References

2. S. Brown, Z. Vranesic: *Fundamentals of Digital Logic with VHDL Design*
   (McGraw-Hill, New York, 1999)
64. K. Kammeyer: “Digital Filter Realization in Distributed Arithmetic,” in *Proc. European Conf. on Circuit Theory and Design* (1976), Genoa, Italy
88. R. Saal: Handbook of Filter Design (AEG-Telefunken, Frankfurt, Germany, 1979)
References


118. Altera: “Farrow-Based Decimating Sample Rate Converter,” in Altera Application Note AN-347 San Jose (2004)


128. A. Gotchev, J. Vesma, T. Saramäki, K. Egiazarian: “Modified B-Spline Functions for Efficient Image Interpolation,” in First IEEE Balkan Conference on


133. I. Daubechies: Ten Lectures on Wavelets (Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992)


161. E. Brigham: *FFT*, 3rd edn. (Oldenbourg Verlag, München Wien, 1987)
163. R.E. Blahut: *Theory and practice of error control codes* (Addison-Wesley, Melo Park, California, 1984)
169. C. Rader: “Discrete Fourier Transform when the Number of Data Samples is Prime,” *Proceedings of the IEEE* **56**, 1107–8 (1968)
193. H. Park, V. Prasanna: “Modular VLSI Architectures for Computing the
Arithmetic Fourier Transform,” IEEE Transactions on Signal Processing 41
(6), 2236–2246 (1993)
using Signal Processor TMS320C25,” Master’s thesis, Institute for Data Tech-
nics, Darmstadt University of Technology
Implementable Cryptography Algorithms,” in 3rd International Symposium
197. U. Meyer-Bäse, R. Watzel: “An Optimized Format for Long Frequency Pag-
ing Systems,” in 3rd International Symposium on Communication Theory &
Applications (1995), pp. 78–79
in Computer Science 1142, 376–175 (1996)
199. R. Watzel (1993): “Design of Paging Scheme and Implementation of the Suit-
able Crypto-Controller using FPGAs,” Master’s thesis, Institute for Data Tech-
nics, Darmstadt University of Technology
for Error Correction in a Paging System,” Master’s thesis, Institute for Data Tech-
nics, Darmstadt University of Technology
201. U. Meyer-Bäse et al.: “Zum bestehenden Übertragungsprotokoll kompatible
112
nals using Programmable Gate Arrays,” Master’s thesis, Institute for Data Tech-
nics, Darmstadt University of Technology
204. D. Wiggert: Error-Control Coding and Applications (Artech House, Dedham,
Mass., 1988)
205. G. Clark, J. Cain: Error-Correction Coding for Digital Communications
(Plenum Press, New York, 1988)
pp. 977–980 (1973)
207. W. Funy, H. Riess: Kryptographie (R. Oldenbourg Verlag, München, 1988)
209. M. Langhammer: “Reed-Solomon Codec Design in Programmable Logic,”
509–516 (1978)
211. R. Bryant: “Graph-Based Algorithms for Boolean Function Manipulation,”
214. U. Meyer-Bäse: “Optimal Strategies for Incoherent Demodulation of Narrow
Band FM Signals,” in 3rd International Symposium on Communication The-
216. R. Johannesson: “Robustly Optimal One-Half Binary Convolutional Codes,”
238. H. Geschwind: *Einführung in die PLL-Technik* (Vieweg, Braunschweig, 1984)


References

321. C. Fraser, D. Hanson: A Retargetable C Compilers: Design and Implementation, 1st edn. (Addison-Wesley, Boston, 2003)
346. J. Ousterhout: Tcl and the Tk Toolkit, 1st edn. (Addison-Wesley, Boston, 1994)
347. M. Harrison, M. McLennan: Effective Tcl/Tk Programming, 1st edn. (Addison-Wesley, Reading, Massachusetts, 1998)
The first and second editions of the book include Verilog using the IEEE 1364-1995 standard [340]. This third edition takes advantage of several improvements that are documented in the IEEE 1364-2001 standard, which is available for about $100 from the IEEE bookstore [341, 342]. The Verilog 2001 improvements have now found implementations in all major design tools, [343, 344, 345] and we want to briefly to review the most important new features that are used. We only review the implemented new features; for all the new features see [341].

- The entity description in the 1364-1995 standard requires that (similar to the Kernighan and Ritchie C coding) all ports appear twice, first in the port list and then in the port data-type description, e.g.,

```verilog
module iir_pipe (x_in, y_out, clk); //----> Interface

    parameter W = 14; // Bit width - 1
    input       clk;
    input [W:0] x_in; // Input
    output [W:0] y_out; // Result

... 
```

Note that all ports (x_in, y_out, and clk) are defined twice. In the 1364-2001 standard (see Sect. 12.3.4 LRM) this duplication is no longer required, i.e., the new coding is done as follows:

```verilog
module iir_pipe //----> Interface
    #(parameter W = 14) // Bit width - 1
    (input       clk,
     input signed [W:0] x_in, // Input
     output signed [W:0] y_out); // Result
```

- Signed data types are available in the 1364-2001 standard, which allows arithmetic signed operations to be simplified. In the signal definition line the signed keyword is introduced after the input, output, reg or wire keywords, e.g.,

```verilog
reg signed [W:0] x, y;
```
Conversion between signed and unsigned type can be accomplished via the $signed or $unsigned conversion functions, see Sect. 4.5 LRM. For signed constants we introduce a small s or capital S between the hyphen and the base, e.g., 'sd90 for a signed constant 90. Signed arithmetic operations can be done using the conventional divide / or multiply * operators. For power-of-2 factors we can use the new arithmetic left <<< or right shift >>> operations, see Sect. 4.1.23 LRM. Note the three shift operations symbols used to distinguished to the unsigned shifts that use two shift symbols. Signed or zero extension is automatically done depending on the data type. From the IIR filter examples we can now replace the old-style operation:

```verilog
y <= x + [y[1:0],y[1:2]] + [2{y[1:0]}],y[1:2];
```

with the 1364-2001 Verilog style operations using the divide operator:

```verilog
y <= x + y / 'sd2 + y / 'sd4; // div with / uses 92 LEs.
```

Note the definition as signed divide for the constants, the code

```verilog
y <= x + y / 2 + y / 4;
```

will show incorrect simulation results in Quartus II in versions 4.0, 4.2, 5.0, 5.1 but works fine in our web edition 6.0. Alteratively we can use the arithmetic right shift operator to implement the divide by power-of-2 values.

```verilog
y <= x + (y >>> 1) + (y >>> 2); // div with >>> uses 60 LEs
```

It is evident that this notation makes the arithmetic operation much more readable than the old-style coding. Although both operations are functional equivalent, the Quartus synthesis results reveals that the divide is mapped to a different architecture and needs therefore more LEs (92 compared with 60 LEs) than used by the arithmetic shift operations. From the comparison to the VHDL synthesis data we conclude that 60 LEs is the expect result. This is the reason we will use the arithmetic left and right shift throughout the examples.

- The implicit event_expression list allows one to add automatically all right-hand-side variables to be added to the event expression, i.e., from the bfproc examples

```verilog
always @(Are or Bre or Aim or Bim)
begin
...
```

we now simply use
always @(*)
begin
...

This reduces essentially the RTL simulation errors due to missing variables in the event listing, see Sect. 9.7.5 LRM for details.

- The generate statement introduced in the 1364-2001 Verilog standard allows one to instantiate several components using a single generate loop construct. The LRM Sects. 12.1.3.2-4 show eight different generate examples. We use the generate in the fir_gen, fir_lms, and fir6dlms files. Note that you also need to define a genvar as a loop variable used for the generate statement, see Sect. 12.1.3.1 LRM for details.

The 1364-2001 Verilog standard introduces 21 new keywords. We use the new keywords endgenerate, generate, genvar, signed, and unsigned. We have not used the new keywords:

- automatic, cell, config, design, endconfig, incdir, include, instance, liblist, library, localparam, noshowcancelled, pulsestyle_onevent, pulsestyle_ondetect, showcancelled, use

The next pages contain the Verilog 1364-2001 code of all design examples. The old style Verilog 1364-1995 code can be found in [57]. The synthesis results for the examples are listed on page 731.

//*********************************************************
// IEEE STD 1364-2001 Verilog file: example.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
//'include "220model.v" // Using predefined components

module example //----> Interface
#@parameter WIDTH =8 // Bit width
(input clk,
 input [WIDTH-1:0] a, b, op1,
 output [WIDTH-1:0] sum, d);

wire [WIDTH-1:0] c; // Auxiliary variables
reg [WIDTH-1:0] s; // Infer FF with always
wire [WIDTH-1:0] op2, op3;

wire clkena, ADD, ena, aset, sclr, sset, aload, sload,
aclr, ovf1, cin1; // Auxiliary lpm signals

// Default for add:
assign cin1=0; assign aclr=0; assign ADD=1;
assign ena = 1; assign aclr = 0; assign aset = 0;
assign sclr = 0; assign sset = 0; assignaload = 0;
assign sload = 0; assign clkena = 0; // Default for FF

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

// Note when using 220model.v ALL component’s signals
// must be defined, default values can only be used for
// the parameters.

lpm_add_sub add1 //----> Component instantiation
                ( .result(op3), .dataa(op1), .datab(op2)); // Used ports
                // .cin(cin1),.cout(cr1), .add_sub(ADD), .clken(clkena),
                // .clock(clk), .overflow(oval1), .aclr(aclr)); // Unused
                defparam add1.lpm_width = WIDTH;
                defparam add1.lpm_representation = "SIGNED";

lpm_ff reg1
( .data(op3), .q(sum), .clock(clk)); // Used ports
                // .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
                // .sset(sset), .aload(aload), .sload(sload)); // Unused
                defparam reg1.lpm_width = WIDTH;

assign c = a + b; //----> Continuous assignment statement

always @(posedge clk) //----> Behavioral style
begin : p1 // Infer register
s = c + s; // Signal assignment statement
end
assign d = s;
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
//’include "220model.v"

module fun_text //----> Interface
    #(parameter WIDTH = 32) // Bit width
    (input    clk,
A. Verilog Source Code 2001

```verilog
input [WIDTH-1:0] M,
output [7:0] sin, acc;

wire [WIDTH-1:0] s, acc32;
wire [7:0] msbs;              // Auxiliary vectors
wire ADD, ena, aset, sclr, sset; // Auxiliary signals
wire aset, sclerosis, aclr, ovf1, cin1, clkena;

// Default for add:
assign clkena=0; assign cin1=0; assign ADD=1;
// default for FF:
assign ena=1; assign aclr=0; assign aset=0;
assign sclr=0; assign sset=0; assign aset=0;
assign aclr=0;

lpm_add_sub add_1 // Add M to acc32
    ( .result(s), .dataa(acc32), .datab(M)); // Used ports
    // .cout(c1), .add_sub(ADD), .overflow(ovf1), // Unused
    // .clock(clk), cin(cin1), .clkena(clkena), .aclr(aclr));
//
defparam add_1.lpm_width = WIDTH;
defparam add_1.lpm_representation = "UNSIGNED";

lpm_ff reg_1 // Save accu
    ( .data(s), .q(acc32), .clock(clk)); // Used ports
    // .enable(ena), .aclr(aclr), .aset(aset), // Unused ports
    // .sset(sset), .asload(aload), .sload(sload), .sclr(sclr));
defparam reg_1.lpm_width = WIDTH;
assign msbs = acc32[WIDTH-1:WIDTH-8];
assign acc = msbs;

lpm_rom rom1
    ( .q(sin), .inclock(clk), .outclock(clk),
      .address(msbs)); // Used ports
//    .memenab(ena) ); // Unused port
defparam rom1.lpm_width = 8;
defparam rom1.lpm_widthad = 8;
defparam rom1.lpm_file = "sine.mif";

endmodule

//*********************************************************
// IEEE STD 1364-2001 Verilog file: cmul7p8.v
```

A. Verilog Source Code 2001
module cmul7p8 // ------> Interface
(input signed [4:0] x,
output signed [4:0] y0, y1, y2, y3);

assign y0 = 7 * x / 8;
assign y1 = x / 8 * 7;
assign y2 = x/2 + x/4 + x/8;
assign y3 = x - x/8;
endmodule

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, // 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 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

//'include "220model.v"

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
(output [WIDTH-1:0] x, y, // Inputs
output [WIDTH-1:0] sum, // Result
output LSBs_Carry, MSBs_Carry, // Single bits
input clk); // 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
v2 <= q1[WIDTH1] + {1’b0,q2[WIDTH2-1:0]}; // Add result from middle bits (x+y) and carry from LSBs
v3 <= q2[WIDTH2] + q3; // Add result from MSBs bits (x+y) and carry from middle

 Third stage of the adder
s1 <= v1; // Save v1
s2 <= v2[WIDTH2-1:0]; // Save v2
s3 <= v2[WIDTH2] + v3; // Add result from MSBs bits (x+y) and 2. carry from middle

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 stages
// 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
     include "220model.v"

}
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] l0, l1, r0, v0, s0; // LSBs of inputs
reg [WIDTH0:0] q0; // LSBs of inputs
reg [WIDTH1-1:0] l2, l3, r1, s1; // 2. LSBs of input
reg [WIDTH1:0] v1, q1; // 2. LSBs of input
reg [WIDTH2-1:0] l4, l5, s2, h7; // 2. MSBs bits
reg [WIDTH2:0] q2, v2, r2; // 2. MSBs bits
reg [WIDTH3-1:0] l6, l7, 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
  l0[WIDTH0-1:0] <= x[WIDTH0-1:0];
  l1[WIDTH0-1:0] <= y[WIDTH0-1:0];
  // Split 2. LSBs from input x,y
  l2[WIDTH1-1:0] <= x[WIDTH1-1+WIDTH0:WIDTH0];
  l3[WIDTH1-1:0] <= y[WIDTH1-1+WIDTH0:WIDTH0];
  // Split 2. MSBs from input x,y
  l4[WIDTH2-1:0] <= x[WIDTH2-1+WIDTH0:WIDTH0];
  l5[WIDTH2-1:0] <= y[WIDTH2-1+WIDTH0:WIDTH0];
  // Split MSBs from input x,y
  l6[WIDTH3-1:0] <= x[WIDTH3-1+WIDTH0:WIDTH0];
  l7[WIDTH3-1:0] <= y[WIDTH3-1+WIDTH0:WIDTH0];

  //************* 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 **********************
\[ \text{r0} \gets v0; \quad \text{// Delay for LSBs} \]
\[ \text{r1} \gets v1[\text{WIDTH1-1:0}]; \quad \text{// Delay for 2. LSBs} \]
// Add result from 2. MSBs (x+y) and carry from 2. LSBs
\[ \text{r2} \gets v1[\text{WIDTH1}] + \{1'b0, v2[\text{WIDTH2-1:0}]\}; \]
// Add result from MSBs (x+y) and carry from 2. MSBs
\[ \text{r3} \gets v2[\text{WIDTH2}] + v3; \]

//*************** Fourth stage of the adder **********************
\[ \text{s0} \gets \text{r0}; \quad \text{// Delay for LSBs} \]
\[ \text{s1} \gets \text{r1}; \quad \text{// Delay for 2. LSBs} \]
\[ \text{s2} \gets \text{r2[WIDTH2-1:0]}; \quad \text{// Delay for 2. MSBs} \]
// Add result from MSBs (x+y) and carry from 2. MSBs
\[ \text{s3} \gets \text{r2[WIDTH2]} + \text{r3}; \]
end

assign \text{LSBs\_Carry} = \text{q0[WIDTH1]}; \quad \text{// Provide test signals}
assign \text{Middle\_Carry} = \text{v1[WIDTH1]};
assign \text{MSBs\_Carry} = \text{r2[WIDTH2]};

// Build a single output word of
// \text{WIDTH} = \text{WIDTH0 + WIDTH1 + WIDTH2 + WIDTH3}
assign \text{sum} = \{\text{s3, s2, s1, s0}\}; \quad \text{// Connect sum to output}
endmodule
s <= s0;
else
  case (s)
    s0 : begin // Initialization step
      a_reg <= a;
      s <= s1;
      count = 0;
      p <= 0; // Product register reset
      t <= x; // Set temporary shift register to x
    end
    s1 : begin // Processing step
      if (count == 7) // Multiplication ready
        s <= s2;
      else
        begin
          if (a_reg[0] == 1) // Use LSB for bit select
            p <= p + t; // Add 2^k
          a_reg <= a_reg >>> 1;
          t <= t <<< 1;
          count = count + 1;
          s <= s1;
        end
      end
    s2 : begin // Output of result to y and
      y <= p; // start next multiplication
      s <= s0;
    end
  endcase
end

/***************************************************************************/
// IEEE STD 1364-2001 Verilog file: div_res.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
/***************************************************************************/
// Restoring Division
/***************************************************************************/
module div_res(
  input clk, reset,
  input [7:0] n_in,
  input [5:0] d_in,
output reg [5:0] r_out,
output reg [7:0] q_out);

parameter s0=0, s1=1, s2=2, s3=3; // State assignments

// Divider in behavioral style
always @(posedge clk or posedge reset)
begin : F // Finite state machine
reg [3:0] count;
reg [1:0] s; // FSM state
reg [13:0] d; // Double bit width unsigned
reg signed [13:0] r; // Double bit width signed
reg [7:0] q;

if (reset) // Asynchronous reset
s <= s0;
else

  case (s)
    s0 : begin // Initialization step
      s <= s1;
      count = 0;
      q <= 0; // Reset quotient register
      d <= d_in << 7; // Load aligned denominator
      r <= n_in; // Remainder = numerator
    end
    s1 : begin // Processing step
      r <= r - d; // Subtract denominator
      s <= s2;
    end
    s2 : begin // Restoring step
      if (r < 0) begin // Check r < 0
        r <= r + d; // Restore previous remainder
        q <= q << 1; // LSB = 0 and SLL
      end
      else
        q <= (q << 1) + 1; // LSB = 1 and SLL
      count = count + 1;
      d <= d >> 1;
    end
    s3 : begin // Division ready?
      s <= s3;
    end
  endcase
end
module div_aegp
(input clk, reset,
 input [8:0] n_in,
 input [8:0] d_in,
 output reg [8:0] q_out);

always @(posedge clk or posedge reset) //-> Divider in
begin : States // behavioral style
 parameter s0=0, s1=1, s2=2;
 reg [1:0] count;
 reg [1:0] state;
 reg [9:0] x, t, f; // one guard bit
 reg [17:0] tempx, tempt;

 if (reset) // Asynchronous reset
 state <= s0;
 else
 case (state)
 s0 : begin // Initialization step
 state <= s1;
 count = 0;
 t <= {1'b0, d_in}; // Load denominator
 x <= {1'b0, n_in}; // Load numerator
 end
 s1 : begin // Processing step

 s3 : begin // Output of result
 q_out <= q[7:0];
 r_out <= r[5:0];
 s <= s0; // 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
// OR: Numerator / Denominator = Quotient and Remainder
// Numerator = Quotient * Denominator + Remainder

s3 : begin // Output of result
 q_out <= q[7:0];
 r_out <= r[5:0];
 s <= s0; // Start next division
 end
endcase
end
module cordic #(parameter W = 7) // Bit width - 1
(input clk,
 input signed [W:0] x_in, y_in,
 output reg signed [W:0] r, phi, eps);

