13. Recommended HDL Coding Styles

QII51007-13.1.0

This chapter provides Hardware Description Language (HDL) coding style recommendations to ensure optimal synthesis results when targeting Altera® devices.

HDL coding styles can have a significant effect on the quality of results that you achieve for programmable logic designs. Synthesis tools optimize HDL code for both logic utilization and performance; however, synthesis tools have no information about the purpose or intent of the design. The best optimizations often require conscious interaction by you, the designer.

This chapter includes the following sections:

For additional guidelines about structuring your design, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*. For additional handcrafted techniques you can use to optimize design blocks for the adaptive logic modules (ALMs) in many Altera devices, including a collection of circuit building blocks and related discussions, refer to the *Advanced Synthesis Cookbook*.

The Altera website also provides design examples for other types of functions and to target specific applications. For more information about design examples, refer to the Design Examples page and the Reference Designs page on the Altera website.

For style recommendations, options, or HDL attributes specific to your synthesis tool (including Quartus® II integrated synthesis and other EDA tools), refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Using the Quartus II Templates

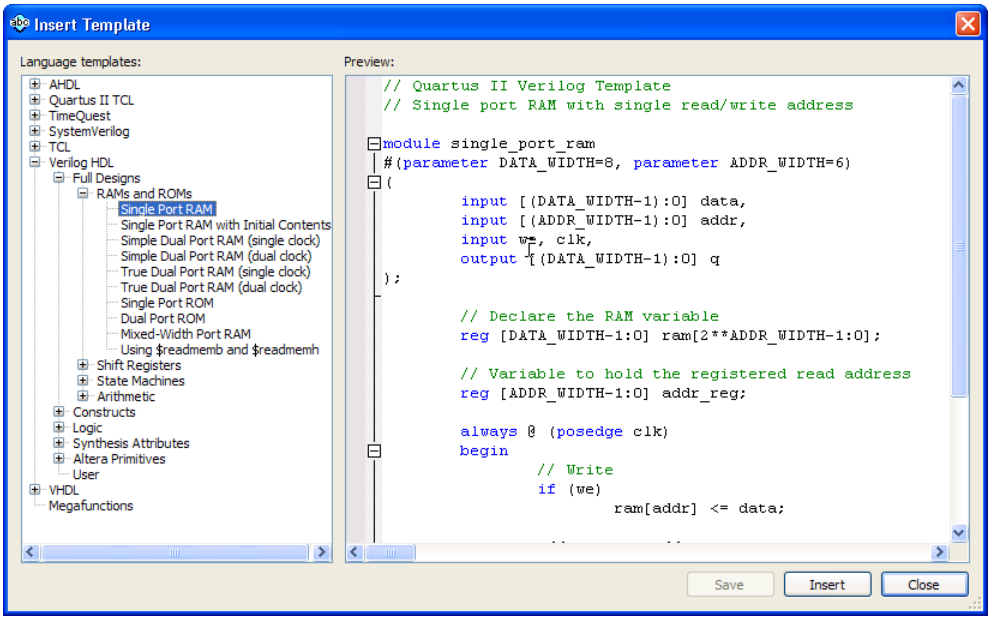
Many of the Verilog HDL and VHDL examples in this chapter correspond with examples in the **Full Designs** section of the **Quartus II Templates**. You can easily insert examples into your HDL source code with the Quartus II Text Editor, in the Quartus II software, or with your preferred text editor.

Inserting a Template with the Quartus II Text Editor

To use a template with the Quartus II software, follow these steps:

1. On the **File** menu, click **New**.
2. In the **New** dialog box, select the type of design file corresponding to the type of HDL you want to use, SystemVerilog HDL File, VHDL File, or Verilog HDL File.
3. Click the **Insert Template** button on the text editor menu, or, right-click in the blank Verilog or VHDL file, then click **Insert Template**.
4. In the **Insert Template** dialog box shown in Figure 13–1, expand the section corresponding to the appropriate HDL, then expand the **Full Designs** section.
5. Expand the design category, for example **RAMs and ROMs**.
6. Select a design. The HDL appears in the **Preview** pane.
7. Click **Insert** to paste the HDL design to the blank Verilog or VHDL file you created in step 2.
8. Click **Close** to close the **Insert Template** dialog box.

Figure 13–1. Insert Template Dialog Box



You can use any of the standard features of the Quartus II Text Editor to modify the HDL design or save the template as an HDL file to edit in your preferred text editor. For more information about inserting a template with the Quartus II Text Editor, refer to *About the Quartus II Text Editor* in Quartus II Help.

