

Introduction

Your hardware description language (HDL) coding style can have a big effect on the quality of results (QoR) that you achieve for your Altera® design. This chapter discusses coding style recommendations to ensure optimal synthesis results when targeting Altera devices. This chapter provides code examples for inferring Altera megafunctions from HDL code, and targeting certain functions in Altera device architectures, along with some general coding guidelines.

Altera Megafunctions

Altera provides parameterizable megafunctions that are optimized for Altera device architectures. Using megafunctions instead of coding your own logic saves valuable design time. Additionally, the Altera-provided functions may offer more efficient logic synthesis and device implementation. You can scale the megafunction's size by simply setting parameters.

Megafunctions include the library of parameterized modules (LPM), device-specific megafunctions, intellectual property (IP) available as Altera MegaCore® functions, and IP available from Altera Megafunction Partners Program (AMPPSM) partners.



You must use megafunctions to access some Altera device-specific features, such as memory, digital signal processing (DSP) blocks, low-voltage differential signal (LVDS) drivers, phase-locked loops (PLLs), transceivers, and double data rate input/output (DDIO) circuitry.

Altera megafunctions are easy to instantiate and offer efficient device implementation. However, for some designs, generic HDL code may provide better results. The following general guidelines provide some examples:

- For simple addition or subtraction functions, use the + or - symbol instead of an LPM function. The LPM implementation for simple arithmetic functions may result in a less efficient result because the function will be hard-coded and the synthesis algorithms cannot take advantage of basic logic optimizations. For more complicated arithmetic such as synchronous loadable counters, LPM functions may offer the most efficient results.

- For simple multiplexers and decoders, use array notation (such as `out = data [sel]`) instead of an LPM function. Array notation works very well and has simple syntax. You may want to use the `LPM_MUX` function to take advantage of architectural features such as cascade chains in APEX™ devices, but use the LPM function only if you want to force a specific implementation.
- Avoid division operations where possible. Division is an inherently slow operation. Many designers use multiplications creatively to produce division results. If you must divide, the `LPM_DIVIDE` function provides the best results possible.

Instantiating versus Inferring Altera Megafunctions

This section describes how to use megafunctions by instantiating them in your HDL code or inferring them from generic HDL code.

Instantiating Altera Megafunctions in HDL Code

If you decide to instantiate a megafunction in your HDL code, use one of the following methods:

- Use the Quartus® II software MegaWizard® Plug-In Manager to parameterize the function and create a wrapper file.
- Instantiate the function using the port and parameter definitions.

Instantiating Megafunctions Using the MegaWizard Plug-In Manager

Altera recommends that you use the MegaWizard Plug-In Manager to instantiate megafunctions. The wizard provides a graphical interface to customize and parameterize megafunctions, and ensures that you set all megafunction parameters properly. When you finish setting parameters you can specify which files should be generated. The wizard generates an Altera HDL (AHDL) or Verilog HDL (VHDL) wrapper file that instantiates the megafunction with the correct parameters, as well as a Component Declaration File (**.cmp**) for VHDL and an Include File (**.inc**) for AHDL. You can then instantiate the wrapper file in your HDL code.



Altera strongly recommends that you use the wizard for complex megafunctions such as PLLs, transceivers, and LVDS drivers.

For certain megafunctions, you also have the option of creating a clear box body instead of a wrapper file. The clear box netlist file is a fully synthesizable Altera megafunction, or LPM function, for use with electronic design automation (EDA) synthesis tools. When implementing a megafunction with the clear box model, you provide the EDA synthesis

tool with knowledge of the architectural details used in the Quartus II software. This enables certain synthesis tools to report better timing and resource utilization estimates.

To generate a clear box model, select **Generate a clear box body (for EDA tools only)** option from the first page of the MegaWizard Plug-in Manager.

Table 2–1 lists and describes the MegaWizard Plug-In Manager-generated files.

File	Description
<output file>.bsf	Block Symbol File used in the Quartus II schematic editor
<output file>.cmp	Component Declaration File used in VHDL designs.
<output file>.inc	Include File used in AHDL designs.
<output file>.tdf (1)	Megafunction wrapper file for instantiation in an AHDL design.
<output file>.vhd (2) (4)	Megafunction wrapper file, or clear box netlist file, for instantiation in a VHDL design.
<output file>.v (3) (4)	Megafunction wrapper file, or clear box netlist file, for instantiation in a Verilog HDL design.
<output file>_bb.v (3)	Hollow-body declaration used in Verilog HDL designs to specify port directions when black-boxing in third-party synthesis tools.
<output file>_inst.tdf (2)	Sample AHDL instantiation of the subdesign in the megafunction wrapper file.
<output file>_inst.vhd (2)	Sample VHDL instantiation of the entity in the megafunction wrapper file.
<output file>_inst.v (3)	Sample Verilog HDL instantiation of the module in the megafunction wrapper file.

Notes to Table 2–1:

- (1) The wizard only generates this file if you select AHDL output files.
- (2) The wizard only generates this file if you select VHDL output files.
- (3) The wizard only generates this file if you select Verilog HDL output files.
- (4) A megafunction wrapper file will be created by default for most megafunctions. If you select the **Generate a clear box body (for EDA tools only)** option, the wizard will create a clear box netlist file to be used with third-party EDA synthesis tools. For more information about how to use the MegaWizard Plug-In Manager, see Quartus II Help.