// There is bit access in Quartus array types
// in Verilog 2001, therefore use single vectors
// but use a separate line 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 clk) begin //----> Infer registers
 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
  end
else
  begin
  end

if (y[2] >= 0) // Rotate 14 degrees
  begin
  end
else
  begin
  end

  r  <= x[3];
  phi <= z[3];
  eps <= y[3];
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,
   input signed [W-1:0] x_in,
   //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,
   output reg signed [W-1:0] f_out);
  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) begin
    x <= x_in; // FF for input and output
    f_out <= f;
  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[2] = ((x * d[3]) / 128) - d[4];
  assign f   = ((x * d[1]) / 256) - d[2];
module ln #(parameter N = 5, // -- Number of coefficients-1
    parameter W= 17) // -- Bitwidth -1
(input clk,
    input signed [W:0] x_in,
    output reg signed [W:0] f_out);

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[3] = 18’sd18601;

always @(posedge clk)
begin : Store
    x <= x_in; // Store input in register
end

always @(posedge clk)  // 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_out <= 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, reset,
 input [16:0] x_in,
 output [16:0] a_o, imm_o, f_o,
 output reg [2:0] ind_o,
 output reg [1:0] count_o,
 output [16:0] x_o, pre_o, post_o,
 output reg [16:0] f_out);

// 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 [2:0] s, op;
reg [16:0] x; // Auxiliary
reg signed [16:0] a, b, f, imm; // ALU data
reg [16:0] pre, post;
// Chebychev poly coefficients for 16-bit precision:
wire signed [16:0] p [0:4];

assign p[0] = 7563;
assign p[1] = 42299;
assign p[3] = 15813;

always @(posedge reset or posedge clk) //------> SQRT FSM
begin : States  // sample at clk rate
  reg signed [3:0] ind;
  reg [1:0] count;

  if (reset)  // Asynchronous reset
    s <= start;
  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 == 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;
      end  // start next cycle
    endcase
  end
  ind_o <= ind;
  count_o <= count;
end

always @(posedge clk)  // Define the ALU operations
begin : ALU
case (op)
  load : f <= imm;
  mac : f <= (a * f / 32768) + imm;
  scale : f <= a * f;
  denorm : f <= (a * f /32768);
  nop : f <= f;
  default : f <= f;
endcase
end

always @(*
begin : EXP
  reg [16:0] slv;
  reg [16:0] po, pr;
  integer K, L;

  slv = x_in;
  // Compute pre-scaling:
  for (K=0; K <= 15; K= K+1)
    if (slv[K] == 1)
      L <= K;
  pre = 1 << (14-L);
  // 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 +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;
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
     Mpipe = 3) // Pipeline steps of multiplier
(input clk, Load_x, // std_logic
 input signed [W1-1:0] x_in, c_in, // Inputs
 output signed [W4-1:0] y_out); // Results

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
wire signed [W2-1:0] sum; // Auxilary signals
wire clken, aclr;

assign sum=0; assign aclr=0; // Default for mult
assign clken=0;

//----> Load Data or Coefficient
always @ (posedge clk)
begin: Load
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 begin
  x <= x_in; // Get one data sample at a time
end
//----> Compute sum-of-products
always @(posedge clk)
begin: SOP
  // Compute the transposed filter additions
  a[0] <= p[0] + a[1];
  a[3] <= p[3]; // First TAP has only a register
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 pipelined multiplier
    lpm_mult mul_I // Multiply x*c[I] = p[I]
      (.clock(clk), .dataa(x), .datab(c[I]), .result(p[I])); // Unused ports
    defparam mul_I.lpm_widtha = W1;
    defparam mul_I.lpm_widthb = W1;
    defparam mul_I.lpm_widthp = W2;
    defparam mul_I.lpm_widths = W2;
    defparam mul_I.lpm_pipeline = Mpipe;
    defparam mul_I.lpm_representation = "SIGNED";
  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,
   input signed [7:0] x,
   output reg signed [7:0] y);

  // Tapped delay line array of bytes
  reg signed [7:0] tap [0:3];
  // For bit access use single vectors in Verilog
integer I;

always @(posedge clk) //-----> Behavioral style
begin : p1
    // Compute output y with the filter coefficients weight.
    // The coefficients are [-1 3.75 3.75 -1].
    // Multiplication and division for Altera MaxPlusII can
    // be done in Verilog 2001 with signed shifts!

    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
endmodule

module dafsm //--> Interface
    (input clk, reset,
    input [2:0] x_in0, x_in1, x_in2,
    output [2:0] lut,
    output reg [5:0] y);

    reg [2:0] x0, x1, x2;
    wire [2:0] table_in, table_out;

    reg [5:0] p; // temporary register

    assign table_in[0] = x0[0];
    assign table_in[1] = x1[0];
    assign table_in[2] = x2[0];

    always @(posedge clk or posedge reset)
    begin : DA //----> DA in behavioral style
        parameter s0=0, s1=1;
        reg [0:0] state;
    end
endmodule
reg [1:0] count; // Counts the shifts

if (reset) // Asynchronous reset
    state <= s0;
else
    case (state)
        s0 : begin // Initialization
            state <= s1;
            count = 0;
            p <= 0;
            x0 <= x_in0;
            x1 <= x_in1;
            x2 <= x_in2;
        end
        s1 : begin // Processing step
            if (count == 3) begin // Is sum of product done?
                y <= p; // Output of result to y and
                state <= s0; // start next sum of product
            end
            else begin
                p <= (p >> 1) + (table_out << 2); // p/2+table*4
                x0[0] <= x0[1];
                x0[1] <= x0[2];
                x1[0] <= x1[1];
                x1[1] <= x1[2];
                x2[0] <= x2[1];
                x2[1] <= x2[2];
                count = count + 1;
                state <= s1;
            end
        endcase
end

case3 LC_Table0
( .table_in(table_in), .table_out(table_out));

assign lut = table_out; // Provide test signal

endmodule

//***********************************************************************
// IEEE STD 1364-2001 Verilog file: case3.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//***********************************************************************
module case3
(input [2:0] table_in, // Three bit
 output reg [2:0] table_out); // Range 0 to 6

// 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 = 5;
    4 : table_out = 1;
    5 : table_out = 3;
    6 : table_out = 4;
    7 : table_out = 6;
    default : ;
  endcase
end
endmodule

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;
        4 : table0out01 = 14;
module darom //--> Interface
(input clk, reset,
 input [2:0] x_in0, x_in1, x_in2,
 output [2:0] lut,
 output reg [5:0] y);

reg [2:0] x0, x1, x2;
wire [2:0] table_in, table_out;

reg [5:0] p; // Temporary register
wire ena;

assign ena=1;
assign table_in[0] = x0[0];
assign table_in[1] = x1[0];
assign table_in[2] = x2[0];

always @(posedge clk or posedge reset)
begin : DA //----> DA in behavioral style
    parameter s0=0, s1=1;
    reg [0:0] state;
    case (state)
        0b00: state = s0;
        0b01: state = s1;
        default: state = s0;
    endcase
end
endmodule
reg [1:0] count; // Counts the shifts

if (reset) // Asynchronous reset
    state <= s0;
else
    case (state)
        s0 : begin // Initialization
            state <= s1;
            count = 0;
            p <= 0;
            x0 <= x_in0;
            x1 <= x_in1;
            x2 <= x_in2;
        end
        s1 : begin // Processing step
            if (count == 3) begin // Is sum of product done?
                y <= (p >> 1) + (table_out << 2); // Output to y
                state <= s0; // and start next sum of product
            end
            else begin
                p <= (p >> 1) + (table_out << 2);
                x0[0] <= x0[1];
                x0[1] <= x0[2];
                x1[0] <= x1[1];
                x1[1] <= x1[2];
                x2[0] <= x2[1];
                x2[1] <= x2[2];
                count = count + 1;
                state <= s1;
            end
        end
    default : ;
    endcase
end

lpm_rom rom_1 // Used ports:
    ( .outclock(clk), .address(table_in), .q(table_out));
    // .inclock(clk), .memenab(ena)); // Unused
    defparam rom_1.lpm_width = 3;
    defparam rom_1.lpm_widthad = 3;
    defparam rom_1.lpm_outdata = "REGISTERED";
    defparam rom_1.lpm_address_control = "UNREGISTERED";
    defparam rom_1.lpm_file = "darom3.mif";
assign lut = table_out; // Provide test signal

endmodule

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

module dasign //-> Interface
(input clk, reset,
 input signed [3:0] x_in0, x_in1, x_in2,
 output [3:0] lut,
 output reg signed [6:0] y);

reg signed [3:0] x0, x1, x2;
wire signed [2:0] table_in;
wire signed [3:0] table_out;

reg [6:0] p; // Temporary register

assign table_in[0] = x0[0];
assign table_in[1] = x1[0];
assign table_in[2] = x2[0];

always @(posedge clk or posedge reset)// DA in behavioral
begin : DA // style
    parameter s0=0, s1=1;
    integer k;
    reg [0:0] state;
    reg [2:0] count; // Counts the shifts

    if (reset) // Asynchronous reset
        state <= s0;
    else
        case (state)
            s0 : begin // Initialization step
                state <= s1;
                count = 0;
                p <= 0;
                x0 <= x_in0;
                x1 <= x_in1;
                x2 <= x_in2;
            end

            default : begin // Reset state
                state <= s0;
                count = 0;
                p <= 0;
                x0 <= x_in0;
                x1 <= x_in1;
                x2 <= x_in2;
            end
        endcase
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
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;
  endcase
end
module dapara //----> Interface

(input clk,
input signed [3:0] x_in,
output reg signed[6:0] y);

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) //----> DA in behavioral style
begin : DA
    integer k, l;
    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

//*********************************************************
// IEEE STD 1364-2001 Verilog file: dapara.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
'include "case3s.v" // User-defined component
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 signed [W:0] x_in, // Input
      output signed [W:0] y_out, // Result
      input clk);

    reg signed [W:0] x, y;

    // initial begin
    // y=0;
    // x=0;
    // end

    // Use FFs for input and recursive part
    always @(posedge clk) begin // Note: there is a signed
        x <= x_in; // integer in Verilog 2001
        y <= x + (y >>> 1) + (y >>> 2); // >>> uses fewer LEs
    // y <= x + y / 2 + y / 4; // div with / uses more LEs
    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,
input signed [W:0] x_in, // Input
output signed [W:0] y_out); // Result

reg signed [W:0] x, x3, sx;
reg signed [W:0] y, y9;

always @(posedge clk) // Infer FFs for input, output and
begin // pipeline stages;
x <= x_in; // use nonblocking 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

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

endmodule

Không có thông tin xuất sắc cụ thể về phần còn lại của văn bản.
begin : Multiplex // and odd samples;
parameter even=0, odd=1; // recombine y at clk rate
reg [0:0] state;

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

always @(negedge clk_div2)
begin: Arithmetic
  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

endmodule

//*********************************************************
// IEEE STD 1364-2001 Verilog file: cic3r32.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
module cic3r32 //----> Interface
(input clk, reset,
input signed [7:0] x_in,
output signed [9:0] y_out,
output reg clk2);

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) // 3 integrator sections
begin : Int
x <= x_in;
i0 <= i0 + x;
i1 <= i1 + i0;
i2 <= i2 + i1;
end

always @(posedge clk) // 3 comb sections
begin : Comb
  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, reset,
   output reg clk2,
   input signed [7:0] x_in,
   output signed [9:0] y_out);
  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;
    end
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

always @(posedge clk) // 3 integrator sections
begin : Int
    x <= x_in;
    i0 <= i0 + x;
    i1 <= i1 + i0[25:5];
    i2 <= i2 + i1[20:5];
end

always @(posedge clk) // 3 comb sections
begin : Comb
    if (state == sample) begin
        c0 <= i2[15:2];
        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: db4poly.v
module db4poly //----> Interface
  (input clk, reset,
   output clk2,
   input signed [7:0] x_in,
   output signed [16:0] x_e, x_o, g0, g1, // Test signals
   output signed [8:0] y_out);

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) // Asynchronous reset
    state <= even;
  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 @(negedge clk_div2) // Infer registers;
begin : AddPolyphase // use nonblocking assignments
//---------- 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

// 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: 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, reset, // Clock + reset for the registers
input signed [7:0] x_in,
output [3:0] count_o,
output ena_in_o, ena_out_o, ena_io_o,
output signed [8:0] f0_o, f1_o, f2_o,
output signed [8:0] y_out;

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[6] = 29; assign c0[7] = -21; assign c0[8]=16;
assign c0[9] = -13; assign c0[10] = 11;

// filter coefficients for filter c2

always @(posedge reset or posedge clk)
begin : FSM // Control the system and sample at clk rate
  if (reset) // Asynchronous reset
count <= 0;
else begin
  if (count == 11) count <= 0;
else count <= count + 1;
end
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) //----> Input delay line
begin : INPUTMUX
  integer I; // loop variable

  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) //----> Output delay line
begin : OUTPUTMUX
  integer I; // loop variable

  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

always @(posedge clk) //----> One tapped delay line
begin : TAP // get 4 samples at one time
  integer I; // loop variable

  if (ena_io) begin
    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

always @(posedge clk) // Compute sum-of-products for f0
begin : SOP0
  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

  f0 <= sum >>> 8;
end

always @(posedge clk) // Compute sum-of-products for f1
begin : SOP1
  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

  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[OL]; // 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, reset, // Clock/reset for the registers
input signed [7:0] x_in,
output [3:0] count_o,
output ena_in_o, ena_out_o,
output signed [8:0] c0_o, c1_o, c2_o, c3_o,
output [8:0] d_out,
output reg signed [8:0] y_out);

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
d <= dnew;
end
end

always @(posedge clk)
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) //----> One tapped delay line
begin : TAP
  integer I; // loop variable

  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) // Get 4 samples at one time
begin : GET
  integer I; // loop variable

  if (ena_out) begin

for (I=0; I<=IL; I=I+1)
    x[I] <= ibuf[I]; // take over input buffer
end

// Compute sum-of-products:
always @\(\text{posedge clk}\) // Compute sum-of-products for \(f_0\)
begin : SOP
    reg\ signed [8:0]\ y; // temp's
    // Matrix multiplier \(iV=\text{inv}(\text{Vandermonde})\) \(c=iV*x(n-1:\text{n+2})'\)
    // \begin{bmatrix}
    //     x(0) & x(1) & x(2) & x(3) \\
    //     \hline
    //     0 & 1 & 0 & 0 \\
    //     -0.3333 & -0.5 & 1 & -0.1667 \\
    //     0.5 & -1 & 0.5 & 0 \\
    //     -0.1667 & 0.5 & -0.5 & 0.1667 \\
    // \end{bmatrix}
    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;
    y = c2 + ((c3 * d) >>> 8); // \(d\) is scale by 256
    y = ((y * d) >>> 8) + c1;
    y = ((y * d) >>> 8) + c0;
    y_out <= y; // Connect to output + store in register
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;
module cmoms #(parameter IL = 3) // Input buffer length -1
  (input clk, reset, // Clock/reset for registers
   input signed [7:0] x_in,
   output [3:0] count_o,
   output ena_in_o, ena_out_o,
   output signed [8:0] c0_o, c1_o, c2_o, c3_o, xiir_o,
   output signed [8:0] y_out);

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) //----> Behavioral Style
begin : IIR // Compute iir coefficients first
  reg signed [8:0] x1; // x * 1
  if (ena_in) begin
    xiir <= (3 * x1 >> 1) - (xiir >> 1);
    x1 = x_in;
  end
end

always @(posedge clk) //----> One tapped delay line
begin : TAP
  integer I; // loop variable
  if (ena_in) begin
    for (I=1; I<=IL; I=I+1)
      ibuf[I-1] <= ibuf[I]; // Shift one
    ibuf[IL] <= xiir; // Input in register IL
always @(posedge clk) //----> One tapped delay line
begin : GET // get 4 samples at one time
    integer I; // loop variable
    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) // Compute sum-of-products for f0
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 (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);
    end

// 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
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, reset,
output clk2,
input signed [7:0] x_in,
output signed [16:0] x_e, x_o,
output reg signed [8:0] g, h);

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) // Asynchronous reset
state <= even;
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;

            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
//*********************************************************
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 @(negedge clk_div2)
begin: LowerTreeFF
    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 @(negedge clk_div2)
begin: OutputScale
    g <= up1 >>> 8; // i.e. up1 / 256
    h <= low1 >>> 8; // i.e. low1 / 256;
end
endmodule

//*****************************************************************************
module rader7
//---> Interface
(input clk, reset,
input [7:0] x_in,
output reg signed [10:0] y_real, y_imag);

reg signed [10:0] accu; // Signal for X[0]
// Direct bit access of 2D vector in Quartus Verilog 2001
// possible 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]

always @(posedge clk or posedge reset) // State machine
begin : States // for RADER filter
parameter Start=0, Load=1, Run=2;
reg [1:0] state;
reg [4:0] count;

if (reset) // Asynchronous reset
state <= Start;
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;
      end
    end
  endcase
end // States
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
state <= Start; // Output of result
end // and start again
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;
end

count <= count + 1;
end
endcase
end

always @(posedge clk) // Structure of the two FIR
begin : Structure // filters in transposed form
    x <= x_in;
    // Real part of FIR filter in transposed form
re[0] <= re[1] + x160; // W^1
re[5] <= -x57;        // W^5
    // Imaginary part of FIR filter in transposed form
im[0] <= im[1] - x200; // W^1
im[5] <= x250;        // W^5
end

always @(posedge clk) // Note that all signals
begin : Coeffs // are globally defined
    // Compute the filter coefficients and use FFs
A. Verilog Source Code 2001

```
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

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
```