Using 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 megafunctions may offer more efficient logic synthesis and device implementation. You can scale the megafunction's size and specify various options by setting parameters. Megafunctions include the library of parameterized modules (LPM) and Altera device-specific megafunctions.

To use megafunctions in your HDL code, you can instantiate them as described in "Instantiating Altera Megafunctions in HDL Code" on page 13–3.

Sometimes it is preferable to make your code independent of device family or vendor. In this case, you might not want to instantiate megafunctions directly. For some types of logic functions, such as memories and DSP functions, you can infer device-specific dedicated architecture blocks instead of instantiating a megafunction. Synthesis tools, including Quartus II integrated synthesis, recognize certain types of HDL code and automatically infer the appropriate megafunction or map directly to device atoms. Synthesis tools infer megafunctions to take advantage of logic that is optimized for Altera devices or to target dedicated architectural blocks.

In cases where you prefer to use generic HDL code instead of instantiating a specific function, follow the guidelines and coding examples in "Inferring Multiplier and DSP Functions from HDL Code" on page 13–5 and "Inferring Memory Functions from HDL Code" on page 13–13 to ensure your HDL code infers the appropriate function.

You can infer or instantiate megafunctions to target some Altera device-specific architecture features such as memory and DSP blocks. You must instantiate megafunctions to target certain other device and high-speed features, such as LVDS drivers, phase-locked loops (PLLs), transceivers, and double-data rate input/output (DDIO) circuitry.

Instantiating Altera Megafunctions in HDL Code

The following sections describe how to use megafunctions by instantiating them in your HDL code with the following methods:

- "Instantiating Megafunctions Using the MegaWizard Plug-In Manager"—You can use the MegaWizard™ Plug-In Manager to parameterize the function and create a wrapper file.
- "Creating a Netlist File for Other Synthesis Tools"—You can optionally create a netlist file instead of a wrapper file.
- "Instantiating Megafunctions Using the Port and Parameter Definition"—You can instantiate the function directly in your HDL code.

Instantiating Megafunctions Using the MegaWizard Plug-In Manager

Use the MegaWizard Plug-In Manager as described in this section to create megafunctions in the Quartus II software that you can instantiate in your HDL code. The MegaWizard Plug-In Manager provides a GUI to customize and parameterize megafunctions, and ensures that you set all megafunction parameters properly. When you finish setting parameters, you can specify which files you want generated. Depending on which language you choose, the MegaWizard Plug-In Manager instantiates the megafunction with the correct parameters and generates a megafunction variation file (wrapper file) in Verilog HDL (**.v**), VHDL (**.vhd**), or AHDL (**.tdf**), along with other supporting files.

The MegaWizard Plug-In Manager provides options to create the files listed in Table 13–1.

Table 13–1. MegaWizard Plug-In Manager Generated Files

File	Description
<*output file*>**.v|.vhd|.tdf**	Verilog HDL Variation Wrapper File—Megafunction wrapper file for instantiation in a Verilog HDL, VHDL, or AHDL design respectively. The MegaWizard Plug-In Manager generates a **.v**, **.vhd**, or **.tdf** file, depending on the language you select for the output file on the megafunction selection page of the wizard.
<*output file*>**.inc**	ADHL Include File—Used in AHDL Text Design Files (**.tdf**).
<*output file*>**.cmp**	Component Declaration File—Used in VHDL design files.
<*output file*>**.bsf**	Block Symbol File—Used in Quartus II schematic Block Design Files (**.bdf**).
<*output file*>**_inst.v|.vhd|.tdf**	HDL Instantiation Template for the language of the variation file—Sample instantiation of the Verilog HDL module, VHDL entity, or AHDL subdesign.
<*output file*>**_bb.v**	Black box Verilog HDL Module Declaration—Hollow-body module declaration that can be used in Verilog HDL designs to specify port directions when instantiating the megafunction as a black box in third-party synthesis tools.
<*output file*>**_syn.v**	Synthesis timing and resource estimation netlist—Additional synthesis netlist file created if you enable the option to generate a synthesis timing and resource estimation netlist. For more information, refer to "Creating a Netlist File for Other Synthesis Tools" for details.

Creating a Netlist File for Other Synthesis Tools

When you use certain megafunctions with other EDA synthesis tools (that is, tools other than Quartus II integrated synthesis), you can optionally create a netlist for timing and resource estimation instead of a wrapper file.

The netlist file is a representation of the customized logic used in the Quartus II software. The file provides the connectivity of architectural elements in the megafunction but may not represent true functionality. This information enables certain other EDA synthesis tools to better report timing and resource estimates. In addition, synthesis tools can use the timing information to focus timing-driven optimizations and improve the quality of results.