Instantiating Megafunctions Using the Port & Parameter Definition

You can instantiate the megafunction directly in your AHDL, Verilog HDL, or VHDL code by calling the function like any other subdesign, module, or component. However, you also must use a CMP file in VHDL designs, and an INC file in AHDL designs.



See *Quartus II Help* (or your IP documentation) for a list of the megafunction's ports and parameters. *Quartus II Help* also provides a sample VHDL CMP file.

Inferring Megafunctions from HDL Code

Synthesis tools, including Quartus II integrated synthesis, recognize certain types of HDL code and automatically infer the appropriate megafunction when a megafunction will provide optimal results. That is, the Quartus II software uses the Altera megafunction code when compiling your design—even though you did not specifically instantiate the megafunction. The Quartus II software infers megafunctions because they are optimized for Altera devices, so the area and/or performance may be better than generic HDL code. Additionally, you must use megafunctions to access certain Altera, architectural-specific features—such as memory, DSP blocks, and shift registers—that generally provide improved performance compared with basic logic elements.

This section describes the types of logic that standard synthesis tools recognize and map to megafunctions. The software infers only the specific functions listed in this section that are described by HDL code. The software cannot infer other functions, such as PLLs, LVDS drivers, transceivers, or DDIO circuitry from HDL code because these functions cannot be fully or efficiently described in HDL code. In some cases, synthesis tools have the option to disable inference.



Synthesis features specific to a synthesis tool are described in the *Synthesis* chapter of the *Introduction to Quartus II* manual.

Counters

To infer counter functions, synthesis tools look for a set of registers that feed through a plus-one adder, a minus-one adder, or both, and then convert the registers and logic to an `lpm_counter` megafunction. If a design also has logic implementing control signals, the synthesis tool can recognize them as well. For example, the Quartus II software recognizes the following signals:

- Asynchronous clear
- Asynchronous set (only to all logic value1s)
- Asynchronous load
- Count enable
- Synchronous clear
- Synchronous set (only to all logic value1s)
- Synchronous load
- Clock enable
- Up/down

The following code samples show simple Verilog HDL and VHDL counter function examples with different control signals.

Verilog HDL Counter with Count Enable & Asynchronous Clear

```

module counter
(
    clk,
    reset,
    result,
    ena)
;

    input clk;
    input reset;
    input ena;
    output [7:0] result;

    reg [7:0] result;

    always @(posedge clk or posedge reset)
    begin
        if (reset)
            result = 0;
        else if (ena)
            result = result + 1;
    end
endmodule

```

VHDL Counter with Synchronous Load

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY count IS
    PORT
    (
        clock: IN STD_LOGIC;
        sload: IN STD_LOGIC;
        data: IN integer RANGE 0 TO 31;
        result:OUT integer RANGE 0 TO 31
    );
END count;

ARCHITECTURE rtl OF count IS
    SIGNAL result_reg : integer RANGE 0 TO 31;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (sload = '1') THEN
                result_reg <= data;
            ELSE
                result_reg <= result_reg + 1;
            END IF
        END IF
    END PROCESS

```

```

        END IF;
    END IF;
END PROCESS;

    result <= result_reg;
END rtl;

```

Adder/Subtractors

To infer adder/subtractor functions, synthesis tools look for adders and subtractors that have the same set of inputs and outputs multiplexed by a common signal. The software may then merge the adders and subtractors and convert them to an `lpm_addsub` megafunction.

The following code samples show Verilog HDL and VHDL examples of simple adder/subtractors. The VHDL example includes a small user-defined package to configure the widths.

Verilog HDL Adder/Subtractor

```

module addsub (a, b, addnsub, result);

    input  [7:0] a;
    input  [7:0] b;
    input          addnsub;
    output [8:0] result;

    reg [8:0]result;

    always @(a or b or addnsub)
    begin
        if (addnsub)
            result = a + b;
        else
            result = a - b;
        end
    endmodule

```

VHDL Adder/Subtractor

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE my_package IS
    CONSTANT ADDER_WIDTH : integer := 5;
    CONSTANT RESULT_WIDTH : integer := 6;

    SUBTYPE ADDER_VALUE IS integer RANGE 0 TO 2 ** ADDER_WIDTH - 1;
    SUBTYPE RESULT_VALUE IS integer RANGE 0 TO 2 ** RESULT_WIDTH - 1;
END my_package;

LIBRARY ieee;

```

```

USE ieee.std_logic_1164.ALL;
USE work.my_package.ALL;

ENTITY addsub IS
  PORT
  (
    a:      IN ADDER_VALUE;
    b:      IN ADDER_VALUE;
    addnsb: IN STD_LOGIC;
    result: OUT RESULT_VALUE
  );
END addsub;

ARCHITECTURE rtl OF addsub IS
BEGIN
  PROCESS (a, b, addnsb)
  BEGIN
    IF (addnsb = '1') THEN
      result <= a + b;
    ELSE
      result <= a - b;
    END IF;
  END PROCESS;
END rtl;

```

Multipliers

To infer multiplier functions, synthesis tools look for multipliers and convert them to `lpm_mult` megafunctions. For devices with DSP blocks, the software may implement the `lpm_mult` function in a DSP block instead of logic elements (LEs), depending on device utilization. The Quartus II Fitter may also place input and output registers in DSP blocks (i.e., perform register packing) to improve performance and LE utilization.



For more information on the DSP block and which functions it can implement, see the appropriate Altera device family data sheet and the *DSP Solution Center* on the Altera website:

www.altera.com/literature/solutions/dsp/lit-dsp.jsp.