---

```
module ccmul #(parameter W2 = 17, // Multiplier bit width
               W1 = 9, // Bit width c+s sum
               W = 8) // Input bit width
(input clk, // Clock for the output register
input [W-1:0] x_in, y_in, c_in, // Inputs
input [W1-1:0] cps_in, cms_in, // Inputs
output [W-1:0] r_out, i_out); // Results

wire signed [W-1:0] x, y, c;  // Inputs and outputs
wire signed [W1-1:0] r, i, cmsy, cpsx, xmyc, sum; //Prod.
wire signed [W1-1:0] xmy, cps, cms, sxtx, sxty; //x-y etc.

wire clken, cr1, ov11, cin1, aclr, ADD, SUB;
   // Auxiliary signals
assign cin1=0; assign aclr=0; assign ADD=1; assign SUB=0;
assign cr1=0; assign sum=0; assign clken=0;
       // Default for add
```
assign x = x_in;  // x
assign y = y_in;  // j * y
assign c = c_in;  // cos
assign cps = cps_in; // cos + sin
assign cms = cms_in; // cos - sin

always @(posedge clk) begin
  r_out <= r[W2-3:W-1];  // Scaling and FF for output
  i_out <= i[W2-3:W-1];
end

//********** ccmul with 3 mul. and 3 add/sub **************
assign sxtx = x;  // Possible growth for
assign sxty = y;  // sub_1 -> sign extension

lpm_add_sub sub_1 // Sub: x - y
  ( .result(xmy), .dataa(sxtx), .datab(sxty)); // Used ports
// .add_sub(SUB), .cout(cr1), .overflow(ovl1), .cin(cin1),
// .clken(clken), .clock(clk), .aclr(aclr)); // Unused
  defparam sub_1.lpm_width = W1;
  defparam sub_1.lpm_representation = "SIGNED";
  defparam sub_1.lpm_direction = "sub";

lpm_mult mul_1 // Multiply (x-y)*c = xmyc
  ( .dataa(xmy), .datab(c), .result(xmyc)); // Used ports
  // .sum(sum), .clock(clk), .clken(clken), .aclr(aclr));
  defparam mul_1.lpm_widtha = W1;
  defparam mul_1.lpm_widthb = W;
  defparam mul_1.lpm_widthp = W2;
  defparam mul_1.lpm_widths = W2;
  defparam mul_1.lpm_representation = "SIGNED";

lpm_mult mul_2 // Multiply (c-s)*y = cmsy
  ( .dataa(cms), .datab(y), .result(cmsy)); // Used ports
  // .sum(sum), .clock(clk), .clken(clken), .aclr(aclr));
  defparam mul_2.lpm_widtha = W1;
  defparam mul_2.lpm_widthb = W;
  defparam mul_2.lpm_widthp = W2;
  defparam mul_2.lpm_widths = W2;
  defparam mul_2.lpm_representation = "SIGNED";

lpm_mult mul_3 // Multiply (c+s)*x = cpsx
module bfproc #(parameter W2 = 17, // Multiplier bit width W1 = 9, // Bit width c+s sum W = 8) // Input bit width
(input clk, // Clock for the output register input signed [W-1:0] Are_in, Aim_in, // 8-bit inputs input signed [W-1:0] Bre_in, Bim_in, c_in,// 8-bit inputs input signed [W1-1:0] cps_in, cms_in, // coefficients output reg signed [W-1:0] Dre_out, Dim_out,// registered output signed [W-1:0] Ere_out, Eim_out); // results
reg signed [W-1:0] dif_re, dif_im; // Bf out
reg signed [W-1:0] Are, Aim, Bre, Bim; // Inputs integer
reg signed [W-1:0] c; // Input
reg signed [W1-1:0] cps, cms; // Coefficient in

always @(posedge clk) // Compute the additions of the
begin // butterfly using integers
Are <= Are_in; // and store inputs
Aim <= Aim_in; // in flip-flops
Bre <= Bre_in;
Bim <= Bim_in;
c <= c_in; // Load from memory cos
cps <= cps_in; // Load from memory cos+sin
cms <= cms_in; // Load from memory cos-sin
Dre_out <= (Are >>> 1) + (Bre >>> 1); // Are/2 + Bre/2
Dim_out <= (Aim >>> 1) + (Bim >>> 1); // Aim/2 + Bim/2
end

// No FF because butterfly difference "diff" is not an
always @(*) // output port
begin
dif_re = (Are >>> 1) - (Bre >>> 1);//i.e. Are/2 - Bre/2
dif_im = (Aim >>> 1) - (Bim >>> 1);//i.e. Aim/2 - Bim/2
end

/*** Instantiate the complex twiddle factor multiplier
ccmul ccmul_1 // Multiply (x+jy)(c+js)
( .clk(clk), .x_in(dif_re), .y_in(dif_im), .c_in(c),
  .cps_in(cps), .cms_in(cms), .r_out(Ere_out),
  .i_out(Eim_out));
endmodule

/**************************
// IEEE STD 1364-2001 Verilog file: lfsr.v
// Author-EMAIL: Uwe.Meyer-Baes@ieee.org
/**************************
module lfsr //----> Interface
  (input clk,
   output [6:1] y); // Result

reg [6:1] ff; // Note that reg is keyword in Verilog and
  // can not be variable name
integer i;

always @(posedge clk) begin // Length-6 LFSR with xnor
for (i=6; i>=2 ; i=i-1) // Tapped delay line: shift one
  ff[i] <= ff[i-1];
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,
   output [6:1] y); // Result

  reg [6:1] ff; // Note that reg is keyword in Verilog and
               // can not be variable name

  always @(posedge clk) begin // Implement three-step
    ff[6] <= ff[3]; // length-6 LFSR with xnor;
    ff[4] <= ff[1];
  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, //----> Interface
   input signed [W:0] r_in,
   input signed [W:0] phi_in,
   output reg signed [W:0] x_out, y_out, eps);

  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) begin //----- Infer register
    if (phi_in > 90) // Test for |phi_in| > 90
        begin // Rotate 90 degrees
            x[0] <= 0;
            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) // Rotate 45 degrees
        begin
            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) // Rotate 26 degrees
        begin
        end
    else
        begin
            x[2] <= x[1];
            y[2] <= y[1];
            z[2] <= z[1];
        end
end
begin
end

if (z[2] >= 0) // Rotate 14 degrees
begin
end
else
begin
end

  x_out <= x[3];
  eps  <= z[3];
  y_out <= y[3];
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, // 1 bit input
   input signed [W1-1:0] x_in, d_in, // Inputs
   output signed [W2-1:0] e_out, y_out, // Results
   output signed [W1-1:0] f0_out, f1_out); // Results

  // 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;
wire signed [W2-1:0] p [0:1]; // 1. Product array
wire signed [W2-1:0] xemu [0:1]; // 2. Product array
wire signed [W2-1:0] y, sxty, e, sxtd;

wire clken, aclr;
wire signed [W2-1:0] sum; // Auxilary signals

assign sum=0; assign aclr=0; // Default for mult
assign clken=0;

always @(posedge clk) // Store these data or coefficients
begin: Store
    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
end

// Instantiate L pipelined multiplier
genvar I;
generate
for (I=0; I<L; I=I+1) begin: Mul_fx
    lpm_mult mul_xf // Multiply x[I]*f[I] = p[I]
        ( .dataa(x[I]), .datab(f[I]), .result(p[I]));
    // .clock(clk), .sum(sum),
    // .clken(clken), .aclr(aclr)); // Unused ports
    defparam mul_xf.lpm_widtha = W1;
    defparam mul_xf.lpm_widthb = W1;
    defparam mul_xf.lpm_widthp = W2;
    defparam mul_xf.lpm_widths = W2;
    // defparam mul_xf.lpm_pipeline = Delay;
    defparam mul_xf.lpm_representation = "SIGNED";
end // for loop
generate
endgenerate

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 pipelined multiplier
genenerate
  for (I=0; I<L; I=I+1) begin: Mul_xemu
    lpm_mult mul_I // Multiply xemu[I] = emu * x[I];
    ( .dataa(x[I]), .datab(emu), .result(xemu[I]));
  // .clock(clk), .sum(sum),
  // .clken(clken), .aclr(aclr)); // Unused ports
defparam mul_I.lpm_widtha = W1;
defparam mul_I.lpm_widthb = W1;
defparam mul_I.lpm_widthp = W2;
defparam mul_I.lpm_widths = W2;
  // defparam mul_I.lpm_pipeline = Delay;
defparam mul_I.lpm_representation = "SIGNED";
end // for loop
gendefmodule

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

endmodule
/ 2D array types memories are supported by Quartus II
// in Verilog, use therefore single vectors
reg signed [W1-1:0] x [0:4], f0, f1;
reg signed [W1-1:0] f[0:1];
reg signed [W1-1:0] d[0:3]; // Desired signal array
wire signed [W1-1:0] emu;
wire signed [W2-1:0] xemu[0:1]; // Product array
wire signed [W2-1:0] p[0:1]; // Product array
wire signed [W2-1:0] y, sxty, e, sxtd;
wire clken, aclr;
wire signed [W2-1:0] sum; // Auxilary signals

assign sum=0; assign aclr=0; // Default for mult
assign clken=0;

always @(posedge clk) // Store these data or coefficients
begin: Store
d[0] <= d_in; // Shift register for desired data
d[1] <= d[0];
d[2] <= d[1];
d[3] <= d[2];
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
end

// Instantiate L pipelined multiplier
genvar I;
generate
for (I=0; I<L; I=I+1) begin: Mul_fx
lpm_mult mul_xf // Multiply x[I]*f[I] = p[I]
(.clock(clk), .dataa(x[I]), .datab(f[I]), .result(p[I]));
// .sum(sum), .clken(clken), .aclr(aclr)); // Unused ports
defparam mul_xf.lpm_widtha = W1;
defparam mul_xf.lpm_widthb = W1;
defparam mul_xf.lpm_widthp = W2;
defparam mul_xf.lpm_widths = W2;
generate
module reg_file #(parameter W = 7, // Bit width -1  
            N = 15) //Number of register - 1
  (input clk, reg_ena, 
   input [W-1:0] data, 
   input [3:0] rd, rs, rt ,  
endmodule

// Description: This is a W x L bit register file.  
//*********************************************************  
// IEEE STD 1364-2001 Verilog file: reg_file.v  
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org  
//*********************************************************/
output reg [W:0] s, t;

reg [W:0] r [0:N];

always @(posedge clk) // Input mux inferring registers
begin : MUX
  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 three-phase clock cycle design
// It has a stack machine/0-address-type instruction word
// The stack has only four words.
//'include "220model.v"

module trisc0 #(parameter WA = 7, // Address bit width -1
              WD = 7) // Data bit width -1
  (input reset, clk, // Clock for the output register
   output jc_OUT, me_ena,
   input [WD:0] iport,
   output reg [WD:0] oport,
   output [WD:0] s0_OUT, s1_OUT, dmd_IN, dmd_OUT,
   output [WA:0] pc_OUT, dma_OUT, dma_IN,
   output [7:0] ir_imm,
output [3:0] op_code);

//parameter ifetch=0, load=1, store=2, incpc=3;
reg [1:0] state;

wire [3:0] op;
wire [WD:0] imm, dmd;
reg [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;

// Code of FSM:
always @(op) // 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

lpm_rom prog_rom
( .outclock(clk), .address(pc), .q(pmd)); // Used ports
// .inclock(clk), .memenab(ena)); // Unused
    defparam prog_rom.lpm_width = 12;
    defparam prog_rom.lpm_widthad = 8;
    defparam prog_rom.lpm_outdata = "REGISTERED";
    defparam prog_rom.lpm_address_control = "UNREGISTERED";
    defparam prog_rom.lpm_file = "TRISCOFAC.MIF";

assign not_clk = ~clk;

lpm_ram_dq data_ram
( .inclock(not_clk), .address(dma), .q(dmd),
  .data(s0), .we(mem_ena)); // Used ports
// .outclock(clk)); // Unused
    defparam data_ram.lpm_width = 8;
    defparam data_ram.lpm_widthad = 8;
    defparam data_ram.lpm_indata = "REGISTERED";
    defparam data_ram.lpm_outdata = "UNREGISTERED";
    defparam data_ram.lpm_address_control = "REGISTERED";

always @(posedge clk)
begin : P3
    integer temp;

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

// 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 = pmd; 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
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 1076-1987 uses 77 keywords and an extra 19 keywords are used in VHDL 1076-1993 (see VHDL 1076-1993 Language Reference Manual (LRM) on p. 179). New in VHDL 1076-1993 are:

- GROUP
- IMPURE
- INERTIAL
- LITERAL
- POSTPONED
- PURE
- REJECT
- ROL
- ROR
- SHARED
- SLA
- SLL
- SRA
- SRL
- UNAFFECTED
- XNOR

which are unfortunately not highlighted in the MaxPlus II editor but with the Quartus II. The IEEE standard Verilog 1364-1995, on the other hand, has 102 keywords (see LRM, p. 604). Together, both HDL languages have 201 keywords, including 19 in common. Table B.1 shows VHDL 1076-1993 keywords in capital letters, Verilog 1364-2001 keywords in small letters, and the common keywords with a capital first letter. 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
Table B.1. VHDL 1076-1993 and Verilog 1364-2001 keywords.

<table>
<thead>
<tr>
<th>VHDL/Verilog Keyword</th>
<th>VHDL/Verilog Keyword</th>
<th>VHDL/Verilog Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>event</td>
<td>notif0</td>
</tr>
<tr>
<td>ACCESS</td>
<td>EXIT</td>
<td>notif1</td>
</tr>
<tr>
<td>AFTER</td>
<td>FILE</td>
<td>NULL</td>
</tr>
<tr>
<td>ALIAS</td>
<td>For</td>
<td>OF</td>
</tr>
<tr>
<td>ALL</td>
<td>force</td>
<td>ON</td>
</tr>
<tr>
<td>always</td>
<td>forever</td>
<td>OPEN</td>
</tr>
<tr>
<td>And</td>
<td>fork</td>
<td>Or</td>
</tr>
<tr>
<td>ARCHITECTURE</td>
<td>Function</td>
<td>OTHERS</td>
</tr>
<tr>
<td>ARRAY</td>
<td>Generate</td>
<td>OUT</td>
</tr>
<tr>
<td>ASSERT</td>
<td>GENERIC</td>
<td>output</td>
</tr>
<tr>
<td>assign</td>
<td>genvar</td>
<td>PACKAGE</td>
</tr>
<tr>
<td>ATTRIBUTE</td>
<td>GROUP</td>
<td>parameter</td>
</tr>
<tr>
<td>automatic</td>
<td>GUARED</td>
<td>pmos</td>
</tr>
<tr>
<td>Begin</td>
<td>highz0</td>
<td>PORT</td>
</tr>
<tr>
<td>BLOCK</td>
<td>highz1</td>
<td>posedge</td>
</tr>
<tr>
<td>BODY</td>
<td>If</td>
<td>POSTED</td>
</tr>
<tr>
<td>buf</td>
<td>ifnone</td>
<td>primitive</td>
</tr>
<tr>
<td>BUFFER</td>
<td>IMPURE</td>
<td>PROCEDURE</td>
</tr>
<tr>
<td>bufl0</td>
<td>IN</td>
<td>PROCESS</td>
</tr>
<tr>
<td>bufl1</td>
<td>incdir</td>
<td>pull0</td>
</tr>
<tr>
<td>BUS</td>
<td>include</td>
<td>pull1</td>
</tr>
<tr>
<td>Case</td>
<td>INERTIAL</td>
<td>pullup</td>
</tr>
<tr>
<td>casex</td>
<td>initial</td>
<td>tran</td>
</tr>
<tr>
<td>casez</td>
<td>inout</td>
<td>pulsestyle_onevent</td>
</tr>
<tr>
<td>cell</td>
<td>input</td>
<td>pulsestyle_ondetect</td>
</tr>
<tr>
<td>cmos</td>
<td>instance</td>
<td>PURE</td>
</tr>
<tr>
<td>config</td>
<td>integer</td>
<td>tri</td>
</tr>
<tr>
<td>COMPONENT</td>
<td>IS</td>
<td>rcmos</td>
</tr>
<tr>
<td>CONFIGURATION</td>
<td>join</td>
<td>real</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>LABEL</td>
<td>realtime</td>
</tr>
<tr>
<td>deassign</td>
<td>large</td>
<td>RECORD</td>
</tr>
<tr>
<td>default</td>
<td>liblist</td>
<td>reg</td>
</tr>
<tr>
<td>defparam</td>
<td>Library</td>
<td>REGISTER</td>
</tr>
<tr>
<td>design</td>
<td>LINKAGE</td>
<td>REJECT</td>
</tr>
<tr>
<td>disable</td>
<td>LITERAL</td>
<td>release</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>LOOP</td>
<td>UNITS</td>
</tr>
<tr>
<td>DOWNTO</td>
<td>localparam</td>
<td>repeat</td>
</tr>
<tr>
<td>edge</td>
<td>macromodule</td>
<td>REPORT</td>
</tr>
<tr>
<td>Else</td>
<td>MAP</td>
<td>RETURN</td>
</tr>
<tr>
<td>ELSIF</td>
<td>medium</td>
<td>rmos</td>
</tr>
<tr>
<td>End</td>
<td>MOD</td>
<td>rol</td>
</tr>
<tr>
<td>endcase</td>
<td>module</td>
<td>ROR</td>
</tr>
<tr>
<td>endconfig</td>
<td>Nand</td>
<td>rpms</td>
</tr>
<tr>
<td>endfunction</td>
<td>nedge</td>
<td>weak0</td>
</tr>
<tr>
<td>endgenerate</td>
<td>NEW</td>
<td>rtran0</td>
</tr>
<tr>
<td>endmodule</td>
<td>NEXT</td>
<td>rtran1</td>
</tr>
<tr>
<td>endprimitive</td>
<td>nmos</td>
<td>scalared</td>
</tr>
<tr>
<td>endspecify</td>
<td>Nor</td>
<td>SELECT</td>
</tr>
<tr>
<td>endtable</td>
<td>noshowcancelled</td>
<td>SEVERITY</td>
</tr>
<tr>
<td>endtask</td>
<td>Not</td>
<td>SHARED</td>
</tr>
<tr>
<td>ENTITY</td>
<td>showcancelled</td>
<td></td>
</tr>
</tbody>
</table>
B.1 List of Examples