To generate the netlist, turn on **Generate netlist** under **Timing and resource estimation** on the **EDA** page of the MegaWizard Plug-In Manager. The netlist file is called <*output file*>**_syn.v**. If you use this netlist for synthesis, you must include the megafunction wrapper file, either <*output file*>**.v** or <*output file*>**.vhd,** for placement and routing in the project created with the Quartus II software.

Because your synthesis tool may call the Quartus II software in the background to generate this netlist, turning on the **Generate Netlist** option might be optional.

For information about support for timing and resource estimation netlists in your synthesis tool, refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Instantiating Megafunctions Using the Port and Parameter Definition

You can instantiate the megafunction directly in your Verilog HDL, VHDL, or AHDL code by calling the megafunction and setting its parameters as you would any other module, component, or subdesign.

For a list of the megafunction ports and parameters, refer to the specific megafunction in the Quartus II Help. You can also refer to the IP and Megafunction page on the Altera website.

Altera recommends that you use the MegaWizard Plug-In Manager for complex megafunctions such as PLLs, transceivers, and LVDS drivers. For details about using the MegaWizard Plug-In Manager, refer to "Instantiating Megafunctions Using the MegaWizard Plug-In Manager" on page 13–4.

Inferring Multiplier and DSP Functions from HDL Code

The following sections describe how to infer multiplier and DSP functions from generic HDL code, and, if applicable, how to target the dedicated DSP block architecture in Altera devices:

- "Inferring Multipliers from HDL Code"
- "Inferring Multiply-Accumulators and Multiply-Adders from HDL Code" on page 13–8

For synthesis tool features and options, refer to your synthesis tool documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook.*

For more design examples involving advanced multiply functions and complex DSP functions, refer to the DSP Design Examples page on the Altera website.

Inferring Multipliers from HDL Code

To infer multiplier functions, synthesis tools look for multipliers and convert them to LPM_MULT or ALTMULT_ADD megafunctions, or may map them directly to device atoms. For devices with DSP blocks, the software can implement the function in a DSP block instead of logic, depending on device utilization. The Quartus II Fitter can also place input and output registers in DSP blocks (that is, perform register packing) to improve performance and area utilization.

For more information about the DSP block and supported functions, refer to the appropriate Altera device family handbook and the Altera DSP Solutions Center website.

Example 13–1 and Example 13–2 show Verilog HDL code examples, and Example 13–3 and Example 13–4 show VHDL code examples, for unsigned and signed multipliers that synthesis tools can infer as a megafunction or DSP block atoms. Each example fits into one DSP block element. In addition, when register packing occurs, no extra logic cells for registers are required.

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

Example 13–1. 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
```

Example 13–2. Verilog HDL Signed Multiplier with Input and 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
```

Example 13–3. VHDL Unsigned Multiplier with Input and Output Registers (Pipelining = 2)

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

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

ARCHITECTURE rtl OF unsigned_mult IS
   SIGNAL a_reg, b_reg: UNSIGNED (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 <= a_reg * b_reg;
      END IF;
   END PROCESS;
END rtl;
```

Example 13–4. VHDL Signed Multiplier

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

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

ARCHITECTURE rtl OF signed_mult IS
BEGIN
   result <= a * b;
END rtl;
```

Inferring Multiply-Accumulators and Multiply-Adders from HDL Code

Synthesis tools detect multiply-accumulators or multiply-adders and convert them to ALTMULT_ACCUM or ALTMULT_ADD megafunctions, respectively, or may map them directly to device atoms. The Quartus II software then places these functions in DSP blocks during placement and routing.

Synthesis tools infer multiply-accumulator and multiply-adder functions only if the Altera device family has dedicated DSP blocks that support these functions.

A simple multiply-accumulator consists of a multiplier feeding an addition operator. The addition operator feeds a set of registers that then feeds the second input to the addition operator. A simple multiply-adder consists of two to four multipliers feeding one or two levels of addition, subtraction, or addition/subtraction operators. Addition is always the second-level operator, if it is used. In addition to the multiply-accumulator and multiply-adder, the Quartus II Fitter also places input and output registers into the DSP blocks to pack registers and improve performance and area utilization.

Some device families offer additional advanced multiply-add and accumulate functions, such as complex multiplication, input shift register, or larger multiplications.

For details about advanced DSP block features, refer to the appropriate device handbook. For more design examples of DSP functions and inferring advanced features in the multiply-add and multiply-accumulate circuitry, refer to the DSP Design Examples page and *AN639: Inferring Stratix V DSP Blocks for FIR Filtering Applications* on Altera's website.