The following four code samples show Verilog HDL and VHDL examples for unsigned and signed multipliers that synthesis tools infer as an `lpm_mult` megafunction. Each example fits into one DSP block 9-bit element (using no LEs for registers when register packing occurs).



The signed declaration in Verilog HDL is a feature of the Verilog-2001 Standard.

Verilog HDL Unsigned Multiplier

```

module unsigned_mult (out, a, b);

    output [15:0] out;

```

```

input  [7:0] a;
input  [7:0] b;

assign out = a * b;

endmodule

```

Verilog HDL Signed Multiplier with Input & Output Registers (Pipelining = 2)

```

module signed_mult (out, clk, a, b);

output [15:0]out;
input  clk;
input signed[7:0] a;
input signed[7:0] b;

reg signed[7:0] a_reg;
reg signed[7:0] b_reg;
reg signed[15:0]out;

wire signed[15:0]mult_out;

assign mult_out = a_reg * b_reg;

always@(posedge clk)
begin
    a_reg <= a;
    b_reg <= b;
    out <= mult_out;
end

endmodule

```

VHDL Unsigned Multiplier with Input & Output Registers (Pipelining = 2)

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY unsigned_mult IS
    PORT (
        a:    IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        b:    IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        clk:  IN STD_LOGIC;
        aclr: IN STD_LOGIC;
        result: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
    );
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS

```

```

SIGNAL a_reg, b_reg: std_logic_vector (7 DOWNTO 0);

BEGIN
  PROCESS (clk, aclr)
  BEGIN
    IF (aclr ='1') THEN
      a_reg <= (OTHERS => '0');
      b_reg <= (OTHERS => '0');
      result <= (OTHERS => '0');

    ELSIF (clk'event AND clk = '1') THEN
      a_reg <= a;
      b_reg <= b;

      result <= unsigned(a_reg) * unsigned(b_reg);
    END IF;
  END PROCESS;

END rtl;

```

VHDL Signed Multiplier

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY signed_mult IS
  PORT (
    a:      IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    b:      IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    result: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
  );
END signed_mult;

ARCHITECTURE rtl OF signed_mult IS
  SIGNAL a_int, b_int: SIGNED (7 downto 0);
  SIGNAL pdt_int: SIGNED (15 downto 0);

BEGIN
  a_int <= SIGNED (a);
  b_int <= SIGNED (b);
  pdt_int <= a_int * b_int;
  result <= STD_LOGIC_VECTOR(pdt_int);
END rtl;

```

Multiplies-Accumulators & Multiplies-Adders

Synthesis tools detect multiply-accumulators or multiply-adders and convert them to `altmult_accum` or `altmult_add` megafunctions, respectively. The software then places these functions in DSP blocks.



The synthesis tools only infer multiply-accumulator and multiply-adder functions if the Altera device family has dedicated DSP blocks.

A multiply-accumulator consists of a multiply operator feeding an addition operator. The addition operator feeds a set of registers that then feed the second input to the addition operator. A multiply-adder consists of two- to four-multiply operators feeding one- or two-levels of addition, subtraction, or addition/subtraction operators. The second-level operator, if used, is always addition. In addition to the multiply-accumulator and multiply-adder, the Quartus II Fitter can also place input and output registers into the DSP block (i.e., perform register packing) to improve performance and LE utilization.

The following code samples show Verilog HDL and VHDL examples of inference for specific multiply-accumulators and multiply-adders.

Verilog HDL Unsigned Multiply-Accumulator with Input, Output & Pipeline Registers (Latency = 3)

```
module unsig_altmult_accum (dataout, dataa, datab, clk, aclr, clken);

input  [7:0]  dataa;
input  [7:0]  datab;
input          clk;
input          aclr;
input          clken;

output [31:0] dataout;

reg  [31:0] dataout;
reg  [7:0]  dataa_reg;
reg  [7:0]  datab_reg;
reg  [15:0] multareg;

wire [15:0] multareg;
wire [31:0] adder_out;

assign multareg = dataa_reg * datab_reg;
assign adder_out = multareg + dataout;

always @(posedge clk or posedge aclr)
begin
    if(aclr)
    begin
        dataa_reg <= 0;
        datab_reg <= 0;

        multareg <= 0;

        dataout <= 0;
    end
end
```

```

end

else if(clken)
begin
  dataa_reg <= dataa;
  datab_reg <= datab;

  multa_reg <= multa;

  dataout <= adder_out;
end
end

endmodule

```

Verilog HDL Signed Multiply-Adder (Latency = 0)

```

module sig_altmult_add (dataa, datab, datac, datad, result);

  input signed [15:0] dataa;
  input signed [15:0] datab;
  input signed [15:0] datac;
  input signed [15:0] datad;
  output [32:0] result;

  wire signed [31:0] mult0_result;
  wire signed [31:0] mult1_result;

  assign mult0_result = dataa * datab;
  assign mult1_result = datac * datad;

  assign result = (mult0_result + mult1_result);

endmodule

```

VHDL Unsigned Multiply-Adder with Input, Output & Pipeline Registers (Latency = 3)

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY unsignedmult_add IS
  PORT (
    a:    IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    b:    IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    c:    IN STD_LOGIC_VECTOR (7 DOWNTO 0);

```