These synthesis results for all examples can be easily reproduced by using the scripts qvhdl.tcl in the VHDL or Verilog directories of the CD-ROM. Run the TCL script with

```
quartus_sh -t qvhdl.tcl > qvhdl.txt
```

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

```
....
---------------------------------------------------------------------
trisc0 fmax: 115.65 MHz ( period = 8.647 ns )
trisc0 LEs: 198 / 33,216 (< 1 %)
trisc0 M4K bits: 5,120 / 483,840 ( 1 %)
trisc0 DSP blocks: 1 / 70 ( 1 %)
---------------------------------------------------------------------
....
```

then `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 6.0 were used:

- **Device** set Family to Cyclone II and then under Available devices select EP2C35F672C6.
- **For Timing Analysis Settings** set Default required `fmax:` to 3 ns.
- **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 $9 \times 9$-bit multipliers; the number of M4K memory blocks; the Registered Performance; 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 number of $9 \times 9$-bit multiplier and number of M4K memory blocks, but the LEs and Registered Performance do not always match.
<table>
<thead>
<tr>
<th>Design</th>
<th>LEs</th>
<th>9 × 9 Mult.</th>
<th>VHDL Page</th>
<th>M4Ks</th>
<th>$f_{\text{MAX}}$ MHz</th>
<th>Page</th>
<th>LEs</th>
<th>Verilog Page</th>
<th>$f_{\text{MAX}}$ MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_1p</td>
<td>125</td>
<td>no 0</td>
<td>316.46</td>
<td>78</td>
<td>77</td>
<td>666</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add_2p</td>
<td>234</td>
<td>no 0</td>
<td>229.04</td>
<td>78</td>
<td>144</td>
<td>283.85</td>
<td>667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add_3p</td>
<td>372</td>
<td>no 0</td>
<td>215.84</td>
<td>78</td>
<td>229</td>
<td>270.42</td>
<td>668</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ammod</td>
<td>316</td>
<td>no 0</td>
<td>215.98</td>
<td>455</td>
<td>277</td>
<td>288.85</td>
<td>717</td>
<td></td>
<td></td>
</tr>
<tr>
<td>arctan</td>
<td>100</td>
<td>4 0</td>
<td>32.09</td>
<td>134</td>
<td>99</td>
<td>32.45</td>
<td>676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfproc</td>
<td>131</td>
<td>3 0</td>
<td>95.73</td>
<td>370</td>
<td>83</td>
<td>116.09</td>
<td>715</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ccmul</td>
<td>39</td>
<td>3 0</td>
<td>-</td>
<td>368</td>
<td>39</td>
<td>-</td>
<td>713</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cic3r32</td>
<td>337</td>
<td>no 0</td>
<td>282.17</td>
<td>262</td>
<td>337</td>
<td>269.69</td>
<td>694</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cic3s32</td>
<td>205</td>
<td>no 0</td>
<td>284.58</td>
<td>269</td>
<td>205</td>
<td>284.50</td>
<td>696</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmoms</td>
<td>372</td>
<td>10 0</td>
<td>85.94</td>
<td>303</td>
<td>239</td>
<td>107.48</td>
<td>705</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmul7p8</td>
<td>48</td>
<td>no 0</td>
<td>-</td>
<td>59</td>
<td>48</td>
<td>-</td>
<td>665</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cordic</td>
<td>235</td>
<td>no 0</td>
<td>222.67</td>
<td>126</td>
<td>197</td>
<td>317.16</td>
<td>674</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dafsm</td>
<td>32</td>
<td>no 0</td>
<td>420.17</td>
<td>189</td>
<td>30</td>
<td>420.17</td>
<td>683</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dapara</td>
<td>33</td>
<td>no 0</td>
<td>214.96</td>
<td>202</td>
<td>45</td>
<td>420.17</td>
<td>691</td>
<td></td>
<td></td>
</tr>
<tr>
<td>darom</td>
<td>27</td>
<td>no 1</td>
<td>218.29</td>
<td>196</td>
<td>27</td>
<td>218.96</td>
<td>687</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dasign</td>
<td>56</td>
<td>no 0</td>
<td>236.91</td>
<td>199</td>
<td>47</td>
<td>328.19</td>
<td>688</td>
<td></td>
<td></td>
</tr>
<tr>
<td>db4latti</td>
<td>418</td>
<td>no 0</td>
<td>58.81</td>
<td>324</td>
<td>248</td>
<td>74.69</td>
<td>709</td>
<td></td>
<td></td>
</tr>
<tr>
<td>db4poly</td>
<td>173</td>
<td>no 0</td>
<td>136.65</td>
<td>250</td>
<td>158</td>
<td>136.31</td>
<td>697</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_aegp</td>
<td>64</td>
<td>4 0</td>
<td>134.63</td>
<td>94</td>
<td>64</td>
<td>134.63</td>
<td>671</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div_res</td>
<td>127</td>
<td>no 0</td>
<td>265.32</td>
<td>100</td>
<td>115</td>
<td>257.86</td>
<td>673</td>
<td></td>
<td></td>
</tr>
<tr>
<td>example</td>
<td>24</td>
<td>no 0</td>
<td>420.17</td>
<td>15</td>
<td>24</td>
<td>420.17</td>
<td>663</td>
<td></td>
<td></td>
</tr>
<tr>
<td>farrow</td>
<td>279</td>
<td>6 0</td>
<td>43.91</td>
<td>292</td>
<td>175</td>
<td>65.77</td>
<td>703</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fir6dms</td>
<td>138</td>
<td>4 0</td>
<td>176.15</td>
<td>511</td>
<td>138</td>
<td>174.52</td>
<td>721</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fir_gen</td>
<td>184</td>
<td>4 0</td>
<td>329.06</td>
<td>167</td>
<td>184</td>
<td>329.06</td>
<td>680</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fir_lms</td>
<td>50</td>
<td>4 0</td>
<td>74.59</td>
<td>504</td>
<td>50</td>
<td>74.03</td>
<td>719</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fir_srg</td>
<td>114</td>
<td>no 0</td>
<td>97.21</td>
<td>179</td>
<td>70</td>
<td>106.15</td>
<td>682</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun_text</td>
<td>32</td>
<td>no 1</td>
<td>264.20</td>
<td>30</td>
<td>32</td>
<td>264.20</td>
<td>664</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iir</td>
<td>62</td>
<td>no 0</td>
<td>160.69</td>
<td>217</td>
<td>30</td>
<td>234.85</td>
<td>692</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iir_par</td>
<td>268</td>
<td>no 0</td>
<td>168.12</td>
<td>237</td>
<td>199</td>
<td>136.87</td>
<td>693</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iir_pipe</td>
<td>124</td>
<td>no 0</td>
<td>207.08</td>
<td>231</td>
<td>75</td>
<td>354.48</td>
<td>692</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lfsr</td>
<td>6</td>
<td>no 0</td>
<td>420.17</td>
<td>437</td>
<td>6</td>
<td>420.17</td>
<td>716</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lfsr6s3</td>
<td>6</td>
<td>no 0</td>
<td>420.17</td>
<td>440</td>
<td>6</td>
<td>420.17</td>
<td>717</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ln</td>
<td>88</td>
<td>10 0</td>
<td>32.76</td>
<td>145</td>
<td>88</td>
<td>32.76</td>
<td>677</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul_ser</td>
<td>121</td>
<td>no 0</td>
<td>256.15</td>
<td>82</td>
<td>140</td>
<td>245.34</td>
<td>670</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rader7</td>
<td>443</td>
<td>no 0</td>
<td>137.06</td>
<td>355</td>
<td>404</td>
<td>159.41</td>
<td>710</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rc_sinc</td>
<td>448</td>
<td>19 0</td>
<td>61.93</td>
<td>285</td>
<td>416</td>
<td>81.47</td>
<td>699</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg_file</td>
<td>211</td>
<td>no 0</td>
<td>-</td>
<td>559</td>
<td>211</td>
<td>-</td>
<td>723</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>336</td>
<td>2 0</td>
<td>82.16</td>
<td>150</td>
<td>317</td>
<td>82.73</td>
<td>678</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tris0</td>
<td>198</td>
<td>1 2</td>
<td>115.65</td>
<td>606</td>
<td>166</td>
<td>71.94</td>
<td>724</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B.2 Library of Parameterized Modules (LPM)

Throughout the book we use six different LPM megafunctions (see Fig. B.1), namely:

- `lpm_ff`, the flip-flop megafunction
- `lpm_add_sub`, the adder/subtractor megafunction
- `lpm_ram_dq`, the RAM megafunction
- `lpm_rom`, the ROM megafunction
- `lpm_divide`, the divider megafunction, and
- `lpm_mult`, the multiplier megafunction

These megafunctions are explained in the following, along with their port definitions, parameters, and resource usage. This information is also available using the Quartus II help under `megafunctions/LPM`.

![Fig. B.1. Six LPM megafunction used.](image)

B.2.1 The Parameterized Flip-Flop Megafunction (`lpm_ff`)

The `lpm_ff` function is useful if features are needed that are not available in the DFF, DFFE, TFF, and TFFE primitives, such as synchronous or asynchronous set, clear, and load inputs. We have used this megafunction for the following designs: `example`, p. 15 and `fun_text`, p. 30.

Altera recommends instantiating this function as described in “Using the MegaWizard Plug-In Manager” in the Quartus II help.

The port names and order for Verilog HDL prototypes are:
module lpm_ff (q, data, clock, enable, aclr, aset, sclr, sset, aload, sload);

The VHDL component declaration is shown below:

COMPONENT lpm_ff
    GENERIC (LPM_WIDTH: POSITIVE;
        LPM_AVALUE: STRING := "UNUSED";
        LPM_FFTYPE: STRING := "FFTYPE_DFF";
        LPM_TYPE: STRING := "L_FF";
        LPM_SVALUE: STRING := "UNUSED";
        LPM_HINT: STRING := "UNUSED";
    )
    PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
          clock: IN STD_LOGIC;
          enable: IN STD_LOGIC := '1';
          sload: IN STD_LOGIC := '0';
          sclr: IN STD_LOGIC := '0';
          sset: IN STD_LOGIC := '0';
          aload: IN STD_LOGIC := '0';
          aclr: IN STD_LOGIC := '0';
          aset: IN STD_LOGIC := '0';
          q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

Ports

The following table displays all input ports of lpm_ff:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>No</td>
<td>T-type flip-flop: Toggle enable D-type flip-flop: Data input</td>
<td>Input port LPM_WIDTH wide. If the data input is not used, at least one of the aset, aclr, sset, or sclr ports must be used. Unused data inputs default to GND.</td>
</tr>
<tr>
<td>clock</td>
<td>Yes</td>
<td>Positive-edge triggered clock</td>
<td>Default = 1</td>
</tr>
<tr>
<td>enable</td>
<td>No</td>
<td>Clock Enable input</td>
<td>If both sset and sclr are used and both are asserted, sclr is dominant. The sclr signal affects the output q values before polarity is applied to the ports.</td>
</tr>
<tr>
<td>sclr</td>
<td>No</td>
<td>Synchronous clear input</td>
<td>Sets q outputs to the value specified by LPM_SVALUE, if that value is present, or sets the q outputs to all 1s. If both sset and sclr are used and both are asserted, sclr is dominant. The sset signal affects the output q values before polarity is applied to the ports.</td>
</tr>
<tr>
<td>sset</td>
<td>No</td>
<td>Synchronous set input</td>
<td>Sets q outputs to the value specified by LPM_AVALUE, if that value is present, or sets the q outputs to all 1s.</td>
</tr>
<tr>
<td>sload</td>
<td>No</td>
<td>Synchronous load input</td>
<td>Default = 0. If sload is used, data must be used. For load operation, sload must flop with the value be high (1) and enable must be high (1) on the data input on or unconnected. The sload port is ignored when the LPM_FFTYPE parameter is set to DFF.</td>
</tr>
<tr>
<td>aclr</td>
<td>No</td>
<td>Asynchronous clear input</td>
<td>If both aset and aclr are used and both are asserted, aclr is dominant. The aclr signal affects the output q values before polarity is applied to the ports.</td>
</tr>
<tr>
<td>aset</td>
<td>No</td>
<td>Asynchronous set input</td>
<td>Sets q outputs to the value specified by LPM_AVALUE, if that value is present, or sets the q outputs to all 1s.</td>
</tr>
<tr>
<td>aload</td>
<td>No</td>
<td>Asynchronous load input. Asynchronously loads the</td>
<td>Default = 0. If aload is used, data must flop with the be used. value on the data input.</td>
</tr>
</tbody>
</table>

The following table displays all OUTPUT ports of lpm_ff:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>Yes</td>
<td>Data output from D or T flip-flops</td>
<td>Output port LPM_WIDTH wide</td>
</tr>
</tbody>
</table>
Parameters

The following table shows the parameters of the `lpm_ff` component:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPM_WIDTH</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the data and q ports</td>
</tr>
<tr>
<td>LPM_AVALUE</td>
<td>Integer</td>
<td>No</td>
<td>Constant value that is loaded when <code>aset</code> is high. If omitted, defaults to all 1s. The LPM_AVALUE parameter is limited to a maximum of 32 bits.</td>
</tr>
<tr>
<td>LPM_SVALUE</td>
<td>Integer</td>
<td>No</td>
<td>Constant value that is loaded on the rising edge of <code>clock</code> when <code>sset</code> is high. If omitted, defaults to all 1s.</td>
</tr>
<tr>
<td>LPM_FFTYPE</td>
<td>String</td>
<td>No</td>
<td>Values are DFF, TFF, and UNUSED. Type of flip-flop. If omitted, the default is DFF. When the LPM_FFTYPE parameter is set to DFF, the sload port is ignored.</td>
</tr>
<tr>
<td>LPM_HINT</td>
<td>String</td>
<td>No</td>
<td>Allows you to specify Altera-specific parameters in VHDL design files. The default is UNUSED.</td>
</tr>
<tr>
<td>LPM_TYPE</td>
<td>String</td>
<td>No</td>
<td>Identifies the LPM entity name in the VHDL design files.</td>
</tr>
</tbody>
</table>

Note that for Verilog LPM 220 synthesizable code (i.e., `220model.v`) the following parameter ordering applies: `lpm_type, lpm_width, lpm_avalue, lpm_svalue, lpm_pvalue, lpm_fftype, lpm_hint`.

Function

The following table is an example of the T-type flip-flop behavior in `lpm_ff`:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>aclr aset enable clock sclr sset sload</td>
<td>Q[LPM_WIDTH-1..0]</td>
</tr>
<tr>
<td>1 X X X X X X</td>
<td>000...</td>
</tr>
<tr>
<td>0 1 X X X X X X</td>
<td>111... or LPM_AVALUE</td>
</tr>
<tr>
<td>0 0 0 X X X X X</td>
<td>q[LPM_WIDTH-1..0] 000...</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 X X</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 X</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>
Resource Usage

The megafuction `lpm_ff` uses one logic cell per bit.

B.2.2 The Parameterized Adder/Subtractor Megafuction (`lpm_add_sub`)

Altera recommends using the `lpm_add_sub` function to replace all other types of adder/subtractor functions, including old-style adder/subtractor macrofunctions. We have used this megafunction for the following designs: example, p. 15, `fun_text`, p. 30, `ccmul`, p. 368.

Altera recommends instantiating this function as described in “Using the MegaWizard Plug-In Manager” in the Quartus II help.

The port names and order for Verilog HDL prototypes are:

```verilog
defmodule lpm_add_sub (cin,
                      dataa, datab,
                      add_sub, clock, aclr,
                      result, cout, overflow);
```

The VHDL component declaration is shown below:

```vhdl
COMPONENT lpm_add_sub
   GENERIC (LPM_WIDTH: POSITIVE;
                LPM_REPRESENTATION: STRING := "SIGNED";
                LPM_DIRECTION: STRING := "UNUSED";
                LPM_HINT: STRING := "UNUSED";
                LPM_PIPELINE: INTEGER := 0;
                LPM_TYPE: STRING := "L_ADD_SUB");
   PORT (dataa, datab : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
         aclr, clken, clock, cin : IN STD_LOGIC := '0';
         add_sub : IN STD_LOGIC := '1';
         result : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
         cout, overflow : OUT STD_LOGIC);
END COMPONENT;
```
### Ports

The following table displays all input ports of `lpm_add_sub`:

<table>
<thead>
<tr>
<th>Port name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cin</td>
<td>No</td>
<td>Carry-in to the low-order bit. If the operation is ADD, low = 0 and high = +1. If the operation is SUB, low = −1 and high = 0.</td>
<td>If omitted, the default is 0 (i.e., low if the operation is ADD and high if the operation is SUB).</td>
</tr>
<tr>
<td>dataa</td>
<td>Yes</td>
<td>Augend/Minuend</td>
<td>Input port LPM_WIDTH wide</td>
</tr>
<tr>
<td>datab</td>
<td>Yes</td>
<td>Addend/Subtrahend</td>
<td>Input port LPM_WIDTH wide</td>
</tr>
<tr>
<td>add_sub</td>
<td>No</td>
<td>If the signal is high, the operation = dataa + datab. If the signal is low, the operation = dataa − datab.</td>
<td>If the LPM_DIRECTION parameter is used, add_sub cannot be used. If omitted, the default is ADD. Altera recommends that you use the LPM_DIRECTION parameter to specify the operation of the <code>lpm_add_sub</code> function, rather than assigning a constant to the add_sub port.</td>
</tr>
<tr>
<td>clock</td>
<td>No</td>
<td>Clock for pipelined usage</td>
<td>The clock port provides pipelined operation for the <code>lpm_add_sub</code> function. For LPM_PIPELINE values other than 0 (default value), the clock port must be connected.</td>
</tr>
<tr>
<td>clken</td>
<td>No</td>
<td>Clock enable for pipelined usage</td>
<td>Available for VHDL only</td>
</tr>
<tr>
<td>aclr</td>
<td>No</td>
<td>Asynchronous clear for pipelined usage</td>
<td>The pipeline initializes to an undefined (X) logic level. The aclr port can be used at any time to reset the pipeline to all 0s, asynchronously to the clock signal.</td>
</tr>
</tbody>
</table>
The following table displays all output ports of `lpm_add_sub`:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Yes</td>
<td><code>dataa + or - datab + or - cin</code></td>
<td>Output port LPM_WIDTH wide</td>
</tr>
<tr>
<td>cout</td>
<td>No</td>
<td>Carry-out (borrow-in) of the MSB</td>
<td>If overflow is used, cout cannot be used. The cout port has a physical interpretation as the carry-out (borrow-in) of the MSB. cout is most meaningful for detecting overflow in UNSIGNED operations.</td>
</tr>
<tr>
<td>overflow</td>
<td>No</td>
<td>Result exceeds available precision.</td>
<td>If overflow is used, cout cannot be used. The overflow port has a physical interpretation as the XOR of the carry-in to the MSB with the carry-out of the MSB. overflow is meaningful only when the LPM_REPRESENTATION parameter value is SIGNED.</td>
</tr>
</tbody>
</table>