The Verilog HDL and VHDL code samples in Example 13–5 through Example 13–8 on pages 13–9 through 13–12 infer multiply-accumulators and multiply-adders with input, output, and pipeline registers, as well as an optional asynchronous clear signal. Using the three sets of registers provides the best performance through the function, with a latency of three. You can remove the registers in your design to reduce the latency.

Example 13–5. Verilog HDL Unsigned Multiply-Accumulator

```
module unsig_altmult_accum (dataout, dataa, datab, clk, aclr, clken);
   input [7:0] dataa, datab;
   input clk, aclr, clken;
   output reg[16:0] dataout;

   reg [7:0] dataa_reg, datab_reg;
   reg [15:0] multa_reg;
   wire [15:0] multa;
   wire [16:0] adder_out;
   assign multa = dataa_reg * datab_reg;
   assign adder_out = multa_reg + dataout;

   always @ (posedge clk or posedge aclr)
   begin
      if (aclr)
      begin
         dataa_reg <= 8'b0;
         datab_reg <= 8'b0;
         multa_reg <= 16'b0;
         dataout <= 17'b0;
      end
      else if (clken)
      begin
         dataa_reg <= dataa;
         datab_reg <= datab;
         multa_reg <= multa;
         dataout <= adder_out;
      end
   end
endmodule
```

Example 13–6. Verilog HDL Signed Multiply-Adder

```
module sig_altmult_add (dataa, datab, datac, datad, clock, aclr,
result);
   input signed [15:0] dataa, datab, datac, datad;
   input clock, aclr;
   output reg signed [32:0] result;

   reg signed [15:0] dataa_reg, datab_reg, datac_reg, datad_reg;
   reg signed [31:0] mult0_result, mult1_result;

   always @ (posedge clock or posedge aclr) begin
       if (aclr) begin
           dataa_reg <= 16'b0;
           datab_reg <= 16'b0;
           datac_reg <= 16'b0;
           datad_reg <= 16'b0;
           mult0_result <= 32'b0;
           mult1_result <= 32'b0;
           result <= 33'b0;
        end
        else begin
           dataa_reg <= dataa;
           datab_reg <= datab;
           datac_reg <= datac;
           datad_reg <= datad;
           mult0_result <= dataa_reg * datab_reg;
           mult1_result <= datac_reg * datad_reg;
           result <= mult0_result + mult1_result;
       end
   end
endmodule
```

Example 13–7. VHDL Signed Multiply-Accumulator

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

ENTITY sig_altmult_accum IS
   PORT (
      a: IN SIGNED(7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      accum_out: OUT SIGNED (15 DOWNTO 0)
   ) ;
END sig_altmult_accum;

ARCHITECTURE rtl OF sig_altmult_accum IS
   SIGNAL a_reg, b_reg: SIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg: SIGNED (15 DOWNTO 0);
   SIGNAL adder_out: SIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr = '1') then
          a_reg <= (others => '0');
          b_reg <= (others => '0');
          pdt_reg <= (others => '0');
          adder_out <= (others => '0');
      ELSIF (clk'event and clk = '1') THEN
          a_reg <= (a);
          b_reg <= (b);
          pdt_reg <= a_reg * b_reg;
          adder_out <= adder_out + pdt_reg;
      END IF;
   END process;
   accum_out <= adder_out;
END rtl;
```

Example 13–8. VHDL Unsigned Multiply-Adder

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

ENTITY unsignedmult_add IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      c: IN UNSIGNED (7 DOWNTO 0);
      d: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsignedmult_add;

ARCHITECTURE rtl OF unsignedmult_add IS
   SIGNAL a_reg, b_reg, c_reg, d_reg: UNSIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg, pdt2_reg: UNSIGNED (15 DOWNTO 0);
   SIGNAL result_reg: UNSIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr = '1') THEN
         a_reg <= (OTHERS => '0');
         b_reg <= (OTHERS => '0');
         c_reg <= (OTHERS => '0');
         d_reg <= (OTHERS => '0');
         pdt_reg <= (OTHERS => '0');
         pdt2_reg <= (OTHERS => '0');

      ELSIF (clk'event AND clk = '1') THEN
         a_reg <= a;
         b_reg <= b;
         c_reg <= c;
         d_reg <= d;
         pdt_reg <= a_reg * b_reg;
         pdt2_reg <= c_reg * d_reg;
         result_reg <= pdt_reg + pdt2_reg;
      END IF;
      END PROCESS;
   result <= result_reg;
END rtl;
```

Inferring Memory Functions from HDL Code

The following sections describe how to infer memory functions from generic HDL code and, if applicable, to target the dedicated memory architecture in Altera devices:

- "Inferring RAM functions from HDL Code" on page 13–14
- "Inferring ROM Functions from HDL Code" on page 13–36
- "Shift Registers—Inferring the ALTSHIFT_TAPS Megafunction from HDL Code" on page 13–40

For synthesis tool features and options, refer to your synthesis tool documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Altera's dedicated memory architecture offers a number of advanced features that can be easily targeted using the MegaWizard Plug-In Manager, as described in "Instantiating Altera Megafunctions in HDL Code" on page 13–3. The coding recommendations in the following sections provide portable examples of generic HDL code that infer the appropriate megafunction. However, if you want to use some of the advanced memory features in Altera devices, consider using the megafunction directly so that you can control the ports and parameters easily.