```

        d:      IN STD_LOGIC_VECTOR (7 DOWNT0 0);
        clk :  IN STD_LOGIC;
        aclr  : IN STD_LOGIC;
        result: OUT STD_LOGIC_VECTOR (15 DOWNT0 0)
    );
END unsignedmult_add;

ARCHITECTURE rtl OF unsignedmult_add IS
    SIGNAL a_int, b_int, c_int, d_int : STD_LOGIC_VECTOR (7 DOWNT0 0);
    SIGNAL pdt_int, pdt2_int: unsigned (15 DOWNT0 0);
    SIGNAL result_int: unsigned (15 DOWNT0 0);
BEGIN
    PROCESS (clk, aclr)
    BEGIN
        IF (aclr = '1') THEN
            a_int <= (OTHERS => '0');
            b_int <= (OTHERS => '0');
            c_int <= (OTHERS => '0');
            d_int <= (OTHERS => '0');
            pdt_int <= (OTHERS => '0');
            pdt2_int <= (OTHERS => '0');
            result_int <= (OTHERS => '0');

            ELSIF (clk'event AND clk = '1') THEN
                a_int <= a;
                b_int <= b;
                c_int <= c;
                d_int <= d;

                pdt_int <= unsigned(a_int) * unsigned(b_int);
                pdt2_int <= unsigned(c_int) * unsigned(d_int);
                result_int <= pdt_int + pdt2_int;
            END IF;
        END PROCESS;

        result <= STD_LOGIC_VECTOR(result_int);
    END rtl;

```

VHDL Signed Multiply-Accumulator with Input, Output & Pipeline Registers (Latency = 3)

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY sig_altmult_accum IS
    PORT (
        a:      IN STD_LOGIC_VECTOR (7 DOWNT0 0);
        b:      IN STD_LOGIC_VECTOR (7 DOWNT0 0);

```

```

        clk:    IN STD_LOGIC;
        accum_out: OUT STD_LOGIC_VECTOR (15 DOWNT0 0)
    ) ;
END sig_altmult_accum;

ARCHITECTURE rtl OF sig_altmult_accum IS
    SIGNAL a_reg, b_reg : signed (7 DOWNT0 0);
    SIGNAL pdt_reg : signed (15 DOWNT0 0);
    SIGNAL adder_out : signed (15 DOWNT0 0);
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'event and clk = '1') THEN
            a_reg <= SIGNED (a);
            b_reg <= SIGNED (b);

            pdt_reg <= a_reg * b_reg;
            adder_out <= adder_out + pdt_reg ;
        END IF;
    END process;

    accum_out <= std_logic_vector(adder_out);
END rtl;

```

RAM

To infer RAM functions, synthesis tools detect sets of registers and logic that can be replaced with the `altsyncram` or `lpm_ram_dp` megafunctions, depending on the target device family.



The software only recognizes RAM blocks for device families that have dedicated RAM blocks.

Synthesis tools recognize single-port and simple, dual-port RAM blocks, but not true dual-port and quad-port RAM blocks. Because of language limitations, you cannot describe true dual-port RAM blocks in Verilog HDL or VHDL code. Additionally, the software may not infer very small RAM blocks because very small RAM blocks typically achieve the best performance when placed in LEs.



If the synthesis tool does not recognize and infer RAM block in your design, it may use a large amount of memory and could potentially cause runtime compilation problems.



For certain RAM configurations in certain device families, using a RAM megafunction may slightly change the design functionality if the RAM reads from and writes to the same location. In this scenario, the software generally issues a warning. If using Quartus II integrated synthesis, the Quartus II Help explains the condition under which the functionality changes.

The following code samples show Verilog HDL and VHDL examples that infer single- and dual-clock synchronous RAM. Depending on the device family's dedicated RAM architecture, the RAM may or may not need to be synchronous.



See the appropriate Altera device family data sheet for more information about your specific device at www.altera.com/literature/.

The following code samples show Verilog HDL and VHDL code examples of RAM with asynchronous read addresses and registered outputs.

The implementation of RAM example code in the following samples varies depending on the dedicated RAM architecture of the appropriate device family.

For example, implementing asynchronous read addresses in an APEX device's RAM block is straightforward because the APEX architecture supports asynchronous read addresses. However, read addresses in Stratix™ devices must be registered; therefore, you cannot directly implement the RAM example code in the following samples. Implement the RAM example code in the Stratix architecture by inferring an `altsyncram` megafunction, synthesis tools may move the output registers to the inputs of the RAM block. If the read and write clocks are not the same, moving the output registers to the inputs of the RAM block may slightly change the functionality. Thus, the software will issue a warning. When using Quartus II integrated synthesis, Quartus II Help explains the differences.

For the dual-clock examples—if you are reading and writing to the same address—the functionality of the inferred megafunction may differ from the original HDL code. (Synthesis tools issues a warning to inform you of this functional difference.)