**Parameters**

The following table shows the parameters of the `lpm_add_sub` component:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPM_WIDTH</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the dataa, datab, and result ports.</td>
</tr>
<tr>
<td>LPM_DIRECTION</td>
<td>String</td>
<td>No</td>
<td>Values are ADD, SUB, and UNUSED. If omitted, the default is DEFAULT, which directs the parameter to take its value from the add_sub port. The add_sub port cannot be used if LPM_DIRECTION is used. Altera recommends that you use the LPM_DIRECTION parameter to specify the operation of the lpm_add_sub function, rather than assigning a constant to the add_sub port.</td>
</tr>
<tr>
<td>LPM_REPRESENTATION</td>
<td>String</td>
<td>No</td>
<td>Type of addition performed: SIGNED, UNSIGNED, or UNUSED. If omitted, the default is SIGNED.</td>
</tr>
<tr>
<td>LPM_PIPELINE</td>
<td>Integer</td>
<td>No</td>
<td>Specifies the number of clock cycles of latency associated with the result output. A value of zero (0) indicates that no latency exists, and that a purely combinatorial function will be instantiated. If omitted, the default is 0 (nonpipelined).</td>
</tr>
<tr>
<td>LPM_HINT</td>
<td>String</td>
<td>No</td>
<td>Allows you to specify Altera-specific parameters in VHDL design files. The default is UNUSED.</td>
</tr>
<tr>
<td>LPM_TYPE</td>
<td>String</td>
<td>No</td>
<td>Identifies the LPM entity name in VHDL design files.</td>
</tr>
<tr>
<td>ONE_INPUT_IS_CONSTANT</td>
<td>String</td>
<td>No</td>
<td>Altera-specific parameter. Values are YES, NO, and UNUSED. Provides greater optimization, if one input is constant. If omitted, the default is NO.</td>
</tr>
<tr>
<td>MAXIMIZE_SPEED</td>
<td>Integer</td>
<td>No</td>
<td>Altera-specific parameter. You can specify a value between 0 and 10. If used, MaxPlus II attempts to optimize a specific instance of the lpm_add_sub function for speed rather than area, and overrides the setting of the Optimize option in the Global Project Logic Synthesis dialog box (Assign menu). If MAXIMIZE_SPEED is unused, the value of the Optimize option is used instead. If the setting for MAXIMIZE_SPEED is 6 or higher, the compiler will optimize lpm_add_sub megafunctations for higher speed; if the setting is 5 or less, the compiler will optimize for smaller area.</td>
</tr>
</tbody>
</table>

Note that for Verilog LPM 220 synthesizable code (i.e., 220model.v) the following parameter ordering applies: lpm_type, lpm_width, lpm_direction, lpm_representation, lpm_pipeline, lpm_hint.
Function

The following table is an example of the UNSIGNED behavior in \texttt{lpm_add_sub}:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_sub dataa datab</td>
<td>cout,result,overflow</td>
</tr>
<tr>
<td>1 a b</td>
<td>a + b + cin cout</td>
</tr>
<tr>
<td>0 a b</td>
<td>a - b - cin !cout</td>
</tr>
</tbody>
</table>

The following table is an example of the SIGNED behavior in \texttt{lpm_add_sub}:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_sub dataa datab</td>
<td>cout,sum,overflow</td>
</tr>
<tr>
<td>1 a b</td>
<td>a + b + cin a \geq 0 and b \geq 0 and sum &lt; 0 or a &lt; 0 and b &lt; 0 and sum \geq 0</td>
</tr>
<tr>
<td>0 a b</td>
<td>a - b - cin a \geq 0 and b &lt; 0 and sum &lt; 0 or a &lt; 0 and b \geq 0 and sum \geq 0</td>
</tr>
</tbody>
</table>

Resource Usage

The following table summarizes the resource usage for an \texttt{lpm_add_sub} mega-function used to implement a 16-bit unsigned adder with a carry-in input and a carry-out output. Logic cell usage scales linearly in proportion to adder width.

<table>
<thead>
<tr>
<th>Design goals</th>
<th>Design results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device family</td>
<td>Optimization</td>
</tr>
<tr>
<td>FLEX 6K, 8K, and 10K</td>
<td>Routability</td>
</tr>
<tr>
<td>FLEX 6K, 8K, and 10K</td>
<td>Speed</td>
</tr>
<tr>
<td>MAX 5K, 7K, and 9K</td>
<td>Routability</td>
</tr>
</tbody>
</table>

Numbers of shared expanders used are shown in parentheses.

B.2.3 The Parameterized Multiplier Megafuction \texttt{(lpm_mult)}

Altera recommends that you use \texttt{lpm_mult} to replace all other types of multiplier functions, including old-style multiplier macrofunctions. We have used
this megafunction for the designs \texttt{fir\_gen}, p. 167, \texttt{ccmul}, p. 368, \texttt{fir\_lms}, p. 504, and \texttt{fir6dlms}, p. 511.

Altera recommends instantiating this function as described in “Using the MegaWizard Plug-In Manager” in the Quartus II help.

The port names and order for Verilog HDL prototype are:

```verilog
module lpm_mult ( dataa, datab, sum, aclr, clock, result);
```

The VHDL component declaration is shown below:

```vhdl
COMPONENT lpm_mult
    GENERIC (LPM_WIDTHA: POSITIVE;
              LPM_WIDTHB: POSITIVE;
              LPM_WIDTHS: POSITIVE;
              LPM_WIDTHP: POSITIVE;
              LPM_REPRESENTATION: STRING := "UNSIGNED";
              LPM_PIPELINE: INTEGER := 0;
              LPM_TYPE: STRING := "L\_MULT";
              LPM_HINT: STRING := "UNUSED");
    PORT (dataa : IN STD\_LOGIC\_VECTOR(LPM\_WIDTHA\_1 DOWNTO 0);
              datab : IN STD\_LOGIC\_VECTOR(LPM\_WIDTHB\_1 DOWNTO 0);
              aclr, clken, clock : IN STD\_LOGIC := '0';
              sum : IN STD\_LOGIC\_VECTOR(LPM\_WIDTHS\_1 DOWNTO 0)
              := (OTHERS => '0');
              result: OUT STD\_LOGIC\_VECTOR(LPM\_WIDTHP\_1 DOWNTO 0)
              );
END COMPONENT;
```

**Ports**

The following table displays all input ports of \texttt{lpm\_mult}:
<table>
<thead>
<tr>
<th>Port name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataa</td>
<td>Yes</td>
<td>Multiplicand</td>
<td>Input port LPM_WIDTHA wide</td>
</tr>
<tr>
<td>datab</td>
<td>Yes</td>
<td>Multiplier</td>
<td>Input port LPM_WIDTHB wide</td>
</tr>
<tr>
<td>sum</td>
<td>No</td>
<td>Partial sum</td>
<td>The clock port provides pipelined operation for the lpm_mult function. For LPM_PIPELINE values other than 0 (default value), the clock port must be connected.</td>
</tr>
<tr>
<td>clock</td>
<td>No</td>
<td>Clock for pipelined usage</td>
<td></td>
</tr>
<tr>
<td>clken</td>
<td>No</td>
<td>Clock enable for pipelined usage</td>
<td>Available for VHDL only.</td>
</tr>
<tr>
<td>aclr</td>
<td>No</td>
<td>Asynchronous clear for pipelined usage</td>
<td>The pipeline initializes to an undefined (X) logic level. The aclr port can be used at any time to reset the pipeline to all 0s, asynchronously to the clock signal.</td>
</tr>
</tbody>
</table>

The following table displays all output ports of lpm_mult:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Yes</td>
<td>result = dataa * datab + sum. The product LSB is aligned with the sum LSB.</td>
<td>Output port LPM_WIDTHP wide. If LPM_WIDTHP &lt; max (LPM_WIDTHA + LPM_WIDTHB, LPM_WIDTHS) or (LPM_WIDTHA + LPM_WIDTHS), only the LPM_WIDTHP MSBs are present.</td>
</tr>
</tbody>
</table>

Parameters

The following table shows the parameters of the lpm_mult component:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPM_WIDTHA</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the dataa port</td>
</tr>
<tr>
<td>LPM_WIDTHB</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the datab port</td>
</tr>
<tr>
<td>LPM_WIDTHP</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the result port</td>
</tr>
<tr>
<td>LPM_WIDTHS</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the sum port. Required even if the sum port is not used.</td>
</tr>
<tr>
<td>LPM_REPRESENTATION</td>
<td>String</td>
<td>No</td>
<td>Type of multiplication performed: SIGNED, UNSIGNED, or UNUSED. If omitted, the default is UNSIGNED.</td>
</tr>
<tr>
<td>LPM_PIPELINE</td>
<td>Integer</td>
<td>No</td>
<td>Specifies the number of clock cycles of latency associated with the result output. A value of zero (0) indicates that no latency exists, and that a purely combinatorial function will be instantiated. If omitted, the default is 0 (non-pipelined).</td>
</tr>
<tr>
<td>LPM_HINT</td>
<td>String</td>
<td>No</td>
<td>Allows you to assign Altera-specific parameters in VHDL design files. The default is UNUSED.</td>
</tr>
<tr>
<td>LPM_TYPE</td>
<td>String</td>
<td>No</td>
<td>Identifies the LPM entity name in VHDL design files.</td>
</tr>
<tr>
<td>INPUT_A_IS_CONSTANT</td>
<td>String</td>
<td>No</td>
<td>Altera-specific parameter. Values are YES, NO, and UNUSED. If dataa is connected to a constant value, setting INPUT_A_IS_CONSTANT to YES optimizes the multiplier for resource usage and speed. If omitted, the default is NO.</td>
</tr>
<tr>
<td>INPUT_B_IS_CONSTANT</td>
<td>String</td>
<td>No</td>
<td>Altera-specific parameter. Values are YES, NO, and UNUSED. If datab is connected to a constant value, setting INPUT_B_IS_CONSTANT to YES optimizes the multiplier for resource usage and speed. The default is NO.</td>
</tr>
</tbody>
</table>
### B.2 Library of Parameterized Modules (LPM)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE_EAB</td>
<td>String</td>
<td>No</td>
<td>Altera-specific parameter. Values are ON, OFF, and UNUSED. Setting the USE_EAB parameter to ON allows Quartus II to use EABs to implement 4 × 4 or (8 × constant value) building blocks in FLEX 10K devices. Altera recommends that you set USE_EAB to ON only when LCELLS are in short supply. If you wish to use this parameter, when you instantiate the function in a GDF, you must specify it by entering the parameter name and value manually with the Edit Ports/Parameters dialog box (Symbol menu). You can also use this parameter name in a TDF or a Verilog design file. You must use the LPM_HINT parameter to specify the USE_EAB parameter in VHDL design files.</td>
</tr>
<tr>
<td>DEDICATED_MULTIPLIER_CIRCUITRY</td>
<td>String</td>
<td>No</td>
<td>Altera-specific parameter. You must use the LPM_HINT parameter to specify the DEDICATED_MULTIPLIER_CIRCUITRY parameter in VHDL design files. Specifies whether to use dedicated multiplier circuitry. Values are ‘AUTO’, YES, and ‘NO’. If omitted, the default is AUTO.</td>
</tr>
<tr>
<td>LATENCY</td>
<td>Integer</td>
<td>No</td>
<td>Altera-specific parameter. Same as LPM_PIPELINE. (This parameter is provided only for backward compatibility with MaxPlus II pre-version 7.0 designs. For all new designs, you should use the LPM_PIPELINE parameter instead.)</td>
</tr>
<tr>
<td>MAXIMIZE_SPEED</td>
<td>Integer</td>
<td>No</td>
<td>Altera-specific parameter. You can specify a value between 0 and 10. If used, MaxPlus II attempts to optimize a specific instance of the lpm_mult function for speed rather than area, and overrides the setting of the Optimize option in the Global Project Logic Synthesis dialog box (Assign menu). If MAXIMIZE_SPEED is unused, the value of the Optimize option is used instead. If the setting for MAXIMIZE_SPEED is 6 or higher, the compiler will optimize lpm_mult megafunctions for higher speed; if the setting is 5 or less, the compiler will optimize for smaller area.</td>
</tr>
<tr>
<td>LPM_HINT</td>
<td>String</td>
<td>No</td>
<td>Allows you to specify Altera-specific parameters in VHDL design files. The default is UNUSED.</td>
</tr>
</tbody>
</table>

Note that specifying a value for MAXIMIZE_SPEED has an effect only if LPM_REPRESENTATION is set to SIGNED.

Note that for Verilog LPM 220 synthesizable code (i.e., 220model.v) the following parameter ordering applies: lpm_type, lpm_width, lpm_widthb, lpm_widths, lpm_widthp, lpm_representation, lpm_pipeline, lpm_hint.
Function

The following table is an example of the UNSIGNED behavior in lpm_mult:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataa</td>
<td>datab</td>
</tr>
<tr>
<td>datab</td>
<td>sum</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>s</td>
<td>LPM_WIDTHP most significant bits of (a \cdot b + s)</td>
</tr>
</tbody>
</table>

Resource Usage

The following table summarizes the resource usage for an lpm_mult function used to implement 4-bit and 8-bit multipliers with \(\text{LPM\_PIPELINE} = 0\) and without the optional \(\text{sum}\) input. Logic cell usage scales linearly in proportion to the square of the input width.

<table>
<thead>
<tr>
<th>Design goals</th>
<th>Design results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device family</td>
<td>Optimization</td>
</tr>
<tr>
<td>FLEX 6K, 8K, and 10K</td>
<td>Routability</td>
</tr>
<tr>
<td>FLEX 6K, 8K, and 10K</td>
<td>Speed</td>
</tr>
<tr>
<td>MAX 5K, 7K, and 9K</td>
<td>Routability</td>
</tr>
<tr>
<td>MAX 5K, 7K, and 9K</td>
<td>Speed</td>
</tr>
</tbody>
</table>

Numbers of shared expanders used are shown in parentheses. In the FLEX 10K device family, the 4-bit by 4-bit multiplier example shown above can be implemented in a single EAB.

B.2.4 The Parameterized ROM Megafunction (lpm_rom)

The lpm_rom block is parameterized ROM with separate input and output clocks. We have used this megafonction for the designs fun_text, p. 30 and darom, p. 196.

The lpm_rom block can also be used with older device families, e.g., Flex 10K. Altera translates for newer devices like Cyclone II the lpm_rom in the altsyncram megafonction block. But the altsyncram is not supported for Flex 10K that is used on the popular UP2 boards.

Altera recommends instantiating this function as described in “Using the MegaWizard Plug-In Manager” in the Quartus II help.

The port names and order for Verilog HDL prototype are:
module lpm_rom ( address, inclock, outclock, memenab, q);

The VHDL component declaration is shown below:

```
COMPONENT lpm_rom
  GENERIC (LPM_WIDTH : POSITIVE;
          LPM_TYPE : STRING := "L_ROM";
          LPM_WIDTHAD : POSITIVE;
          LPM_NUMWORDS : POSITIVE;
          LPM_FILE : STRING;
          LPM_ADDRESS_CONTROL : STRING := "REGISTERED";
          LPM_OUTDATA : STRING := "REGISTERED";
          LPM_HINT : STRING := "UNUSED";
  PORT(address : IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
        inclock : IN STD_LOGIC := '1';
        outclock : IN STD_LOGIC := '1';
        memenab : IN STD_LOGIC := '1';
        q : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0)
  );
END COMPONENT;
```

Ports

The following table displays all input ports of lpm_rom:

<table>
<thead>
<tr>
<th>Port name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Yes</td>
<td>Address input to the memory</td>
<td>Input port LPM_WIDTHAD wide</td>
</tr>
<tr>
<td>inclock</td>
<td>No</td>
<td>Clock for input registers</td>
<td>The address port is synchronous (registered) when the inclock port is connected, and is asynchronous (registered) when the inclock port is not connected.</td>
</tr>
<tr>
<td>outclock</td>
<td>No</td>
<td>Clock for output registers</td>
<td>The addressed memory content-to-q response is synchronous when the outclock port is connected, and is asynchronous when it is not connected.</td>
</tr>
<tr>
<td>memenab</td>
<td>No</td>
<td>Memory enable</td>
<td>High = data output on q. Low = high-impedance outputs</td>
</tr>
</tbody>
</table>

The following table displays all output ports of lpm_rom:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>Yes</td>
<td>Output of memory</td>
<td>Output port LPM_WIDTH wide</td>
</tr>
</tbody>
</table>
Parameters

