428. J. Jain, A. Jain: "Displacement Measurement and Its Application in Inter-frame Image Coding," *IEEE Transactions On Communications* **29**(12), 1799–1808 (1981)
429. T. Koga, K. Inuma, A. Hirano, Y. Iijima, T. Ishiguro: "Motioncompensated interframe coding for video conferencing," in *Proc. National Telecommunication Conference* New Orleans (1981), pp. C9.6.1–9.6.5
430. R. Kappagantula, K. Rao: "Motion Compensated Interframe Image Prediction," *IEEE Transactions On Communications* **33**(9), 1011–1015 (1985)
431. R. Srinivasan, K. Rao: "Predictive Coding Based on Efficient Motion Estimation," *IEEE Transactions On Communications* **33**(8), 888–896 (1985)
432. A. Puri, H. Hang, D. Schilling: "An Efficient Block-Matching Algorithm For Motion-Compensated Coding," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1987), pp. 1063–1066
433. D. Gonzalez, G. Botella, U. Meyer-Baese, C. Garca, C. Sanz, M. Prieto-Matas, F. Tirado: "A Low Cost Matching Motion Estimation Sensor Based on the NIOS II Microprocessor," *Sensors* **12**(10), 13 126–13 149 (2012)
434. D. Gonzalez, G. Botella, A. Meyer-Baese, U. Meyer-Baese: "Optimization of block-matching algorithms unsing custom instruction based paradigm on Nios II microprocessors," in *Proc. SPIE Int. Soc. Opt. Eng., Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering XI* (2013), pp. 87500Q1–8
435. ITU: (1993), "Line Transmission of non-telephone signals: Video codec for audiovisual services at $p \times 64$ kbits," ITU-T Recommendation H.261
URL <http://www.itu.int/rec/T-REC-H.261/>
436. ITU: (2013), "Advanced video coding for generic audiovisual services," ITU-T Recommendation H.264
URL <http://www.itu.int/rec/T-REC-H.264/>
437. ITU: (1995), "Transmission of non-telephone signals Information technology - Generic coding of moving pictures and associated audio information: Video," ITU-T Recommendation H.262
URL <http://www.itu.int/rec/T-REC-H.262/>
438. ITU: (2005), "Video coding for low bit rate communication," ITU-T Recommendation H.263
URL <http://www.itu.int/rec/T-REC-H.263/>
439. G. Sullivan, J. Ohm, W. Han, T. Wiegand: "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions On Circuits And Systems For Video Technology* **22**(12), 1649–1668 (2012)
440. ITU: (2013), "High efficiency video coding," ITU-T Recommendation H.265
URL <http://www.itu.int/rec/T-REC-H.265/>
441. U. Meyer-Baese: *Digital Signal Processing with Field Programmable Gate Arrays*, 2nd edn. (Springer-Verlag, Berlin, 2004), 527 pages
442. J. Ousterhout: *Tcl and the Tk Toolkit*, 1st edn. (Addison-Wesley, Boston, 1994)
443. M. Harrison, M. McLennan: *Effective Tcl/Tk Programming*, 1st edn. (Addison-Wesley, Reading, Massachusetts, 1998)
444. B. Welch, K. Jones, H. J: *Practical Programming in Tcl and Tk*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2003)