You can also use the Quartus II templates provided in the Quartus II software as a starting point. For more information, refer to "Inserting a Template with the Quartus II Text Editor" on page 13–2. Table 13–2 lists the full designs for RAMs and ROMs available in the Quartus II templates.

Most of these designs can also be found on the Design Examples page on the Altera website.

Table 13–2. RAM and ROM Full Designs from the Quartus II Templates (Part 1 of 2)

Language	Full Design Name
VHDL	Single-Port RAM Single-Port RAM with Initial Contents Simple Dual-Port RAM (single clock) Simple Dual-Port RAM (dual clock) True Dual-Port RAM (single clock) True Dual-Port RAM (dual clock) Mixed-Width RAM Mixed-Width True Dual-Port RAM Byte-Enabled Simple Dual-Port RAM Byte-Enabled True Dual-Port RAM Single-Port ROM Dual-Port ROM

Table 13–2. RAM and ROM Full Designs from the Quartus II Templates (Part 2 of 2)

Language	Full Design Name
Verilog HDL	Single-Port RAM Single-Port RAM with Initial Contents Simple Dual-Port RAM (single clock) Simple Dual-Port RAM (dual clock) True Dual-Port RAM (single clock) True Dual-Port RAM (dual clock) Single-Port ROM Dual-Port ROM
System Verilog	Mixed-Width Port RAM Mixed-Width True Dual-Port RAM Mixed-Width True Dual-Port RAM (new data on same port read during write) Byte-Enabled Simple Dual Port RAM Byte-Enabled True Dual-Port RAM

Inferring RAM functions from HDL Code

To infer RAM functions, synthesis tools detect sets of registers and logic that can be replaced with the ALTSYNCRAM or ALTDPRAM megafunctions for device families that have dedicated RAM blocks, or may map them directly to device memory atoms. Tools typically consider all signals and variables that have a multi-dimensional array type and then create a RAM block, if applicable, based on the way the signals, variables, or both are assigned, referenced, or both in the HDL source description.

Standard synthesis tools recognize single-port and simple dual-port (one read port and one write port) RAM blocks. Some tools (such as the Quartus II software) also recognize true dual-port (two read ports and two write ports) RAM blocks that map to the memory blocks in certain Altera devices.

Some tools (such as the Quartus II software) also infer memory blocks for array variables and signals that are referenced (read/written) by two indices, to recognize mixed-width and byte-enabled RAMs for certain coding styles.

If your design contains a RAM block that your synthesis tool does not recognize and infer, the design might require a large amount of system memory that can potentially cause compilation problems.

When you use a formal verification flow, Altera recommends that you create RAM blocks in separate entities or modules that contain only the RAM logic. In certain formal verification flows, for example, when using Quartus II integrated synthesis, the entity or module containing the inferred RAM is put into a black box automatically because formal verification tools do not support RAM blocks. The Quartus II software issues a warning message when this situation occurs. If the entity or module contains any additional logic outside the RAM block, this logic cannot be verified because it also must be treated as a black box for formal verification.

The following sections present several guidelines for inferring RAM functions that match the dedicated memory architecture in Altera devices, and then provide recommended HDL code for different types of memory logic.

Use Synchronous Memory Blocks

Altera recommends using synchronous memory blocks for Altera designs. Because memory blocks in the newest devices from Altera are synchronous, RAM designs that are targeted towards architectures that contain these dedicated memory blocks must be synchronous to be mapped directly into the device architecture. For these devices, asynchronous memory logic is implemented in regular logic cells.

Synchronous memory offers several advantages over asynchronous memory, including higher frequencies and thus higher memory bandwidth, increased reliability, and less standby power. In many designs with asynchronous memory, the memory interfaces with synchronous logic so that the conversion to synchronous memory design is straightforward. To convert asynchronous memory you can move registers from the data path into the memory block.