The following table shows the parameters of the \texttt{lpm\_rom} component:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{LPM_WIDTH}</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the address port. \texttt{LPM_WIDTHAD} should be ( \log_2(\texttt{LPM_NUMWORDS}) ) if \texttt{LPM_WIDTHAD} is too small. If it is too large, addresses that are too high will return undefined logic levels.</td>
</tr>
<tr>
<td>\texttt{LPM_WIDTHAD}</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the address port. \texttt{LPM_WIDTHAD} should be ( \log_2(\texttt{LPM_NUMWORDS}) ). If \texttt{LPM_WIDTHAD} is too small, some memory locations will not be addressable. If it is too large, addresses that are too high will return undefined logic levels.</td>
</tr>
<tr>
<td>\texttt{LPM_NUMWORDS}</td>
<td>Integer</td>
<td>Yes</td>
<td>Number of words stored in memory. In general, this value should be ( 2^\texttt{LPM_WIDTHAD} - 1 \leq \texttt{LPM_NUMWORDS} ). If omitted, the default is ( 2^\texttt{LPM_WIDTHAD} ).</td>
</tr>
<tr>
<td>\texttt{LPM_FILE}</td>
<td>String</td>
<td>No</td>
<td>Name of the Memory Initialization File (<em>.mif) or Hexadecimal (Intel-format) File (</em>.hex) containing ROM initialization data (&lt;filename&gt;), or \texttt{UNUSED}.</td>
</tr>
<tr>
<td>\texttt{LPM_ADDRESS_CONTROL}</td>
<td>String</td>
<td>No</td>
<td>Values are \texttt{REGISTERED}, \texttt{UNREGISTERED}, and \texttt{UNUSED}. Indicates whether the address port is registered. If omitted, the default is \texttt{REGISTERED}.</td>
</tr>
<tr>
<td>\texttt{LPM_OUTDATA}</td>
<td>String</td>
<td>No</td>
<td>Values are \texttt{REGISTERED}, \texttt{UNREGISTERED}, and \texttt{UNUSED}. Indicates whether the q and eq ports are registered. If omitted, the default is \texttt{REGISTERED}.</td>
</tr>
<tr>
<td>\texttt{LPM_HINT}</td>
<td>String</td>
<td>No</td>
<td>Allows you to specify Altera-specific parameters in VHDL design files. The default is \texttt{UNUSED}.</td>
</tr>
<tr>
<td>\texttt{LPM_TYPE}</td>
<td>String</td>
<td>No</td>
<td>Identifies the LPM entity name in VHDL design files.</td>
</tr>
</tbody>
</table>

Note that for Verilog LPM 220 synthesizable code (i.e., \texttt{220model.v}) the following parameter ordering applies: \texttt{lpm\_type, lpm\_width, lpm\_widthad, lpm\_numwords, lpm\_address\_control, lpm\_outdata, lpm\_file, lpm\_hint}. |
Function

The following table shows the *synchronous* read from memory behavior of `lpm_rom`:

<table>
<thead>
<tr>
<th>OUTCLOCK</th>
<th>MEMENAB</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>L</td>
<td>q output is high impedance (memory not enabled)</td>
</tr>
<tr>
<td>l</td>
<td>H</td>
<td>The output register is loaded with the contents of the memory location pointed to by <code>address</code>. q outputs the contents of the output register.</td>
</tr>
</tbody>
</table>

The output q is asynchronous and reflects the data in the memory to which `address` points. The following table shows the *asynchronous* memory operations behavior of `lpm_rom`:

<table>
<thead>
<tr>
<th>MEMENAB</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>q output is high-impedance (memory not enabled)</td>
</tr>
<tr>
<td>H</td>
<td>The memory location pointed to by <code>address</code> is read</td>
</tr>
</tbody>
</table>

Totally asynchronous memory operations occur when neither `inclock` nor `outclock` is connected. The output q is asynchronous and reflects the memory location pointed to by `address`. Since this totally asynchronous memory operation is only available with Flex 10K devices, but not with Cyclone II, we do not use this mode in our designs. Either input or output is registered in all of our designs that use memory blocks.

Resource Usage

The Megafunction `lpm_rom` uses one embedded cell per memory bit.

B.2.5 The Parameterized Divider Megafunction (`lpm_divide`)

Altera recommends that you use `lpm_divide` to replace all other types of divider functions, including old-style divide macrofunction. We have used this megafunction for the array divider designs p. 103.

Altera recommends instantiating this function as described in “Using the MegaWizard Plug-In Manager” in the Quartus II help. The port names and order for Verilog HDL prototype are:

```verilog
module lpm_divide ( quotient, remain, numer, denom, clock, clken, aclr )
```

The VHDL component declaration is shown below:

```vhdl
COMPONENT lpm_divide
    GENERIC ( LPM_WIDTHN: POSITIVE;
               LPM_WIDTHD: POSITIVE;
               LPM_NREPRESENTATION: STRING: = "UNSIGNED";
               LPM_DREPRESENTATION: STRING: = "UNSIGNED";
               LPM_TYPE: STRING :="LPM_DIVIDE";
```
LPM_PIPELINE: INTEGER := 0;
LPM_HINT: STRING := "UNUSED";
);
PORT ( numer: IN STD_LOGIC_VECTOR(LPM_WIDTHN-1 DOWNTO 0);
      denom: IN STD_LOGIC_VECTOR(LPM_WIDTHD-1 DOWNTO 0);
      clock, aclr: IN STD_LOGIC := '0';
      clken: IN STD_LOGIC := '1';
      quotient: OUT STD_LOGIC_VECTOR(LPM_WIDTHN-1 DOWNTO 0);
      remain: OUT STD_LOGIC_VECTOR(LPM_WIDTHD-1 DOWNTO 0)
    );
END COMPONENT;

Ports

The following table displays all input ports of lpm_divide:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>numer</td>
<td>Yes</td>
<td>Numerator</td>
<td>Input port LPM_WIDTHN wide.</td>
</tr>
<tr>
<td>denom</td>
<td>Yes</td>
<td>Denominator</td>
<td>Input port LPM_WIDTHD wide.</td>
</tr>
<tr>
<td>clock</td>
<td>No</td>
<td>Clock</td>
<td>pipelined usage. for You must connect the clock input if you set LPM_PIPELINE to a value other than 0.</td>
</tr>
<tr>
<td>clken</td>
<td>No</td>
<td>Clock enable</td>
<td>for pipelined usage.</td>
</tr>
<tr>
<td>aclr</td>
<td>No</td>
<td>Asynchronous</td>
<td>clear signal. The aclr port may be used at any time to reset the pipeline to all 0s asynchronously to the clock input.</td>
</tr>
</tbody>
</table>

The following table displays all output ports of lpm_divide:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>quotient</td>
<td>Yes</td>
<td>Output port</td>
<td>LPM_WIDTHN wide. You must use either the quotient or the remain ports.</td>
</tr>
<tr>
<td>remain</td>
<td>Yes</td>
<td>Output port</td>
<td>LPM_WIDTHD wide. You must use either the quotient or the remain ports.</td>
</tr>
</tbody>
</table>
Parameters

The following table shows the parameters of the \texttt{lpm\_divide} component:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{LPM_WIDTHN}</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the numerator and quotient port</td>
</tr>
<tr>
<td>\texttt{LPM_WIDTHD}</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the denominator and remainder port</td>
</tr>
<tr>
<td>\texttt{LPM_NREPRESENTATION}</td>
<td>String</td>
<td>No</td>
<td>Specifies whether the numerator is \texttt{SIGNED} or \texttt{UNSIGNED}. Only \texttt{UNSIGNED} is supported for now.</td>
</tr>
<tr>
<td>\texttt{LPM_DREPRESENTATION}</td>
<td>String</td>
<td>No</td>
<td>Specifies whether the denominator is \texttt{SIGNED} or \texttt{UNSIGNED}. Only \texttt{UNSIGNED} is supported for now.</td>
</tr>
<tr>
<td>\texttt{LPM_PIPELINE}</td>
<td>Integer</td>
<td>No</td>
<td>Specifies the number of clock cycles of latency associated with the quotient and remain outputs. A value of zero (0) indicates that no latency exists, and that a purely combinatorial function will be instantiated. If omitted, the default is 0 (nonpipelined). You cannot specify a value for the \texttt{LPM_PIPELINE} parameter that is higher than \texttt{LPM_WIDTHN}.</td>
</tr>
<tr>
<td>\texttt{LPM_TYPE}</td>
<td>String</td>
<td>No</td>
<td>Identifies the LPM entity name in VHDL design files.</td>
</tr>
<tr>
<td>\texttt{LPM_HINT}</td>
<td>String</td>
<td>No</td>
<td>Allows you to assign Altera-specific parameters in VHDL design files. The default is \texttt{UNUSED}.</td>
</tr>
</tbody>
</table>

You can pipeline a design by connecting the \texttt{clock} input and specifying the number of clock cycles of latency with the \texttt{LPM\_PIPELINE} parameter.

Note that for Verilog LPM 220 synthesizable code (i.e., \texttt{220model.v}) the following parameter ordering applies: \texttt{lpm\_type, lpm\_widthn, lpm\_widthd, lpm\_nrepresentation, lpm\_drepresentation, lpm\_pipeline}.

B.2.6 The Parameterized RAM Megafuction (\texttt{lpm\_ram\_dq})

The \texttt{lpm\_ram\_dq} block is parameterized RAM with separate input and output ports. The \texttt{lpm\_ram\_dq} block can also be used with older device families, e.g., Flex10K. Altera translates for newer devices like Cyclone II the \texttt{lpm\_ram\_dq} in the \texttt{altsyncram} megafuction block. But the \texttt{altsyncram} is not supported for Flex10K that is used on the popular UP2 boards.

We have used this megafuction for the design \texttt{trisco0}, p. 606.

Altera recommends instantiating this function as described in “Using the MegaWizard Plug-In Manager” in the Quartus II help.

The port names and order for Verilog HDL prototype are:

\begin{verbatim}
module lpm\_ram\_dq ( q, data, inclock, outclock, we, address);
\end{verbatim}
The VHDL component declaration is shown below:

```vhdl
COMPONENT lpm_ram_dq
    GENERIC (LPM_WIDTH : POSITIVE;
              LPM_WIDTHAD : POSITIVE;
              LPM_NUMWORDS : NATURAL := 0;
              LPM_INDATA : STRING := "REGISTERED";
              LPM_ADDRESS_CONTROL : STRING := "REGISTERED";
              LPM_OUTDATA : STRING := "REGISTERED";
              LPM_FILE : STRING := "UNUSED";
              LPM_TYPE : STRING := "LPM_RAM_DQ";
              LPM_HINT : STRING := "UNUSED" );
    PORT (data : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
          address : IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
          inclock, outclock : IN STD_LOGIC := '0';
          we : IN STD_LOGIC;
          q : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0)
          );
END COMPONENT;
```

Ports

The following table displays all input ports of `lpm_ram_dq`:

<table>
<thead>
<tr>
<th>Port name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Yes</td>
<td>Address input to the memory</td>
<td>Input port LPM_WIDTHAD wide</td>
</tr>
<tr>
<td>data</td>
<td>Yes</td>
<td>Data input to the memory</td>
<td>Input port LPM_WIDTHAD wide</td>
</tr>
<tr>
<td>inclock</td>
<td>No</td>
<td>Clock for input registers</td>
<td>The <code>address</code> port is synchronous (registered) when the <code>inclock</code> port is connected, and is asynchronous (registered) when the <code>inclock</code> port is not connected.</td>
</tr>
<tr>
<td>outclock</td>
<td>No</td>
<td>Clock for output registers</td>
<td>The addressed memory content-to-q response is synchronous when the <code>outclock</code> port is connected, and is asynchronous when it is not connected.</td>
</tr>
<tr>
<td>we</td>
<td>Yes</td>
<td>Memory input enable</td>
<td>Write enable input. Enables write operations to the memory when high.</td>
</tr>
</tbody>
</table>

The following table displays all output ports of `lpm_ram_dq`:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>Yes</td>
<td>Output of memory</td>
<td>Output port LPM_WIDTH wide</td>
</tr>
</tbody>
</table>
### Parameters

The following table shows the parameters of the `lpm_ram_dq` component:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPM_WIDTH</td>
<td>Integer</td>
<td>Yes</td>
<td>Width of the address port. LPM_WIDTH should be (but is not required to be)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>equal to $\log_2(LPM_NUMWORDS)$. If LPM_WIDTH is too small, some memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>locations will not be addressable. If it is too large, addresses that are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>too high will return undefined logic levels.</td>
</tr>
<tr>
<td>LPM_WIDTHAD</td>
<td>Integer</td>
<td>Yes</td>
<td>Number of words stored in memory. In general, this value should be (but is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>not required to be) $2^{LPM_WIDTHAD} - 1 &lt; LPM_NUMWORDS \leq 2^{LPM_WIDTHAD}$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If omitted, the default is $2^{LPM_WIDTHAD}$.</td>
</tr>
<tr>
<td>LPM_NUMWORDS</td>
<td>Integer</td>
<td>Yes</td>
<td>Name of the Memory Initialization File (*.mif) or Hexadecimal (Intel-format)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>File (*.hex) containing ROM initialization data (&lt;filename&gt;), or UNUSED.</td>
</tr>
<tr>
<td>LPM_FILE</td>
<td>String</td>
<td>No</td>
<td>Values are REGISTERED, UNREGISTERED, and UNUSED. Indicates whether the address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>port is registered. If omitted, the default is REGISTERED.</td>
</tr>
<tr>
<td>LPM_ADDRESS_CONTROL</td>
<td>String</td>
<td>No</td>
<td>Values are REGISTERED, UNREGISTERED, and UNUSED. Indicates whether the q and eq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ports are registered. If omitted, the default is REGISTERED.</td>
</tr>
<tr>
<td>LPM_OUTDATA</td>
<td>String</td>
<td>No</td>
<td>Allows you to specify Altera-specific parameters in VHDL design files. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>default is UNUSED.</td>
</tr>
<tr>
<td>LPM_HINT</td>
<td>String</td>
<td>No</td>
<td>Identifies the LPM entity name in the VHDL design files.</td>
</tr>
<tr>
<td>LPM_TYPE</td>
<td>String</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Note that for Verilog LPM 220 synthesizable code (i.e., `220model.v`) the following parameter ordering applies: `lpm_type, lpm_width, lpm_widthad, lpm_numwords, lpm_address_control, lpm_outdata, lpm_file, lpm_hint`. 
Function

The following table shows the synchronous read and write memory behavior of `lpm_ram_dq`:

<table>
<thead>
<tr>
<th>inclock</th>
<th>we</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>-</td>
<td>No change (requires rising clock edge).</td>
</tr>
<tr>
<td>✔️</td>
<td>H</td>
<td>The memory location pointed to by <code>address</code> is loaded with <code>data</code>.</td>
</tr>
<tr>
<td>✔️</td>
<td>L</td>
<td>The memory location pointed to by <code>address</code> is read from the array. If <code>outclock</code> is not used, the read data appears at the outputs.</td>
</tr>
</tbody>
</table>

The following table shows the synchronous read from memory from memory operations behavior of `lpm_ram_dq`:

<table>
<thead>
<tr>
<th>outclock</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>No change</td>
</tr>
<tr>
<td>✔️</td>
<td>The memory location pointed to by <code>address</code> is read and written into the output register.</td>
</tr>
</tbody>
</table>

Totally asynchronous memory operations occur when neither `inclock` nor `outclock` is connected. The output `q` is asynchronous and reflects the memory location pointed to by `address`. Since this totally asynchronous memory operation is only available with Flex 10K devices, but not with Cyclone II, we do not use this mode in our designs. Either the input or output is registered in all of our designs that use memory blocks.