Index

- Accumulator 9, 318
- μ P 647, 651
- Actel 8
- Adaptive filter 533–629
- Adder
 - binary 80
 - fast carry 80
 - floating-point 115, 125
 - LPM 81, 301
 - pipelined 83
 - size 81
 - speed 81
- ADPCM 618
- A-law 613
- Algorithms
 - Bluestein 424
 - chirp- z 424
 - Cooley–Tukey 442
 - CORDIC 131–141
 - common factor (CFA) 436
 - Goertzel 424
 - Good–Thomas 437
 - fast RLS 586
 - LMS 546, 588
 - prime factor (PFA) 436
 - Rader 427
 - Radix- r 440
 - RLS 575, 588
 - Widrow–Hoff LMS 546
 - Winograd DFT 434
 - Winograd FFT 452
- Altera
 - ARM922T μ P 686
 - ARM Cortex-A9 689
 - DE2 development board 20
 - Devices 8, 11, 22
 - Intellectual Property (IP) core 40
 - FFT 458
 - FIR filter 215
 - Image processing blocks 771
 - NCO 40
 - Nios 694
 - Nios II 700, 721, 769, 782
 - PREP test bench 10
 - Quartus II 35
 - Floorplan 36
 - RTL viewer 36
 - Qsys 727
 - TimeQuest 28
 - Timing simulation 38
- AMD 8
- Arbitrary rate conversion 345–374
- Arctan approximation 143
- ARM922T μ P 686
- ARM Cortex-A9 689
- Audio compression 613
 - MP3 626
 - MPEG 625
- Bartlett window 189, 419
- Batcher sorting 774
- Bijective 319
- Bison μ P tool 661, 672
- Bitreverse 465, 725
- Bit voting 775
- Blackman window 189, 419
- Blowfish 510
- B-spline rate conversion 362

- Butterfly 440, 442
- Canny edge detector 752
- CAST 510
- C compiler 680, 681
- Chebyshev series 142
- Chirp- z algorithm 424
- CIC filter 318–334
 - compensation filter 334
 - RNS design 320
 - interpolator 415
- Coding bounds 481
- Codes
 - block
 - decoders 483
 - encoder 482
 - convolutional
 - comparison 494
 - complexity 493
 - decoder 487, 491
 - encoder 487, 492
 - tree codes 486
- Contour plot 548
- Convergence 546, 547, 549
 - time constant 548
- Convolution
 - Bluestein 466
 - cyclic 465
 - linear 127, 179
- Cooley–Tuckey FFT 442
- CORDIC algorithm 131–141
- cosine approximation 148
- Costas loop
 - architecture 528
 - demodulation 527
 - implementation 529
- CPLD 6, 5
- Cryptography 494–510
- Cumulative histogram 775
- Custom instruction 721, 724
- Cypress 8
- Daubechies 380, 385, 396, 403, 412
- Data encryption standard (DES)
 - 503–510
 - DCT
 - definition 461
 - fast implementation 464
 - 2D 462
 - JPEG 462
 - Decimation 305
 - Decimator
 - CIC 321
 - IIR 246
 - Demodulator 515
 - Costas loop 527
 - I/Q generation 517
 - zero IF 518
 - PLL 523
 - De-noising 405
 - DFT
 - definition 418
 - inverse 418
 - filter bank 375
 - Rader 438
 - real 421
 - Winograd 434
 - Digital signal processing (DSP) 2, 126
 - Dimension reduction 590
 - Discrete
 - Cosine transform, *see DCT 464*
 - Fourier transform, *see DFT 418*
 - Hartley transform 468
 - Sine transform (DST) 461
 - Wavelet transform (DWT) 398–403
 - LISA μ P 706
 - Distributed arithmetic 127–132
 - Optimization
 - Size 131
 - Speed 132
 - signed 204
 - Divider 93–109
 - array
 - performance 106
 - size 107
 - convergence 103
 - fast 101