Verilog HDL Single-Clock Synchronous RAM

```
module ram_infer(q, a, d, we, clk);
    output [7:0] q;
    input [7:0] d;
    input [6:0] a;
    input we, clk;
    reg [6:0] read_add;
    reg [7:0] mem [127:0];

    always @(posedge clk) begin
        if (we)
            mem[a] <= d;
            read_add <= a;
    end
end
```

```

        assign q = mem[read_addr];
    endmodule

```

Verilog HDL Dual-Clock Synchronous RAM

```

module ram_dual(q, addr_in, addr_out, d, we, clk1,
clk2);
    output[7:0] q;
    input [7:0] d;
    input [6:0] addr_in;
    input [6:0] addr_out;
    input we, clk1, clk2;

    reg [6:0] addr_out_reg;
    reg [7:0] q;
    reg [7:0] mem [127:0];

    always @(posedge clk1)
    begin
        if (we)
            mem[addr_in] <= d;
    end

    always @(posedge clk2) begin
        q <= mem[addr_out_reg];
        addr_out_reg <= addr_out;
    end
endmodule

```

VHDL Single-Clock Synchronous RAM

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ram IS
    PORT
    (
        clock:      IN STD_LOGIC;
        data:       IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN integer RANGE 0 to 31;
        read_address: IN integer RANGE 0 to 31;
        we:        IN std_logic;
        q:         OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END ram;

ARCHITECTURE rtl OF ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF std_logic_vector(2 DOWNTO 0);

    SIGNAL ram_block : MEM;
    SIGNAL read_address_reg : integer RANGE 0 to 31;

```

```

BEGIN
  PROCESS (clock)
  BEGIN
    IF (clock'event AND clock = '1') THEN
      IF (we = '1') THEN
        ram_block(write_address) <= data;
      END IF;

      read_address_reg <= read_address;

    END IF;
  END PROCESS;

  q <= ram_block(read_address_reg);
END rtl;

```

VHDL Dual-Clock Synchronous RAM

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ram_dual IS
  PORT
  (
    clock1, clock2: IN STD_LOGIC;
    data:           IN STD_LOGIC_VECTOR (3 DOWNTO 0);
    write_address: IN integer RANGE 0 to 31;
    read_address:  IN integer RANGE 0 to 31;
    we:            IN std_logic;
    q:             OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
  );
END ram_dual;

ARCHITECTURE rtl OF ram_dual IS
  TYPE MEM IS ARRAY(0 TO 31) OF std_logic_vector(3 DOWNTO 0);

  SIGNAL ram_block : MEM;
  SIGNAL read_address_reg : integer RANGE 0 to 31;

BEGIN
  PROCESS (clock1)
  BEGIN
    IF (clock1'event AND clock1 = '1') THEN
      IF (we = '1') THEN
        ram_block(write_address) <= data;
      END IF;

    END IF;
  END PROCESS;

  PROCESS (clock2)

```

```

BEGIN
  IF (clock2'event AND clock2 = '1') THEN
    q <= ram_block(read_address_reg);
    read_address_reg <= read_address;
  END IF;
END PROCESS;

END rtl;

```

Verilog HDL Single-Clock Synchronous RAM with Asynchronous Read Address

```

module ram (clock, data, write_address, read_address, we, q);
  parameter ADDRESS_WIDTH = 4;
  parameter DATA_WIDTH   = 8;

  input          clock;
  input[DATA_WIDTH-1:0]data;
  input[ADDRESS_WIDTH-1:0]write_address;
  input[ADDRESS_WIDTH-1:0]read_address;
  input          we;
  output[DATA_WIDTH-1:0]q;

  reg [DATA_WIDTH-1:0] q;
  reg [DATA_WIDTH-1:0] ram_block [2**ADDRESS_WIDTH-1:0];

  always @ (posedge clock)
  begin
    if (we)
      ram_block[write_address] <= data;

    q <= ram_block[read_address];
  end
endmodule

```

VHDL Single-Clock Synchronous RAM with Asynchronous Read Address

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ram IS
  GENERIC
  (
    ADDRESS_WIDTH: integer := 4;
    DATA_WIDTH: integer := 8
  );
  PORT
  (
    clock      : IN  std_logic;
    data       : IN  std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
    write_address: IN  std_logic_vector(ADDRESS_WIDTH - 1 DOWNTO 0);

```

```

        read_address: IN  std_logic_vector(ADDRESS_WIDTH - 1 DOWNT0 0);
        we           : IN  std_logic;
        q           : OUT std_logic_vector(DATA_WIDTH - 1 DOWNT0 0)
    );
END ram;

ARCHITECTURE rtl OF ram IS
    TYPE RAM IS ARRAY(0 TO 2 ** ADDRESS_WIDTH - 1) OF
std_logic_vector(DATA_WIDTH - 1 DOWNT0 0);

    SIGNAL ram_block : RAM;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(to_integer(unsigned(write_address))) <= data;
            END IF;

            q <= ram_block(to_integer(unsigned(read_address)));
        END IF;
    END PROCESS;
END rtl;

```

ROM

To infer ROM functions, synthesis tools detect sets of registers and logic that can be replaced with the `altsyncram` or `lpm_rom` megafunctions, depending on the target device family.



The software only recognizes ROM functions for device families that have dedicated RAM memory blocks.

ROMs are inferred when you have a case statement where a value is being set to a constant for every choice in the case statement. Because small ROMs typically achieve the best performance when placed in LEs, each ROM function has to meet a minimum size requirement to be inferred and placed into memory.

The following code samples show Verilog HDL and VHDL examples that infer synchronous ROM with registered address. Depending on the device family's dedicated RAM architecture, the ROM may or may not need to be synchronous; consult the device family data sheet for details. For device architectures with synchronous RAM blocks, such as Stratix devices, either the address or the output has to be registered for ROM code to be inferred. When output registers are used, the registers will be implemented using the input registers of the Stratix RAM block, but the functionality of the ROM will not change. If you register the address, such as in the following code samples, the power-up state of the inferred ROM may be different from the HDL design. In this scenario, the software

generally issues a warning. When using Quartus II integrated synthesis, Quartus II Help explains the condition under which the functionality changes.

Verilog HDL Synchronous ROM (with Registered Address)