Resource Usage

The Megafuncton `lpm_ram_dq` uses one embedded cell per memory bit.
## C. Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Accumulator</td>
</tr>
<tr>
<td>ACT</td>
<td>Actel FPGA family</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog-to-digital converter</td>
</tr>
<tr>
<td>ADCL</td>
<td>All-digital CL</td>
</tr>
<tr>
<td>ADF</td>
<td>Adaptive digital filter</td>
</tr>
<tr>
<td>ADPCM</td>
<td>Adaptive differential pulse code modulation</td>
</tr>
<tr>
<td>ADPLL</td>
<td>All-digital PLL</td>
</tr>
<tr>
<td>ADSP</td>
<td>Analog Devices digital signal processor family</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced encryption standard</td>
</tr>
<tr>
<td>AFT</td>
<td>Arithmetic Fourier transform</td>
</tr>
<tr>
<td>AHDL</td>
<td>Altera HDL</td>
</tr>
<tr>
<td>AHSM</td>
<td>Additive half square multiplier</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic logic unit</td>
</tr>
<tr>
<td>AM</td>
<td>Amplitude modulation</td>
</tr>
<tr>
<td>AMBA</td>
<td>Advanced microprocessor bus architecture</td>
</tr>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices, Inc.</td>
</tr>
<tr>
<td>ASCII</td>
<td>American standard code for information interchange</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-specific IC</td>
</tr>
<tr>
<td>AWGN</td>
<td>Additive white Gaussian noise</td>
</tr>
<tr>
<td>BCD</td>
<td>Binary coded decimal</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary decision diagram</td>
</tr>
<tr>
<td>BLMS</td>
<td>Block LMS</td>
</tr>
<tr>
<td>BP</td>
<td>Bandpass</td>
</tr>
<tr>
<td>BRSS</td>
<td>Base removal scaling</td>
</tr>
<tr>
<td>BS</td>
<td>Barrelshifter</td>
</tr>
<tr>
<td>CAE</td>
<td>Computer-aided engineering</td>
</tr>
<tr>
<td>CAM</td>
<td>Content addressable memory</td>
</tr>
<tr>
<td>CAST</td>
<td>Carlisle Adams and Stafford Tavares</td>
</tr>
<tr>
<td>CBC</td>
<td>Cipher block chaining</td>
</tr>
<tr>
<td>CBIC</td>
<td>Cell-based IC</td>
</tr>
<tr>
<td>CD</td>
<td>Compact disc</td>
</tr>
<tr>
<td>CFA</td>
<td>Common factor algorithm</td>
</tr>
<tr>
<td>CFB</td>
<td>Cipher feedback</td>
</tr>
<tr>
<td>CIC</td>
<td>Cascaded integrator comb</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex instruction set computer</td>
</tr>
<tr>
<td>CL</td>
<td>Costas loop</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable logic block</td>
</tr>
<tr>
<td>C-MOMS</td>
<td>Causal MOMS</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary metal oxide semiconductor</td>
</tr>
<tr>
<td>CODEC</td>
<td>Coder/decoder</td>
</tr>
<tr>
<td>CORDIC</td>
<td>Coordinate rotation digital computer</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial off-the-shelf technology</td>
</tr>
<tr>
<td>CPLD</td>
<td>Complex PLD</td>
</tr>
<tr>
<td>CPU</td>
<td>Central processing unit</td>
</tr>
<tr>
<td>CQF</td>
<td>Conjugate quadrature filter</td>
</tr>
<tr>
<td>CRNS</td>
<td>Complex RNS</td>
</tr>
<tr>
<td>CRT</td>
<td>Chinese remainder theorem</td>
</tr>
<tr>
<td>CSOC</td>
<td>Canonical self-orthogonal code</td>
</tr>
<tr>
<td>CSD</td>
<td>Canonical signed digit</td>
</tr>
<tr>
<td>CWT</td>
<td>Continuous wavelet transform</td>
</tr>
<tr>
<td>CZT</td>
<td>Chirp-z transform</td>
</tr>
<tr>
<td>DA</td>
<td>Distributed arithmetic</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital-to-analog converter</td>
</tr>
<tr>
<td>DAT</td>
<td>Digital audio tap</td>
</tr>
<tr>
<td>DB</td>
<td>Daubechies filter</td>
</tr>
<tr>
<td>DC</td>
<td>Direct current</td>
</tr>
<tr>
<td>DCO</td>
<td>Digital controlled oscillator</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete cosine transform</td>
</tr>
<tr>
<td>DCU</td>
<td>Data cache unit</td>
</tr>
<tr>
<td>DES</td>
<td>Data encryption standard</td>
</tr>
<tr>
<td>DFT</td>
<td>Discrete Fourier transform</td>
</tr>
<tr>
<td>DHT</td>
<td>Discrete Hartley transform</td>
</tr>
<tr>
<td>DIF</td>
<td>Decimation in frequency</td>
</tr>
<tr>
<td>DIT</td>
<td>Decimation in time</td>
</tr>
<tr>
<td>DLMS</td>
<td>Delayed LMS</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct memory access</td>
</tr>
<tr>
<td>DMIPS</td>
<td>Dhrystone MIPS</td>
</tr>
<tr>
<td>DMT</td>
<td>Discrete Morlet transform</td>
</tr>
<tr>
<td>DLL</td>
<td>Digital PLL</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital signal processing</td>
</tr>
<tr>
<td>DST</td>
<td>Discrete sine transform</td>
</tr>
<tr>
<td>DWT</td>
<td>Discrete wavelet transform</td>
</tr>
<tr>
<td>EAB</td>
<td>Embedded array block</td>
</tr>
<tr>
<td>ECB</td>
<td>Electronic code book</td>
</tr>
<tr>
<td>ECL</td>
<td>Emitter coupled logic</td>
</tr>
<tr>
<td>EDIF</td>
<td>Electronic design interchange format</td>
</tr>
<tr>
<td>EFF</td>
<td>Electronic Frontier Foundation</td>
</tr>
<tr>
<td>EPF</td>
<td>Altera FPGA family</td>
</tr>
<tr>
<td>EPROM</td>
<td>Electrically programmable ROM</td>
</tr>
<tr>
<td>ERA</td>
<td>Plessey FPGA family</td>
</tr>
<tr>
<td>ERNS</td>
<td>Eisenstein RNS</td>
</tr>
<tr>
<td>ESA</td>
<td>European Space Agency</td>
</tr>
<tr>
<td>EVR</td>
<td>Eigenvalue ratio</td>
</tr>
<tr>
<td>EXU</td>
<td>Execution unit</td>
</tr>
<tr>
<td>FAEST</td>
<td>Fast a posteriori error sequential technique</td>
</tr>
<tr>
<td>FCT</td>
<td>Fast cosine transform</td>
</tr>
<tr>
<td>FC2</td>
<td>FPGA compiler II</td>
</tr>
<tr>
<td>FF</td>
<td>Flip-flop</td>
</tr>
</tbody>
</table>
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
FSF  Frequency sampling filter
FSK  Frequency shift keying
FSM  Finite state machine
GAL  Generic array logic
GF  Galois field
GNU  GNU’s not Unix
GPP  General-purpose processor
GPR  General-purpose register
HB  Half-band filter
HI  High frequency
HDL  Hardware description language
HSP  Harris Semiconductor DSP ICs
IBM  International Business Machines (corporation)
IC  Integrated circuit
ICU  Instruction cache unit
IDCT  Inverse DCT
IDEA  International data encryption algorithm
IDFT  Inverse discrete Fourier transform
IEEE  Institute of Electrical and Electronics Engineers
IF  Inter frequency
IFFT  Inverse fast Fourier transform
IIR  Infinite impulse response
I-MOMS  Interpolating MOMS
INTT  Inverse NTT
IP  Intellectual property
I/Q  In-/Quadrature phase
ISA  Instruction set architecture
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
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 Lowpass
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

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
µP 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
NP Nonpolynomial complex problem
NRE Nonrecurring engineering costs
NTT Number theoretic transform

OFB Open feedback (mode)
O-MOMS Optimal MOMS

PAM Pulse-amplitude modulated
PC Personal computer
PCI Peripheral component interconnect
PD Phase detector
PDSP Programmable digital signal processor
PFA Prime factor algorithm
PIT Programmable interval timer
PLA Programmable logic array
PLD Programmable logic device
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL</td>
<td>Phase-locked loop</td>
</tr>
<tr>
<td>PM</td>
<td>Phase modulation</td>
</tr>
<tr>
<td>PREP</td>
<td>Programmable Electronic Performance (cooperation)</td>
</tr>
<tr>
<td>PRNS</td>
<td>Polynomial RNS</td>
</tr>
<tr>
<td>PROM</td>
<td>Programmable ROM</td>
</tr>
<tr>
<td>PSK</td>
<td>Phase shift keying</td>
</tr>
<tr>
<td>PSM</td>
<td>Programmable state machine</td>
</tr>
<tr>
<td>QDFT</td>
<td>Quantized DFT</td>
</tr>
<tr>
<td>QLI</td>
<td>Quick look-in</td>
</tr>
<tr>
<td>QFFT</td>
<td>Quantized FFT</td>
</tr>
<tr>
<td>QMF</td>
<td>Quadrature mirror filter</td>
</tr>
<tr>
<td>QRNS</td>
<td>Quadratic RNS</td>
</tr>
<tr>
<td>QSM</td>
<td>Quarter square multiplier</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-access memory</td>
</tr>
<tr>
<td>RC</td>
<td>Resistor/capacity</td>
</tr>
<tr>
<td>RF</td>
<td>Radio frequency</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced instruction set computer</td>
</tr>
<tr>
<td>RLS</td>
<td>Recursive least square</td>
</tr>
<tr>
<td>RNS</td>
<td>Residue number system</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-only memory</td>
</tr>
<tr>
<td>RPFA</td>
<td>Rader prime factor algorithm</td>
</tr>
<tr>
<td>RS</td>
<td>Serial right shifter</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest, Shamir, and Adelman</td>
</tr>
<tr>
<td>SD</td>
<td>Signed digit</td>
</tr>
<tr>
<td>SG</td>
<td>Stochastic gradient</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single instruction multiple data</td>
</tr>
<tr>
<td>SLMS</td>
<td>Signed LMS</td>
</tr>
<tr>
<td>SM</td>
<td>Signed magnitude</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal-to-noise ratio</td>
</tr>
<tr>
<td>SPEC</td>
<td>System performance evaluation cooperation</td>
</tr>
<tr>
<td>SPLD</td>
<td>Simple PLD</td>
</tr>
<tr>
<td>SPT</td>
<td>Signed power-of-two</td>
</tr>
<tr>
<td>SR</td>
<td>Shift register</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static random-access memory</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming SIMD extension</td>
</tr>
<tr>
<td>STFT</td>
<td>Short-term Fourier transform</td>
</tr>
<tr>
<td>TDLMS</td>
<td>Transform-domain LMS</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation look-aside buffer</td>
</tr>
<tr>
<td>TLU</td>
<td>Table look-up</td>
</tr>
<tr>
<td>TMS</td>
<td>Texas Instruments DSP family</td>
</tr>
<tr>
<td>TI</td>
<td>Texas Instruments</td>
</tr>
<tr>
<td>TOS</td>
<td>Top of stack</td>
</tr>
<tr>
<td>TTL</td>
<td>Transistor transistor logic</td>
</tr>
<tr>
<td>TVP</td>
<td>True vector processor</td>
</tr>
<tr>
<td>UART</td>
<td>Universal asynchronous receiver/transmitter</td>
</tr>
<tr>
<td>VCO</td>
<td>Voltage-control oscillator</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC hardware description language</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very-high-speed integrated circuit</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very long instruction word</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very large integrated ICs</td>
</tr>
<tr>
<td>WDT</td>
<td>Watchdog timer</td>
</tr>
<tr>
<td>WFTA</td>
<td>Winograd Fourier transform algorithm</td>
</tr>
<tr>
<td>WSS</td>
<td>Wide sense stationary</td>
</tr>
<tr>
<td>XC</td>
<td>Xilinx FPGA family</td>
</tr>
<tr>
<td>XNOR</td>
<td>Exclusive NOR gate</td>
</tr>
<tr>
<td>YACC</td>
<td>Yet another compiler-compiler</td>
</tr>
</tbody>
</table>
D. CD-ROM File: “1readme.ps”

The accompanying CD-ROM includes
- A full version of the Quartus II software
- Altera datasheets for Cyclone II devices
- All VHDL/Verilog design examples and utility programs and files

To install the Quartus II 6.0 web edition software first read the licence agreement carefully. Since the Quartus II 6.0 web edition software uses many other tools (e.g., GNU, Berkeley Tools, SUN microsystems tool, etc.) you need to agree to their licence agreements too before installing the software. To install the software start the self-extracting file `quartusii_60_web_edition.exe` on the CD-ROM in the `Altera` folder. After the installation the user must register the software through Altera’s web page at `www.altera.com` in order to get a permanent licence key. Otherwise the temporary licence key expires after the 30-day grace period and the software will no longer run. Altera frequently update the Quartus II software to support new devices 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. Altera’s University program now delivers the files via download, which can take long time with a 56 Kbit/s MODEM.

The design examples for the book are located in the directories `book3e/vhdl` and `book3e/verilog` for the VHDL and Verilog examples, respectively. These directories contain, for each example, the following four files:
- The VHDL or Verilog source code (`*.vhd` and `*.v`)
- The Quartus project files (`*.qpf`)
- The Quartus setting files (`*.qsf`)
- The Simulator wave form file (`*.vwf`)

For the design `fun_graf`, the block design file (`*.bdf`) is included in `book3e/vhdl`. For the examples that utilize M4Ks (i.e., `fun_text`, `darom`, and `trisc0`), the memory initialization file (`*.mif`) can be found on the CD-ROM. To simplify the compilation and postprocessing, the source code directories include the additional (`*.bat`) files and Tcl scripts shown below:
### File Comment

<table>
<thead>
<tr>
<th>File</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>qvhdl.tcl</td>
<td>Tcl script to compile all design examples. Note that the device can be changed from Cyclone II to Flex, Apex or Stratix just by changing the comment sign # in column 1 of the script.</td>
</tr>
<tr>
<td>qclean.bat</td>
<td>Cleans all temporary Quartus II compiler files, but not the report files (<em>.map.rpt), the timing analyzer output files (</em>.tan.rpt), and the project files *.qpf and *.qsf.</td>
</tr>
<tr>
<td>qveryclean.bat</td>
<td>Cleans all temporary compiler files, including all report files (*.rep) and project files.</td>
</tr>
</tbody>
</table>

Use the DOS prompt and type

```
quartus_sh -t qvhdl.tcl > qvhdl.txt
```

... to compile all design examples and then `qclean.bat` to remove the unnecessary files. The Tcl script `qvhdl.tcl` is included on the CD. The Tcl script language developed by the Berkeley Professor John Ousterhout [346, 347, 348] (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 looks almost the same.

Two search procedures (`show_fmax` and `show_resources`) are used within the Tcl script `qvhdl.tcl` to display resources and Registered Performance. The script includes all settings and also alternative device definitions. The protocol file `qvhdl.txt` has all the useful synthesis data. For the trisc0 processor, for instance, the list for the Cyclone II device EP2C35F672C6 is:

```
....
----------------------------------------
trisc0 fmax: 115.65 MHz ( period = 8.647 ns )
trisc0 LEs: 198 / 33,216 ( < 1 % )
trisc0 M4K bits: 5,120 / 483,840 ( 1 % )
trisc0 DSP blocks: 1 / 70 ( 1 % )
----------------------------------------
....
```

The results for all examples are summarized in Table B.1, p. 731.

Other devices are prespecified and include the EPF10K20RC240-4 and EPF10K70RC240-4 from the UP1 and UP2 University boards, the EP20K200EFC484-2X from the Nios development boards, and three devices from other DSP boards available from Altera, i.e., the EP1S10F484C5, EP1S25F780C5, and EP2S60F1020C4ES.

### Using Compilers Other Then Quartus II

**Synopsys FPGA_ComplierII**

The main advantage of using the FPGA_ComplierII (FC2) from Synopsys was that it was possible to synthesize examples for other devices like Xilinx, Vantis, Actel, or QuickLogic with the same tool. The Tcl scripts `vhdl.fc2`, and `verilog.fc2`,...
respectively, were provided the necessary commands for the shell mode of FC2, i.e.,
fc2_shell in the second edition of the book [57]. Synopsys, however, since 2006 no
longer supports the FPGA_CompilerII and it is therefore not a good idea to use
the compiler anymore since the newer devices can not be selected.

Model Technology

By using the synthesizable public-domain models provided by the EDIF organiza-
tion (at www.edif.org), it is also possible to use other VHDL/Verilog simulators
then Quartus II.

Using MTI and VHDL. For VHDL, the two files 220pack.vhd and
220model.vhd must first be compiled. For the ModelSim simulator vsim from Model
Technology Inc., the script mti_vhdl.do can be used for device-independent com-
pilation and simulation of the design examples. The script is shown below:

```bash
#----------------------------------------------------------
# Model Technology VHDL compiler script for the book
# Digital Signal Processing with FPGAs (3.edition)
# Author-EMAIL: Uwe.Meyer-Baese@ieee.org
#----------------------------------------------------------

echo Create Library directory lpm
vlib lpm

echo Compile lpm package.
vcom -work lpm -explicit -quiet 220pack.vhd 220model.vhd

echo Compile chapter 1 entitys.
vcom -work lpm -quiet example.vhd fun_text.vhd

echo Compile chapter 2 entitys.
vcom -work lpm -explicit -quiet add1p.vhd add2p.vhd
vcom -work lpm -explicit -quiet add3p.vhd mul_ser.vhd
vcom -work lpm -explicit -quiet cordic.vhd

echo Compile chapter 3 components.
vcom -work lpm -explicit -quiet case3.vhd case5p.vhd
vcom -work lpm -explicit -quiet case3s.vhd
echo Compile chapter 3 entitys.
vcom -work lpm -explicit -quiet fir_gen.vhd fir_srg.vhd
vcom -work lpm -explicit -quiet dafsm.vhd darom.vhd
vcom -work lpm -explicit -quiet dasign.vhd dapara.vhd

echo Compile chapter 4 entitys.
vcom -work lpm -explicit -quiet iir.vhd iir_pipe.vhd
vcom -work lpm -explicit -quiet iir_par.vhd

echo Compile chapter 5 entitys.
vcom -work lpm -explicit -quiet cic3r32.vhd cic3s32.vhd
vcom -work lpm -explicit -quiet db4poly.vhd db4latti.vhd

echo Compile chapter 6 entitys.
```

D. CD-ROM File: “1readme.ps” 763
vcom -work lpm -explicit -quiet rader7.vhd ccmul.vhd
vcom -work lpm -explicit -quiet bfproc.vhd

echo Compile chapter 7 entitys.
vcom -work lpm -explicit -quiet rader7.vhd ccmul.vhd
vcom -work lpm -explicit -quiet bfproc.vhd

echo Compile 2. edition entitys.
vcom -work lpm -explicit -quiet div_res.vhd div_aegp.vhd
vcom -work lpm -explicit -quiet fir_lms.vhd fir61dms.vhd

echo Compile 3. edition entitys from chapter 2.
vcom -work lpm -explicit -quiet cmul7p8.vhd arctan.vhd
vcom -work lpm -explicit -quiet ln.vhd sqrt.vhd

echo Compile 3. edition entitys from chapter 5.
vcom -work lpm -explicit -quiet rc_sinc.vhd farrow.vhd
vcom -work lpm -explicit -quiet cmoms.vhd

vcom -work lpm -explicit -quiet reg_file.vhd trisc0.vhd

Start the ModelSim simulator and then type

domti_vhdl1.do

to execute the script.

Using MTI and Verilog. Using the Verilog interface with the lpm library from EDIF, i.e., 220model.v, needs some additional effort. When using 220model.v it is necessary to specify all ports in the Verilog lpm components. There is an extra directory book3e/verilog/mti, which provides the design examples with a full set of lpm port specifications. The designs use

`\include "220model.v"

at the beginning of each Verilog file to include the lpm components, if necessary. Use the script mti_v1.csh and mti_v2.csh to compile all Verilog design examples with Model Technology’s vcom compiler.

In order to load the memory initialization file (*.mif), it is required to be familiar with the programming language interface (PLI) of the Verilog 1364-1995 IEEE standard (see LRM Sect. 17, p. 228 ff). With this powerful PLI interface, conventional C programs can be dynamically loaded into the Verilog compiler. In order to generate a dynamically loaded object from the program convert_hex2ver.c, the path for the include files veriuser.h and acc_user.h must be specified. Use -I when using the gcc or cc compiler under SUN Solaris. Using, for instance, the gcc compiler under SUN Solaris for the Model Technology Compiler, the following commands are used to produce the shared object:

gcc -c -I/<install_dir>/modeltech/include convert_hex2ver.c
ld -G -B symbolic -o convert_hex2ver.sl convert_hex2ver.o

By doing so, ld will generate a warning “Symbol referencing errors,” because all symbols are first resolved within the shared library at link time, but these warnings can be ignored.

It is then possible to use these shared objects, for instance, with Model Technology’s vsim in the first design example fun_text.v, with
To learn more about PLIs, check out the Verilog IEEE standard 1364-1995, or the vendor's user manual of your Verilog compiler.

We can use the script `mti_v1.do` to compile all Verilog examples with MTI's `vlog`. Just type

```
do mti_v1.do
```

in the ModelTech command line. But `vlog` does not perform a check of the correct component port instantiations or shared objects. A second script, `mti_v2.do`, can be used for this purpose. Start the `vsim` simulator (without loading a design) and execute the `DO` file with

```
do mti_v2.do
```

to perform the check for all designs.

**Using Xilinx ISE** The conversion of designs from Altera Quartus II to Xilinx ISE seems to be easy if we use standard HDL. Unfortunately there a couple of issues that needs to be addressed. We assume that the ModelTech simulation environment and the web version (i.e., no core generation) is used. We like to discuss in the following a couple of items that address the ISE/ModelTech design entry. We describe the Xilinx ISE 6.2 web edition and ModelTech 5.7g version.