Synchronous memories are supported in all Altera device families. A memory block is considered synchronous if it uses one of the following read behaviors:

- Memory read occurs in a Verilog `always` block with a clock signal or a VHDL clocked process. The recommended coding style for synchronous memories is to create your design with a registered read output.
- Memory read occurs outside a clocked block, but there is a synchronous read address (that is, the address used in the read statement is registered). This type of logic is not always inferred as a memory block, or may require external bypass logic, depending on the target device architecture.

The synchronous memory structures in Altera devices can differ from the structures in other vendors' devices. For best results, match your design to the target device architecture.

Later sections provide coding recommendations for various memory types. All of these examples are synchronous to ensure that they can be directly mapped into the dedicated memory architecture available in Altera FPGAs.

For additional information about the dedicated memory blocks in your specific device, refer to the appropriate Altera device family data sheet on the Altera website at www.altera.com.

Avoid Unsupported Reset and Control Conditions

To ensure that your HDL code can be implemented in the target device architecture, avoid unsupported reset conditions or other control logic that does not exist in the device architecture.

The RAM contents of Altera memory blocks cannot be cleared with a reset signal during device operation. If your HDL code describes a RAM with a reset signal for the RAM contents, the logic is implemented in regular logic cells instead of a memory block. Altera recommends against putting RAM read or write operations in an `always` block or `process` block with a reset signal. If you want to specify memory contents, initialize the memory as described in "Specifying Initial Memory Contents at Power-Up" on page 13–33 or write the data to the RAM during device operation.

Example 13–9 shows an example of undesirable code where there is a reset signal that clears part of the RAM contents. Avoid this coding style because it is not supported in Altera memories.

Example 13–9. Verilog RAM with Reset Signal that Clears RAM Contents: Not Supported in Device Architecture

```
module clear_ram
(
    input clock, reset, we,
    input [7:0] data_in,
    input [4:0] address,
    output reg [7:0] data_out
);

    reg [7:0] mem [0:31];
    integer i;

    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            mem[address] <= 0;
        else if (we == 1'b1)
            mem[address] <= data_in;

        data_out <= mem[address];
    end
endmodule
```

Example 13–10 shows an example of undesirable code where the reset signal affects the RAM, although the effect may not be intended. Avoid this coding style because it is not supported in Altera memories.

Example 13–10. Verilog RAM with Reset Signal that Affects RAM: Not Supported in Device Architecture

```
module bad_reset
(
    input clock,
    input reset,
    input we,
    input [7:0] data_in,
    input [4:0] address,
    output reg [7:0] data_out,
    input d,
    output reg q
);

    reg [7:0] mem [0:31];
    integer i;

    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            q <= 0;
        else
        begin
            if (we == 1'b1)
                mem[address] <= data_in;

            data_out <= mem[address];
            q <= d;
        end
    end
endmodule
```

In addition to reset signals, other control logic can prevent memory logic from being inferred as a memory block. For example, you cannot use a clock enable on the read address registers in Stratix® devices because doing so affects the output latch of the RAM, and therefore the synthesized result in the device RAM architecture would not match the HDL description. You can use the address stall feature as a read address clock enable in Stratix II, Cyclone® II, Arria® GX, and other newer devices to avoid this limitation. Check the documentation for your device architecture to ensure that your code matches the hardware available in the device.

Check Read-During-Write Behavior

It is important to check the read-during-write behavior of the memory block described in your HDL design as compared to the behavior in your target device architecture. Your HDL source code specifies the memory behavior when you read and write from the same memory address in the same clock cycle. The code specifies that the read returns either the old data at the address, or the new data being written to the address. This behavior is referred to as the read-during-write behavior of the memory block. Altera memory blocks have different read-during-write behavior depending on the target device family, memory mode, and block type.

Synthesis tools map an HDL design into the target device architecture, with the goal of maintaining the functionality described in your source code. Therefore, if your source code specifies unsupported read-during-write behavior for the device RAM blocks, the software must implement the logic outside the RAM hardware in regular logic cells.

One common problem occurs when there is a continuous read in the HDL code, as in the following examples. You should avoid using these coding styles:

```
//Verilog HDL concurrent signal assignment
assign q = ram[raddr_reg];

-- VHDL concurrent signal assignment
q <= ram(raddr_reg);
```

When a write operation occurs, this type of HDL implies that the read should immediately reflect the new data at the address, independent of the read clock. However, that is not the behavior of synchronous memory blocks. In the device architecture, the new data is not available until the next edge of the read clock. Therefore, if the synthesis tool mapped the logic directly to a synchronous memory block, the device functionality and gate-level simulation results would not match the HDL description or functional simulation results. If the write clock and read clock are the same, the synthesis tool can infer memory blocks and add extra bypass logic so that the device behavior matches the HDL behavior. If the write and read clocks are different, the synthesis tool cannot reliably add bypass logic, so the logic is implemented in regular logic cells instead of dedicated RAM blocks. The examples in the following sections discuss some of these differences for read-during-write conditions.