```

module sync_rom (clock, address, data_out);
    input    clock;
    input [7:0]address;
    output [5:0]data_out;
    reg     [5:0]data_out;
    reg    [7:0]address_reg;

    always @ (posedge clock)
    begin
        address_reg = address;
    end

    always @ (address_reg)
    begin
        case (address_reg)
            8'b00000000: data_out = 6'b101111;
            8'b00000001: data_out = 6'b110110;
            ...
            8'b11111110: data_out = 6'b000001;
            8'b11111111: data_out = 6'b101010;
        endcase
    end
endmodule

```

VHDL Synchronous ROM (with Registered Address)

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY sync_rom IS
    PORT
    (
        clock: IN std_logic;
        address:IN STD_LOGIC_VECTOR(7 downto 0);
        data_out:OUT STD_LOGIC_VECTOR(5 downto 0)
    );
END sync_rom;

ARCHITECTURE rtl OF sync_rom IS
    SIGNAL address_reg : STD_LOGIC_VECTOR(7 downto 0);
BEGIN

    PROCESS (clock)
        BEGIN

```

```

        IF (clock'event and clock='1') THEN
            address_reg <= address;
        END IF;
    END PROCESS;

PROCESS (address_reg)
    BEGIN
        CASE address_reg IS
            WHEN "00000000" => data_out <= "101111";
            WHEN "00000001" => data_out <= "110110";
            ...
            WHEN "11111110" => data_out <= "000001";
            WHEN "11111111" => data_out <= "101010";
            WHEN OTHERS => data_out <= "101111";
        END CASE;
    END PROCESS;
END rtl;

```

The following code samples show Verilog HDL and VHDL examples of asynchronous ROM. The implementation of these ROM examples varies depending on the dedicated RAM architecture of the target device family. For example, in an APEX device this ROM is straightforward to implement because the APEX architecture supports asynchronous memories. In Stratix devices, this ROM is implemented in LEs.

Note that you can also use a DEFAULT choice in the case statement (not shown in any of the examples).

Verilog HDL Asynchronous ROM

```

module rom (address, data_out);
    input [7:0] address;
    output [5:0] data_out;
    reg [5:0] data_out;

    always @ (address)
    begin
        case (address)
            8'b00000000: data_out = 6'b101111;
            8'b00000001: data_out = 6'b110110;
            ...
            8'b11111110: data_out = 6'b000001;
            8'b11111111: data_out = 6'b101010;
        endcase
    end
endmodule

```

VHDL Asynchronous ROM

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY rom IS

```

```

PORT
(
    address:IN STD_LOGIC_VECTOR(7 downto 0);
    data_out:OUT STD_LOGIC_VECTOR(5 downto 0)
);
END rom;

ARCHITECTURE rtl OF rom IS
BEGIN
    PROCESS (address)
    BEGIN
        CASE address IS
            WHEN "00000000" => data_out <= "101111";
            WHEN "00000001" => data_out <= "110110";
            ...
            WHEN "11111110" => data_out <= "000001";
            WHEN "11111111" => data_out <= "101010";
            WHEN OTHERS => data_out <= "101111";
        END CASE;
    END PROCESS;
END rtl;

```

Shift Registers

To infer shift registers, synthesis tools detect a group of shift registers of the same length and convert them to an `altshift_taps` megafunction. To be detected, all of the shift registers must:

- Use the same clock and clock enable
- Not have any other secondary signals
- Have equally spaced taps that are at least three registers apart.

The software only recognizes shift registers for device families that have dedicated RAM blocks and uses certain guidelines to determine the best implementation, for example:

- For FLEX® 10K and ACEX® 1K devices, the software does not infer `altshift_taps` megafunctions because FLEX 10K and ACEX 1K devices have a relatively small amount of dedicated memory.
- For APEX 20K and APEX II devices, the software infers `altshift_taps` megafunctions if the shift register has more than a total of 128 bits. Smaller shift registers typically do not benefit from implementation in dedicated memory.
- For Stratix, Stratix II, and Cyclone™ devices, the software determines whether to infer `altshift_taps` megafunctions based on the width of the registered bus (\bar{w}), the length between each tap (L), and the number of taps (N).
 - If the registered bus width is one ($\bar{w} = 1$), the software infers `altshift_taps` if the number of taps times the length between each tap is greater than or equal to 64 ($N \times L \geq 64$).
 - If the registered bus width is greater than one ($\bar{w} > 1$), the software infers `altshift_taps` if the registered bus width

times the number of taps times the length between each tap is greater than or equal to 32 ($W \times N \times L \geq 32$).



If the length between each tap (L) is not a power of two, the software uses more LEs to decode the read and write counters. This situation occurs because for different sizes of shift registers, external decode logic (using LEs) is required to implement the function, which eliminates the advantage of implementing shift registers in memory.



The registers that the software maps to the `altshift_taps` megafunction and places in RAM are not available in simulation tools because their node names do not exist after synthesis.

The following code sample shows a Verilog HDL example of a simple, single-bit wide, 64-bit long shift register. The software implements the register ($W = 1$ and $M = 64$) in an `altshift_taps` megafunction for supported devices. If the length of the register is less than 64 bits, the software implements the shift register in LEs.

Verilog HDL Single-Bit Wide, 64-Bit Long Shift Register