1) The Xilinx simulation with timing (“Post-Place & Route”) uses a bitwise simulation model on the LUT level. Back annotations are only done for the I/O ports, and are ALL from type `standard_logic` or `standard_logic_vector`. In order to match the behavior and the simulation with timing we therefore need to use only the `standard_logic` or `standard_logic_vector` data type for I/O. As a consequence no integers, generic, or custom I/O data type, (e.g., like the subtype byte see `cordic.vhd`) can be used.

2) The ISE software supports the development of testbenches with the “Test Bench Waveform.” Use `New Source...` under the `Project` menu. This waveform will give you a quick way to generate a testbench that is used by ModelTech, both for behavior as well as simulation with timing. There are some benefits and drawbacks with the testbencher. For instance, you can not assign negative integers in the waveforms, you need to build the two’s complement, i.e., equivalent unsigned number by hand.

3) If you have feedback, you need to initialize the register to zero in your HDL code. You can not do this with the testbencher: for instance, ModelTech initialize all integer signals to the smallest value, i.e., $-128$ for a 8-bit number, if you add two integers, the result will be $-128 - 128 = -256 < -128$ and ModelTech will stop and report an overflow. Some designs, e.g., `cordic`, `cic3r32`, `cic3s32`, only work correctly in behavior simulation if all integers are changed to `standard_logic_vector` data type. Changing I/O ports alone and using the conversion function does not always guarantee correct simulation results.

4) Simulation with timing usually needs one clock cycle more than behavior code until all logic is settled. The input stimuli should therefore be zero in the first clock cycle (ca. 100 ns) and, if you want to match behavior and timing simulation, and the design uses a (small) FSM for control, you need to add a synchronous or asynchronous reset. You need to do this for the following 2/e designs: `dafsm`, `dadrom`, `dasign`, `db4latti`, `db4poly`, `div_aegp`, `div_res`, `iir_par`, `mul_ser`, `rader7`. Just add a control part for the FSM like this:
-- IF rising_edge(clk) THEN -- Synchronous reset
--  IF reset = '1' THEN
--    state <= s0;
-- ELSE
--  IF reset = '1' THEN -- Asynchronous reset
--    state <= s0;
--  ELSIF rising_edge(clk) THEN
--    CASE state IS
--      WHEN s0 => -- Initialization step
--          ...

Although at first glance this synchronous or asynchronous control seems to be cheap because the FSM is small, we need to keep in mind that, if the reset is active, all signals that are assigned in the state s0 of the FSM need to be preserved with their initial state value. The following table shows the synthesis results for the three different reset styles for the design file dafsm.vhd (small distribute arithmetic state machine):

<table>
<thead>
<tr>
<th>Reset style</th>
<th>Performance/ns</th>
<th>4-input LUT</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>No reset (original code)</td>
<td>3.542</td>
<td>20</td>
<td>339</td>
</tr>
<tr>
<td>synchronous</td>
<td>3.287</td>
<td>29</td>
<td>393</td>
</tr>
<tr>
<td>asynchronous</td>
<td>3.554</td>
<td>29</td>
<td>393</td>
</tr>
</tbody>
</table>

Designs with reset usually have a higher LUT and gate count. Depending on the design, synchronous or asynchronous reset can also have a (small) influence on performance.

5) Back annotation is only done for I/O ports. If we want to monitor internal nets, we can try to find the appropriate net name in the *.timesim.vhd file, but that is quite complicated and may change in the next compiler run. A better idea is to introduce additional test outputs, see, for instance, fir_lms.vhd for f1_out and f2_out. In the behavioral (but not in the timing) simulation internal test signals and variables can be monitored. Modify the *.udo file and add, for instance for the fir_srg_tb.vhd file, add wave /fir_srg_tb/uut/tap to the testbench.

3) There are a couple of nice features in the Xilinx ISE package too: there is no need for special lpm blocks to use the internal resources for multiplier, shifter, RAMs or ROMs. Some other features are:
   a) ISE converts a shift register in a single CLB-based shift register. This can save some resources.
   b) Multipliers can be implemented with LUTs only, including block multipliers (if available) or even pipelined LUTs, which is done via pipeline retiming. Just right click on the Synthesize-XST menu in the Processes, select HDL Options under Process Properties and the last entry is the multiplier style. But note that for pipelined LUT design the additional register must be placed at the output of the multiplier. Pipeline retiming is not done if the additional registers are at the inputs. You need about log₂(B) additional registers to have good timing (see Chap. 2 on pipeline multiplier). This has an impact on the Registered Performance, LUT usage, and gates as the following table shows for the fir_gen.vhd example, i.e., length 4 programmable FIR filter (from Chap. 3):
Synthesis style & Speed in ns & 4-input LUT & mul. blocks & Gates  
--- & --- & --- & --- & ---  
Block multiplier & 9.838 & 57 & 4 & 17552  
LUT (no pipeline) & 15.341 & 433 & 0 & 6114  
LUT (3 stage pipeline) & 6.762 & 448 & 0 & 9748  

For this multiplier size (9 bit) the pipelined LUT seems to be attractive, both for speed as well as gate count. If the number of LUTs is limited, the block multiplier provides the next best alternative.

c) If you follow the recommended style the Xilinx software synthesis tool (see XST manual and ISE help “Inferring BlockRAM in VHDL”) maps your HDL code to the block RAM (see, `fun_text.vhd`). If the table is small, the ISE auto option selects the LUT-based implementation for a ROM table (see `daron.vhd`). You can also initialize the table in the HDL code and use it as a ROM. Please see the XST manual Chap. 3, “FPGA Optimization” for details on ROM implementation. There are some limitations that apply to the initialization of BlockRAMs (see, XST Chap. 2)

### Utility Programs and Files

A couple of extra utility programs are also included on the CD-ROM\(^1\) and can be found in the directory `book3e/util`:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sine3e.exe</td>
<td>Program to generate the MIF files for the function generator in Chap. 1</td>
</tr>
<tr>
<td>csd3e.exe</td>
<td>Program to find the canonical signed digit representation of integers or fractions as used in Chap. 2</td>
</tr>
<tr>
<td>fpinv3e.exe</td>
<td>Program to compute the floating-point tables for reciprocals as used in Chap. 2</td>
</tr>
<tr>
<td>dagen.exe</td>
<td>Program to generate the VHDL code for the distributed arithmetic files used in Chap. 3</td>
</tr>
<tr>
<td>ragopt.exe</td>
<td>Program to compute the reduced adder graph for constant-coefficient filters as used in Chap. 3. It has 10 predefined lowpass and half-band filters. The program uses a MAG cost table stored in the file <code>mag14.dat</code></td>
</tr>
<tr>
<td>cic.exe</td>
<td>Program to compute the parameters for a CIC filter as used in Chap. 5</td>
</tr>
</tbody>
</table>

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

Also under `book3e/util` we find the following utility files:

\(^1\) You need to copy the programs to your harddrive first; you can not start them from the CD directly since the program write out the results in text files.
<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quickver.pdf</td>
<td>Quick reference card for Verilog HDL from QUALIS</td>
</tr>
<tr>
<td>quickvhd.pdf</td>
<td>Quick reference card for VHDL from QUALIS</td>
</tr>
<tr>
<td>quicklog.pdf</td>
<td>Quick reference card for the IEEE 1164 logic package from QUALIS</td>
</tr>
<tr>
<td>93vhdl.vhd</td>
<td>The IEEE VHDL 1076-1993 keywords</td>
</tr>
<tr>
<td>95key.v</td>
<td>The IEEE Verilog 1364-1995 keywords</td>
</tr>
<tr>
<td>01key.v</td>
<td>The IEEE Verilog 1364-2001 keywords</td>
</tr>
<tr>
<td>95direct.v</td>
<td>The IEEE Verilog 1364-1995 compiler directives</td>
</tr>
<tr>
<td>95tasks.v</td>
<td>The IEEE Verilog 1364-1995 system tasks and functions</td>
</tr>
</tbody>
</table>

In addition, the CD-ROM includes a collection of useful Internet links (see file dsp4fpga.htm under book3e/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.

### Microprocessor Project Files and Programs

All microprocessor-related tools and documents can be found in the book3e/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** is an infix calculator 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 book3e/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.
Index

Accumulator 10, 257
- µP 553, 557
Actel 9
Adaptive filter 477–535
Adder
- binary 75
- fast carry 76
- floating-point 110, 114
- LPM 15, 30, 78, 368, 733, 737
- pipelined 78
- size 77
- speed 77
Agarwal–Burrus NTT 410
Algorithms
- Bluestein 350
- chirp-z 350
- Cooley–Tukey 367
- CORDIC 120–130
- common factor (CFA) 362
- Goertzel 350
- Good–Thomas 363
- fast RLS 10
- LMS 488, 531
- prime factor (PFA) 362
- Rader 353, 413
- Radix-r 366
- RLS 522, 531
- Widrow–Hoff LMS 488
- Winograd DFT 360
- Winograd FFT 376
Altera 9, 20
AMD 9
Arbitrary rate conversion 280–308
Arctan approximation 132
ARM922T µP 592

Bartlett window 175, 345
Bijective 259
Bison µP tool 567, 578
Bitreverse 390
Blackman window 175, 345
Blowfish 452
B-spline rate conversion 296
Butterfly 366, 370
CAST 452
C compiler 586, 587
Chebyshev series 131
Chirp-z algorithm 350
CIC filter 258–273
- RNS design 260
- interpolator 340
Coding bounds 423
Codes
- block
- decoders 426
- encoder 425
- convolutional
- comparison 436
- complexity 435
- decoder 430, 434
- encoder 430, 434
- tree codes 429
Contour plot 490
Convergence 490, 491, 492
- time constant 491
Convolution
- Bluestein 391
- cyclic 391
- linear 116, 165
Cooley–Tukey
- FFT 367
- NTT 409
CORDIC algorithm 120–130
cosine approximation 137
Costas loop
- architecture 470
- demodulation 470
- implementation 472
CPLD 6, 5
Index 771

- linear-phase 327
- lifting 321
- QMF 314
- orthogonal 323
- perfect reconstruction 316
- polyphase 323
- mirror frequency 314
- comparison 328
Filter design
- Butterworth 222
- Chebyshev 223
- Comparison of FIR to IIR 216
- elliptic 222
- equiripple 177
- frequency sampling 277
- Kaiser window 174
- Parks–McClellan 177
Finite impulse response (FIR), see Filter 165–204
Flex µP tool 567, 571
Flip-flop
- LPM 15, 30, 78, 78, 78, 733
Floating-point
- addition 110
- arithmetic 104
- conversion to fixed-point 106
- division 111
- multiplication 108
- numbers 71
- reciprocal 113
- synthesis results 114
FPGA
- Altera’s Cyclone II 22
- architecture 6
- benchmark 10
- Compiler II 762
- design compilation 33
- floor plan 33
- graphical design entry 30
- performance analysis 36
- power dissipation 13
- registered performance 36
- routing 5, 23
- simulation 34
- size 20, 22
- technology 9
- timing 26
- waveform files 43
- Xilinx Spartan-3 20
FPL, see FPGA and CPLD
Fractal 336
Fractional delay rate conversion 284
Frequency sampling filter 277

Function approximation
- arctan 132
- cosine 137
- Chebyshev series 131
- exponential 141
- logarithmic 145
- sine 137
- square root 150
- Taylor series 121

Galois Field 423
Gauss primes 69
General-purpose µP 538, 588
Generator 67, 68
Gibb’s phenomenon 174
Good–Thomas
- FFT 363
- NTT 409
Goodman/Carey half-band filter 275, 318, 337
Gradient 487

Hadamard 474
half-band filter
- decimator 276
- factorization 317
- Goodman and Carey 275, 318
- definition 274
Hamming window 175, 345
Hann window 175, 345
Harvard µP 558
Hogenauer filter, see CIC
Homomorphism 258

IDEA 452
Identification 480, 494, 503
Isomorphism 258
Image compression 387
Index 67
- multiplier 68
- maps
- in FFTs 362
- in NTTs 409
Infinite impulse response (IIR) filter
216–239
- finite wordlength effects 228
- fast filtering using
- time-domain interleaving 231
- clustered look-ahead pipelining 233
- scattered look-ahead pipelining 234
- decimator design 235
- parallel processing 237
- RNS design 239
In-place 381
Instruction set design 544
Intel 539
Interference cancellation 478, 522
Interpolation
   - CIC 340
   - see rate conversion
Inverse
   - multiplicative 363
   - additive 404
   - system modeling 480
IP core
   - FFT 383
   - FIR filter 205
   - NCO 35
JPEG, see Image compression
Kaiser
   - window 345
   - window filter design 175
Kalman gain 520, 522, 526
Kronecker product 376
Learning curves 493
   - RLS 520, 523
Lexical analysis (see Flex)
LISA µP 567, 610–626
LPM
   - add, sub 15, 30, 78, 368, 733, 737
   - divider 103, 733, 749
   - flip-flop 15, 30, 78, 78, 78, 733
   - multiplier 167, 504, 511, 368, 733, 741
   - RAM 606
   - ROM 30, 196, 733, 746
Lifting 321
Linear feedback shift register 438
LMS algorithm 488, 531
   - normalized 496, 498
   - design 506,
     - pipelined 508
     - delayed 508
     - design 511, 514
     - look-ahead 510
     - transposed 511
     - block FFT 500
     - simplified 516, 517
     - error floor 516
Logarithmic approximation 145
MAC 78
Mersenne NTT 408
MicroBlaze µP 603
Microprocessor
   - Accumulator 553, 557
   - Bison tool 567, 578
   - C compiler 586, 587
   - DWT 610
   - GPP 538, 588
   - Instruction set design 544
     - Profile 611, 615, 618, 624
   - Intel 539
   - FFT co-processor 627
   - Flex tool 567, 571
   - Lexical analysis (see Flex)
     - LISA 567, 610–626
   - Hardcore
     - PowerPC 591
     - ARM922T 592
     - Harvard 558
   - Softcore
     - MicroBlaze 603
     - Nios 598
     - PicoBlaze 538, 595
     - Parser (see Bison)
     - PDSP 2, 12, 114, 550, 616
     - RISC 540
     - register file 559
     - Stack 553, 557, 606
     - Super Harvard 558
     - Three address 555, 557
     - Two address 555, 557
     - Vector 620
     - Von-Neuman 558
Möbius function 416
MOMS rate conversion 301
Multiplier
   - adder graph 184, 229
   - array 84
   - block 91
   - Booth 154
   - complex 156, 368
   - FPGA array 85
   - floating-point 108, 114
   - half-square 88
   - index 68
   - LPM 167, 504, 511, 368, 733, 741
   - performance 86
   - QRNS 69
   - quarter square 90, 239
   - serial/parallel 83
   - size 87
Modulation 453
   - using CORDIC 457
Modulo
   - adder 68
- multiplier 68,
- reconstruction 273

NAND 5, 42
NCO IP core 35
Nios µP 598

Number representation
- canonical signed digit (CSD) 58, 229
- diminished by one (D1) 57, 405
- floating-point 74
- fractional 59, 178
- one’s complement (1C) 57, 405
- two’s complement (2C) 57
- sign magnitude (SM) 57

Number theoretic transform 401–417
- Agarwal–Burrus 410
- convolution 405
- definition 401
- Fermat 407
- Mersenne 408
- wordlength 408

Order
- filter 166
- for NTTs 408
Ordering, see index map

Orthogonal
- wavelet transform 319
- filter bank 323

Parser (see Bison)
Perfect reconstruction 316
Phase-locked loop (PLL)
- with accumulator reference 461
- demodulator 466
- digital 468
- implementation 467, 469
- linear 465
PicoBlaze µP 538, 595
Plessey ERA 5
Pole/zero diagram 236, 323
Polynomial rate conversion 290
Polyphase representation 250, 320

Power
- dissipation 13
- estimation 496, 498
- line hum 485, 487, 490, 492, 505, 516
PowerPC µP 591
Prediction 479
- forward 525
- backward 527
Prime number
- Fermat 403

- Mersenne 403
- Primitive element 67
- Programmable signal processor 2, 12, 114, 550, 616
- addressing generation 551
- Public key systems
- Quadratic RNS (QRNS) 69
- Quadrature Mirror Filter (QMF) 314
- Rader
- DFT 363
- NTT 413
- Rate conversion
- arbitrary 280–308
- B-spline 296
- Farrow 292
- FFT-based 282
- fractional delay 284
- MOMS 301
- polynomial 290
- rational 249
- Rational rate conversion 249
- RC5 452
- Rectangular window 175, 345
- Reduced adder graph 184, 229
- RISC µP 540
- register file 559
- RLS algorithm 518, 522, 529
- RNS
- CIC filter 260
- complex 70
- IIR filter 239
- Quadratic 69
- scaling 273
- ROM
- LPM 30, 196, 733, 746
- RSA 452

Sampling
- Frequency 345
- Time 345
- see rate conversion
- Sea of gates Plessey ERA 5
- Self-similar 332
- Sine approximation 137

Simulator
- ModelTechnology 618, 763
- Square root approximation 150
- Stack µP 553, 557, 606
- Step size 492, 492, 493, 502
- Subband filter 309
- Super Harvard µP 558
Symmetry
- in filter 172
- in cryptographic algorithms 452

Synthesizer
- accumulator 29
- PLL with accumulator 461

Taylor series 121

Theorem
- Chinese remainder 67

Three address µP 555, 557

Two-channel filter bank 314–328
- comparison 328
- lifting 321
- orthogonal 323
- QMF 323
- polyphase 320

Transformation
- arithmetic Fourier 417
- continuous Wavelet 332
- discrete cosine 390
- discrete Fourier 344
- inverse (IDFT) 344
- discrete Hartley 393
- discrete Wavelet 332–337
- domain LMS 500
- Fourier 345
- Fermat NTT 407
- pseudo-NTT 409
- short-time Fourier (STFT) 329
- discrete sine 387
- Mersenne NTT 408
- number theoretic 401–417
- Walsh–Hadamard 417

Triple DES 451

Two address µP 555, 557

Vector µP 620

Verilog
- key words 729

VHDL
- styles 15
- key words 729

Von-Neuman µP 558

Walsh 473

Wavelets 332–337
- continuous 332
- linear-phase 327
- LISA processor 610–626
- orthogonal 319

Widrow–Hoff LMS algorithm 488

Wiener–Hopf equation 484