In addition, the MLAB feature in certain device logic array blocks (LABs) does not easily support old data or new data behavior for a read-during-write in the dedicated device architecture. Implementing the extra logic to support this behavior significantly reduces timing performance through the memory.

For best performance in MLAB memories, your design should not depend on the read data during a write operation.

In many synthesis tools, you can specify that the read-during-write behavior is not important to your design; for example, if you never read from the same address to which you write in the same clock cycle. For Quartus II integrated synthesis, add the synthesis attribute `ramstyle` set to `"no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than use the behavior specified by your HDL code. In some cases, this attribute prevents the synthesis tool from using extra logic to implement the memory block, or can allow memory inference when it would otherwise be impossible.

Synchronous RAM blocks require a synchronous read, so Quartus II integrated synthesis packs either data output registers or read address registers into the RAM block. When the read address registers are packed into the RAM block, the read address signals connected to the RAM block contain the next value of the read address signals indexing the HDL variable, which impacts which clock cycle the read and the write occur, and changes the read-during-write conditions. Therefore, bypass logic may still be added to the design to preserve the read-during-write behavior, even if the `"no_rw_check"` attribute is set.

For more information about attribute syntax, the `no_rw_check` attribute value, or specific options for your synthesis tool, refer to your synthesis tool documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

The next section describes how you control the logic implementation in the Altera device, and the following sections provide coding recommendations for various memory types. Each example describes the read-during-write behavior and addresses the support for the memory type in Altera devices.

Controlling Inference and Implementation in Device RAM Blocks

Tools usually do not infer small RAM blocks because small RAM blocks typically can be implemented more efficiently using the registers in regular logic. If you are using Quartus II integrated synthesis, you can direct the software to infer RAM blocks for all sizes with the **Allow Any RAM Size for Recognition** option in the **More Analysis & Synthesis Settings** dialog box.

Some synthesis tools provide options to control the implementation of inferred RAM blocks for Altera devices with synchronous memory blocks. For example, Quartus II integrated synthesis provides the `ramstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block. Quartus II integrated synthesis does not map inferred memory into MLABs unless the HDL code specifies the appropriate `ramstyle` attribute, although the Fitter may map some memories to MLABs.

For details about using the `ramstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

If you want to control the implementation after the RAM function is inferred during synthesis, you can set the `ram_block_type` parameter of the ALTSYNCRAM megafunction. In the Assignment Editor, select **Parameters** in the **Categories** list. You can use the **Node Finder** or drag the appropriate instance from the Project Navigator window to enter the RAM hierarchical instance name. Type `ram_block_type` as the **Parameter Name** and type one of the following memory types supported by your target device family in the **Value** field: "`M-RAM`", "`M512`", "`M4K`", "`M9K`", "`M10K`", "`M20K`", "`M144K`", or "`MLAB`".

You can also specify the maximum depth of memory blocks used to infer RAM or ROM in your design. Apply the `max_depth` synthesis attribute to the declaration of a variable that represents a RAM or ROM in your design file. For example:

```
// Limit the depth of the memory blocks implement "ram" to 512
// This forces the software to use two M512 blocks instead of one M4K block to
implement this RAM

(* max_depth = 512 *) reg [7:0] ram[0:1023];
```

Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior

The code examples in this section show Verilog HDL and VHDL code that infers simple dual-port, single-clock synchronous RAM. Single-port RAM blocks use a similar coding style.

The read-during-write behavior in these examples is to read the old data at the memory address. Refer to "Check Read-During-Write Behavior" on page 13–17 for details. Altera recommends that you use the Old Data Read-During-Write coding style for most RAM blocks as long as your design does not require the RAM location's new value when you perform a simultaneous read and write to that RAM location. For best performance in MLAB memories, use the appropriate attribute so that your design does not depend on the read data during a write operation.

If you require that the read-during-write results in new data, refer to "Single-Clock Synchronous RAM with New Data Read-During-Write Behavior" on page 13–21.

The simple dual-port RAM code samples in Example 13–11 and Example 13–12 map directly into Altera synchronous memory.

Single-port versions of memory blocks (that is, using the same read address and write address signals) can allow better RAM utilization than dual-port memory blocks, depending on the device family.