```
module shift_1x64 (clk,
                 shift,
                 sr_in,
                 sr_out
                );

    input clk, shift;
    input sr_in;
    output sr_out;

    reg [63:0] sr;

    always@(posedge clk)
    begin
        if (shift == 1'b1)
        begin
            sr[63:1] <= sr[62:0];
            sr[0] <= sr_in;
        end
    end

    assign sr_out = sr[63];

endmodule
```

The following code sample shows a Verilog HDL example of an 8-bit wide, 64-bit long shift register ($W > 1$ and $M = 64$) with evenly spaced taps at 15, 31, and 47. The software implements this function in a single `altshift_taps` megafunction and maps it to RAM in supported devices.

Verilog HDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```

module shift_8x64_taps (clk,
    shift,
    sr_in,
    sr_out,
    sr_tap_one,
    sr_tap_two,
    sr_tap_three
);

    input clk, shift;

    input [7:0] sr_in;
    output [7:0] sr_tap_one, sr_tap_two, sr_tap_three,
sr_out;

    reg [7:0] sr [63:0];
    integer n;

    always@(posedge clk)
    begin
        if (shift == 1'b1)
            begin
                for (n = 63; n>0; n = n-1)
                    begin
                        sr[n] <= sr[n-1];
                    end

                sr[0] <= sr_in;
            end
        end

    assign sr_tap_one = sr[15];
    assign sr_tap_two = sr[31];
    assign sr_tap_three = sr[47];
    assign sr_out = sr[63];

endmodule

```

The following code sample shows a VHDL example of a 8-bit wide, 64-bit long shift register with evenly spaced taps at 15, 31, and 47.

VHDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY shift_8x64_taps IS
PORT (
    clk :      IN STD_LOGIC;
    shift :   IN STD_LOGIC;
    sr_in  :   IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    sr_tap_one  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    sr_tap_two  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    sr_tap_three : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    sr_out  :   OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
);
END shift_8x64_taps;

ARCHITECTURE arch OF shift_8x64_taps IS

SUBTYPE sr_width IS STD_LOGIC_VECTOR(7 DOWNTO 0);
TYPE sr_length IS ARRAY (63 DOWNTO 0) OF sr_width;

SIGNAL sr : sr_length;

BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT and clk = '1') THEN
            IF (shift = '1') THEN
                sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
                sr(0) <= sr_in;
            END IF;
        END IF;
    END PROCESS;

    sr_tap_one <= sr(15);
    sr_tap_two <= sr(31);
    sr_tap_three <= sr(47);
    sr_out <= sr(63);

END arch;

```

Device-Specific Coding Recommendations

This section provides device-specific coding recommendations for Altera device architectures. Designing specific logic structures to match the appropriate Altera device architecture can provide significant performance improvements.

Tri-State Signals

When targeting Altera devices, you should only use tri-state signals when they are attached to top-level bidirectional or output pins. Avoid lower-level bidirectional pins, and avoid using the Z logic value unless it is driving an output or bidirectional pin.

The following code samples are simple examples for creating tri-state bidirectional signals.

Tri-State Signal Example in Verilog HDL

```
module tristate(myinput, myenable, mybidir);
    input myinput, myenable;
    inout mybidir;

    assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

Tri-State Signal Example in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity tristate is
port
    (
        mybidir: inoutstd_logic;
        myinput: instd_logic;
        myenable: instd_logic
    );
end tristate;

architecture rtl of tristate is
begin

    mybidir <= 'Z' when (myenable = '0') else myinput;

end rtl;
```

Adder Trees

Structuring adder trees appropriately to match your targeted Altera device architecture can result in significant performance and density improvements. A good example of an application that uses a large adder tree is a finite impulse response (FIR) correlator; using a binary or ternary adder tree appropriately can greatly improve your quality of results.

This section explains why coding recommendations are different for Altera four-input lookup table (LUT) devices (e.g., Stratix, APEX 20K, and FLEX 10K devices) and the Altera six-input LUT logic structures available in Stratix II devices.

Architectures With Four-Input LUTs in Logic Elements (LEs)

Architectures such as Stratix, APEX 20K, and FLEX 10K devices contain four-input LUTs as the standard combinational structure in the logic element (LE).

The fastest way to add three numbers A , B , and C , in Stratix, APEX 20K, or FLEX 10K devices is to add $A + B$, register the output, and then add the registered output to C . Adding $A + B$ takes one level of logic (i.e., 1 bit is added in one LE), so this runs at full clock speed. This can be extended to as many numbers as desired.

However, in the example that follows, five numbers A , B , C , D , and E are added. Adding five numbers in Stratix, APEX 20K, or FLEX 10K devices requires four adders and three levels of registers for a total of 64 LEs (for 16-bit numbers).

Verilog HDL Binary Tree Example

```
module binary_adder_tree(A, B, C, D, E, CLK, OUT);
    parameter WIDTH = 16;

    input [WIDTH-1:0] A, B, C, D, E;
    input             CLK;
    output [WIDTH-1:0] OUT;

    wire [WIDTH-1:0]    sum1, sum2, sum3, sum4;

    reg [WIDTH-1:0]    sumreg1, sumreg2, sumreg3,
sumreg4;

    // Registers
    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
            sumreg2 <= sum2;
            sumreg3 <= sum3;
            sumreg4 <= sum4;
        end