- LPM 106
- nonperforming 99, 171
- nonrestoring 100, 171
- restoring 96
- types 95
- Dyadic DWT 399
- EASI algorithm 605
- Edge detector
 - Canny 752
 - difference of Gaussian 751
 - Laplacian of the Gaussian 750
 - Prewitt 750
 - Sobel 750
- Eigenfrequency 319
- Eigenvalues ratio 552, 558, 559, 582
- Eigenvector 589
 - direct computation 591
 - power method 593
 - Oja learning 594
- Electrocardiogram 592
- Encoder 482, 487, 492
- Error
 - control 475–494
 - cost functions 539
 - residue 542
- Exponential approximation 152
- Farrow rate conversion 358
- Fast RLS algorithm 586
- FFT
 - comparison 456
 - Good–Thomas 437
 - group 440
 - Cooley–Tukey 442
 - in-place 457
 - IP core 458
 - index map 436
 - Nios co-processor 722
 - Radix- r 440
 - rate conversion 347
 - stage 440
 - Winograd 452
- Filter 179–303
 - cascaded integrator comb (CIC) 318–334
 - CIC compensation 334
 - causal 185
 - CSD code 193
 - conjugate mirror 389
 - difference of Gaussian 751
 - distributed arithmetic (DA) 204
 - finite impulse response (FIR) 179–214
 - frequency sampling 342
 - infinite impulse response (IIR) 225–303
 - IP core 215
 - Laplacian of the Gaussian 750
 - lattice 390
 - median 773
 - polyphase implementation 310
 - Prewitt 750
 - signed DA 204
 - Sobel 750
 - symmetric 186
 - 2D 753
 - transposed 181
 - recursive 345
- Filter bank
 - constant
 - bandwidth 395
 - Q 395
 - DFT 375
 - two-channel 380–394
 - aliasing free 383
 - Haar 382
 - lattice 390
 - linear-phase 393
 - lifting 387
 - QMF 380
 - orthogonal 389
 - perfect reconstruction 382
 - polyphase 389
 - mirror frequency 380
 - comparison 394
- Filter design
 - Butterworth 232

- Chebyshev 233
- Comparison of FIR to IIR 226
- elliptic 232
- equiripple 192
- frequency sampling 342
- Kaiser window 188
- Parks–McClellan 191
- Finite impulse response (FIR), *see* *Filter 179–214*
- Fixed-point arithmetic 106
- Flex μ P tool 661, 665
- Flip-flop
 - LPM 17, 33, 83, 83, 83
- Floating-point
 - 754 standard 77, 79
 - addition 115
 - arithmetic 109, 120
 - conversion to fixed-point 111
 - division 116
 - LPM blocks 123
 - multiplication 113
 - numbers 75
 - reciprocal 117
 - rounding 78
 - synthesis results 125
 - VHDL-2008 124
- Floorplan 36
- FPGA
 - Altera's Cyclone IV E 22
 - architecture 6
 - benchmark 9
 - design compilation 35
 - floor plan 36
 - graphical design entry 36
 - performance analysis 40
 - power dissipation 12
 - registered performance 40
 - routing 5, 25, 26
 - simulation 37
 - size 22, 22
 - technology 8
 - timing 27
 - waveform files 47
 - Xilinx Spartan-6 22
- FPL, *see* *FPGA and CPLD*
- Fractal 402
- Fractional delay rate conversion 349
- Frequency
 - sampling filter 342
 - synthesizer 32
- Function approximation
 - arctan 143
 - cosine 148
 - Chebyshev series 142
 - exponential 152
 - logarithmic 156
 - sine 148
 - square root 161
 - Taylor series 132
- Galois Field 480
- Gauss primes 73
- General-purpose μ P 632, 682
- Generator 71, 72
- Gibb's phenomenon 188
- Good–Thomas FFT 437
- Goodman/Carey half-band filter 339, 384, 412
- Gradient 545
- H26x 789, 790
- Half-band filter
 - decimator 340
 - factorization 383
 - Goodman and Carey 339, 384
 - definition 339
- Hamming window 189, 419
- Hann window 189, 419
- Harvard μ P 652
- Hogenauer filter, *see* *CIC*
- Homomorphism 318
- IDEA 510
- Identification 537, 551, 560
- Isomorphism 318
- Image
 - Canny edge detector 752
 - compression 462, 743

- edge detection 748
- format 743
- γ correction 746
- JPEG 743
- JPEG 2000 746
- median filter 773
- morphological operations 749
- Independent Component Analysis 601
- EASI algorithm 605
- Herault and Jutten method 601
- Index 71
- multiplier 72
- maps in FFTs 436
- Infinite impulse response (IIR) filter 225–303
- finite wordlength effects 239, 255
- fast filtering using
 - time-domain interleaving 241
 - clustered look-ahead pipelining 243
 - scattered look-ahead pipelining 244
 - decimator design 246
 - parallel processing 247
- narrow filter
 - BiQuad 261
 - allpass lattice 271,
 - cascade 261
 - lattice WDF 295
 - parallel 265
 - RNS design 250
- In-place 457
- Instruction set design 638
- Intel 633
- Intellectual Property (IP) core 40
- FFT 458
- FIR filter 215
- Image processing blocks 771
- NCO 40
- Interference cancellation 535, 579
- Interpolation
 - CIC 415
 - *see rate conversion*
- Inverse
 - multiplicative 437
 - system modeling 536
- JPEG, *see Image compression*
- Kaiser
 - window 419
 - window filter design 189
- Kalman gain 577, 580, 583
- Kronecker product 452
- Kurtosis 539
- Lattice
 - Gray and Markel 279
 - IIR filter 271
 - Semiconductor 7, 13
 - WDF 280
- Learning curves 551
- RLS 577, 580
- Lexical analysis (*see Flex*)
- LISA μ P 661, 706–722
- Lifting 387
- Linear feedback shift register 495
- LMS algorithm 546, 588
- normalized 554, 554
- design 563,
- pipelined 565
- delayed 565
- design 568
- look-ahead 567
- transposed 568
- block FFT 557
- simplified 573, 574
- error floor 574
- Logarithmic approximation 156
- LPM
 - add_sub 81, 301
 - divider 106, 116
 - multiplier 89
 - RAM 701
 - ROM 48
- LPC-10e method 625

MAC 83

Magnitude 165, 751

Mean absolute difference 783

Median filter 773

MicroBlaze μ P 698

Microprocessor

- Accumulator 647, 651

- Bison tool 661, 672

- C compiler 680, 681

- DWT 706

- GPP 632, 682

- Instruction set design 638

- Profile 707, 711, 714, 719

- Intel 633

- FFT co-processor 721

- Flex tool 661, 665

- Lexical analysis (*see Flex*)

- LISA 661, 706–722

- Hardcore

- PowerPC 685

- ARM922T 686

- ARM Cortex-A9 689

- Harvard 652

- Softcore

- MicroBlaze 698

- Nios 694

- Nios II 700, 721, 769, 782

- PicoBlaze 632, 690

- Media processor 777

- Parser (*see Bison*)