    // 2-bit additions
    assign    sum1 = A + B;
    assign    sum2 = C + D;
    assign    sum3 = sumreg1 + sumreg2;
    assign    sum4 = sumreg3 + E;
```

```

        assign      OUT = sumreg4;

    endmodule

```

Architectures With Six-Input LUTs in Adaptive Logic Modules (ALMs)

Because the Stratix II architecture uses a six-input LUT in its basic logic structure, adaptive logic module (ALM), Stratix II devices benefit from a more streamlined coding style. Specifically, Stratix II device ALMs can simultaneously add three bits. Thus, the tree in the previous example need only be two levels deep and contain just two add-by-three inputs instead of four add-by-two inputs.

Again, although the code in the previous example successfully compiles in Stratix II devices, it is not efficient and does not take advantage of the six-input LUT ALMs. By restructuring the tree as a ternary tree the design becomes much more efficient, significantly improving performance and density utilization. Thus, when targeting Stratix II devices, large binary adder trees designed for four-input LUT architectures should be rewritten to take advantage of the Stratix II device architecture.

The following example uses just 32 half-ALMs in a Stratix II device—more than a four-to-one advantage over the prior example implemented in a Stratix device.



You cannot pack a Stratix II LAB full when using this type of coding style, because the device will run out of LAB inputs. While integrated Quartus II synthesis reports that 32 half-ALMs are used to implement the function, the Quartus II Fitter may report a slightly higher number.

Verilog HDL Ternary Tree Example

```

module ternary_adder_tree(A, B, C, D, E, CLK, OUT);
    parameter WIDTH = 16;

    input [WIDTH-1:0] A, B, C, D, E;
    input      CLK;
    output [WIDTH-1:0] OUT;

    wire [WIDTH-1:0]    sum1, sum2;

    reg [WIDTH-1:0]    sumreg1, sumreg2;

    // Registers
    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
            sumreg2 <= sum2;
        end
end

```

```
// 3-bit additions
assign      sum1 = A + B + C;
assign      sum2 = sumreg1 + D + E;

assign      OUT = sumreg2;

endmodule
```

General Coding Recommendations

This section provides general coding recommendations, specifically regarding latches and state machines.

Latches

When designing combinational logic, certain coding styles can create an unintentional latch. For example, when CASE or IF statements do not cover all possible input conditions, latches may be required to hold the output if a new output value is not assigned. Check your synthesis tool messages for references to latches being inferred.

The `full_case` attribute can be used in Verilog HDL designs to indicate that non-specified cases can be treated as **don't care**. However, using the `full_case` attribute may lead to simulation mismatches because it is a synthesis-only attribute.



For more information on latches, see the *Synthesis* chapter of the *Introduction to Quartus II* manual.

Omitting the final ELSE or WHEN OTHERS clause in an IF or CASE statement can also generate a latch. “Don't care” assignments on the default conditions tend to prevent latch generation. Synthesis software generally treats unknowns as **don't care** conditions to optimize logic. For the best logic optimization, assign the default CASE or final ELSE value to **don't care** instead of a logic value.

The following shows example VHDL code that prevents an unintentional latch. Without the final ELSE clause, the code creates unintentional latches to cover the remaining combinations of the `sel` inputs. When targeting a Stratix device with the following code, omitting the final ELSE condition causes the compiler to use six LEs instead of the three it uses with the ELSE statement. Also, assigning the final ELSE value to 1 instead of X results in four LEs.

Sample VHDL Code Preventing Unintentional Latch Creation

```
LIBRARY ieee;
USE IEEE.std_logic_1164.all;
```

```

ENTITY nolatch IS
    PORT (a,b,c : IN STD_LOGIC;
          sel: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
          oput: OUT STD_LOGIC);
END nolatch;

ARCHITECTURE rtl OF nolatch IS
BEGIN
    PROCESS (a,b,c,sel) BEGIN
        IF sel = "00000" THEN
            oput <= a;
        ELSIF sel = "00001" THEN
            oput <= b;
        ELSIF sel = "00010" THEN
            oput <= c;
        ELSE
            --- Prevents latch inference
            oput <= 'X';--/
        END IF;
    END PROCESS;
END rtl;

```

State Machines

To ensure proper recognition and inference of state machines, as well as to improve performance, Altera recommends the following (applicable for both Verilog HDL and VHDL):

- Assign default values to outputs derived from the state machine to avoid generation of unwanted latches during synthesis.
- Assign a default clause to direct the state machine in case it accidentally reaches an unused state.
- Separate the state machine logic from all arithmetic functions and data paths, including assigning output values.
- If your design contains an operation that is used by more than one state, define the operation outside the state machine and make the output logic of the state machine use this value.
- Use a simple asynchronous or synchronous reset to ensure a defined power-up state.



See your synthesis tool's documentation for more tool-specific coding recommendations. Also see the *Synthesis* chapter of the *Introduction to Quartus II* manual for Quartus II-specific, state machine guidelines, as well as recommended general coding practices and sample VHDL and Verilog HDL code for creating state machines.

Conclusion

Keep the targeted device architecture in mind when selecting your coding style, as certain coding styles can dramatically improve performance results. To improve your design's performance and area utilization, take advantage of advanced device features such as memory and DSP blocks, as well as the specific architecture of the targeted Altera device.



For additional optimization recommendations, see the *Optimizing the Fit* section in the *Place & Route* chapter of the *Introduction to Quartus II* manual.