- PDSP 2, 14, 124, 645, 712

- RISC 634

- register file 653

- Stack 647, 651, 701

- Super Harvard 652

- Three address 649, 651

- Two address 649, 651

- Vector 716

- Von-Neuman 652

μ -law 613

ModelSim 37, 38

Moments 539

MOMS rate conversion 367

Multiplier

- adder graph 199, 239

- array 87

- block 90

- Booth 169

- complex 170, 442

- FPGA array 88

- floating-point 113, 125

- half-square 91

- index 72

- LPM 89

- performance 89

- QRNS 73

- quarter square 93, 250

- size 90

Modulation 511

- using CORDIC 514

Modulo

- adder 72

- multiplier 72

- reconstruction 334

Motion

- detection 783

- vector 784

MPEG 625, 789, 790

NAND 5, 47

NCO IP core 43

Nios μ P 694

Nios II 700, 721, 769, 782

Number representation

- canonical signed digit (CSD) 62, 239

- diminished by one (D1) 59

- fixed 106

- floating-point 75

- fractional 63, 193

- one's complement (1C) 59

- two's complement (2C) 59

- sign magnitude (SM) 59

Oja learning 594

Order filter 180

Ordering, *see index map*

Orthogonal

- wavelet transform 385
- filter bank 389
- Parser (*see* *Bison*)
- Perfect reconstruction 382
- Phase-locked loop (PLL)
 - with accumulator reference 518
 - demodulator 524
 - digital 525
 - implementation 525, 526
 - linear 523
- PicoBlaze μ P 632, 690
- Plessey ERA 5
- Pole/zero diagram 246, 389
- Polynomial rate conversion 356
- Polyphase representation 310, 386
- Power
 - dissipation 29
 - estimation 554, 554
 - line hum 543, 544, 548, 550, 562, 573
 - method 593
- PowerPC μ P 685
- Prediction 535
 - forward 583
 - backward 584
- Primitive element 71
- Programmable signal processor 2, 14, 124, 645, 712
 - addressing generation 644
- Public key systems 510
- Principle Component Analysis 589
 - Sanger's GHA 595
- Quadratic RNS (QRNS) 73
- Quadrature Mirror Filter (QMF) 380
- Quartus II 35
- Qsys 727
- Rader DFT 438
- Rate conversion
 - arbitrary 345–374
 - B-spline 362
 - Farrow 358
 - FFT-based 347
 - fractional delay 349
 - MOMS 367
 - polynomial 356
 - rational 309
- Rational rate conversion 309
- RC5 510
- Rectangular window 189, 419
- Reduced adder graph 199, 239
- RISC μ P 634
 - register file 653
- RLS algorithm 575, 580, 586
- RNS
 - CIC filter 320
 - complex 74
 - IIR filter 250
 - Quadratic 73
 - scaling 334
- ROM
 - LPM 48
- RSA 510
- RTL viewer 36
- Sampling
 - Frequency 419
 - Time 419
 - *see* *rate conversion*
- Sea of gates Plessey ERA 5
- Self-similar 399
- Sine approximation 148
- Simulator
 - ModelSim 37, 38
- Speech compression 613
 - ADPCM 618
 - A-law 613
 - LPC-10e method 625
- Square root approximation 161
- Stack μ P 647, 651, 701
- Step size 549, 550, 559
- Subband filter 375
- Super Harvard μ P 652
- Symmetry
 - in filter 186
 - in cryptographic algorithms 510

Synthesizer

- accumulator 32
- PLL with accumulator 518

Taylor series 132

Theorem

- Chinese remainder 71

Three address μ P 649, 651

TimeQuest 28

Timing 27

Two-channel filter bank 380–394

- comparison 394
- lifting 387
- orthogonal 389
- QMF 389
- polyphase 386

Transformation

- continuous Wavelet 399
- discrete cosine 464
- discrete Fourier 418
- inverse (IDFT) 418
- discrete Hartley 468
- discrete Wavelet 398–403
- domain LMS 557
- Fourier 419
- short-time Fourier (STFT) 395
- discrete sine 461

Triple DES 508

Two address μ P 649, 651

Vector μ P 716

Verilog

- key words 883

VGA 755

VHDL

- styles 17
- key words 883

Video processing

- Motion detection 783
- standard H26x/MPEG 789, 790

Von-Neuman μ P 652

Walsh 531

Wave digital filter 280

- 3-port 281

Wavelets 398–403

- continuous 399
- de-noising 405
- linear-phase 393
- LISA processor 706–722
- orthogonal 385

Widrow–Hoff LMS algorithm 546

Wiener–Hopf equation 542

Windows 189, 419

Winograd DFT algorithm 434

Winograd FFT algorithm 452

Wordlength

- IIR filter 239
- LWDF filter 300
- narrow band filter 295

Xilinx

- ARM Cortex-A9 689
- Atlys development board 20
- Devices 8, 11, 21
- Error correction decoder 479
- Fast carry adder 80
- Frequency sampling filter 345
- FIR filter design 219
- Hardcore FFTs 457
- ISIM 32
- MicroBlaze 698
- PicoBlaze μ P 632, 690
- PowerPC Hardcore 685
- PREP test bench 10
- Timing simulation 38
- XBLOCKS 345

Zech logarithm 72