Example 13–11. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with Old Data Read-During-Write Behavior

```
module single_clk_ram(
    output reg [7:0] q,
    input [7:0] d,
    input [6:0] write_address, read_address,
    input we, clk
);
    reg [7:0] mem [127:0];

    always @ (posedge clk) begin
        if (we)
            mem[write_address] <= d;
        q <= mem[read_address]; // q doesn't get d in this clock cycle
    end
endmodule
```

Example 13–12. VHDL Single-Clock Simple Dual-Port Synchronous RAM with Old Data Read-During-Write Behavior

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

ENTITY single_clock_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 single_clock_ram;

ARCHITECTURE rtl OF single_clock_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            q <= ram_block(read_address);
            -- VHDL semantics imply that q doesn't get data
            -- in this clock cycle
        END IF;
    END PROCESS;
END rtl;
```

Single-Clock Synchronous RAM with New Data Read-During-Write Behavior

The examples in this section describe RAM blocks in which a simultaneous read and write to the same location reads the new value that is currently being written to that RAM location.

To implement this behavior in the target device, synthesis software adds bypass logic around the RAM block. This bypass logic increases the area utilization of the design and decreases the performance if the RAM block is part of the design's critical path. For more information, refer to "Check Read-During-Write Behavior" on page 13–17 for details. If this behavior is not required for your design, use the examples from "Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior" on page 13–19.

The simple dual-port RAM in Example 13–13 and Example 13–14 require the software to create bypass logic around the RAM block.

Single-port versions of the Verilog memory block (that is, using the same read address and write address signals) do not require any logic cells to create bypass logic in the Arria, Stratix, and Cyclone series of devices, because the device memory supports new data read-during-write behavior when in single-port mode (same clock, same read address, and same write address).

Example 13–13. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with New Data Read-During-Write Behavior

```
module single_clock_wr_ram(
    output reg [7:0] q,
    input [7:0] d,
    input [6:0] write_address, read_address,
    input we, clk
);
    reg [7:0] mem [127:0];

    always @ (posedge clk) begin
        if (we)
            mem[write_address] = d;
        q = mem[read_address]; // q does get d in this clock cycle if
                               // we is high
    end
endmodule
```

Example 13–13 is similar to Example 13–11, but Example 13–13 uses a blocking assignment for the write so that the data is assigned immediately.

An alternative way to create a single-clock RAM is to use an assign statement to read the address of `mem` to create the output `q`, as shown in the following coding style example. By itself, the code describes new data read-during-write behavior. However, if the RAM output feeds a register in another hierarchy, a read-during-write results in the old data. Synthesis tools may not infer a RAM block if the tool cannot determine which behavior is described, such as when the memory feeds a hard hierarchical partition boundary. For this reason, avoid using this alternate type of coding style:

```
reg [7:0] mem [127:0];
reg [6:0] read_address_reg;

always @ (posedge clk) begin
   if (we)
      mem[write_address] <= d;

   read_address_reg <= read_address;
end

assign q = mem[read_address_reg];
```

The VHDL sample in Example 13–14 uses a concurrent signal assignment to read from the RAM. By itself, this example describes new data read-during-write behavior. However, if the RAM output feeds a register in another hierarchy, a read-during-write results in the old data. Synthesis tools may not infer a RAM block if the tool cannot determine which behavior is described, such as when the memory feeds a hard hierarchical partition boundary.

Example 13–14. VHDL Single-Clock Simple Dual-Port Synchronous RAM with New Data Read-During-Write Behavior

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

ENTITY single_clock_rw_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 single_clock_rw_ram;

ARCHITECTURE rtl OF single_clock_rw_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;
```

For Quartus II integrated synthesis, if you do not require the read-through-write capability, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than using the behavior specified by your HDL code. As discussed in "Check Read-During-Write Behavior" on page 13–17, this attribute may prevent generation of extra bypass logic but it is not always possible to eliminate the requirement for bypass logic.

Simple Dual-Port, Dual-Clock Synchronous RAM

In dual clock designs, synthesis tools cannot accurately infer the read-during-write behavior because it depends on the timing of the two clocks within the target device. Therefore, the read-during-write behavior of the synthesized design is undefined and may differ from your original HDL code. When Quartus II integrated synthesis infers this type of RAM, it issues a warning because of the undefined read-during-write behavior. Refer to "Check Read-During-Write Behavior" on page 13–17 for details.

The code samples in Example 13–15 and Example 13–16 show Verilog HDL and VHDL code that infers dual-clock synchronous RAM. The exact behavior depends on the relationship between the clocks.

Example 13–15. Verilog HDL Simple Dual-Port, Dual-Clock Synchronous RAM

```
module dual_clock_ram(
   output reg [7:0] q,
   input [7:0] d,
   input [6:0] write_address, read_address,
   input we, clk1, clk2
);
   reg [6:0] read_address_reg;
   reg [7:0] mem [127:0];

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

   always @ (posedge clk2) begin
      q <= mem[read_address_reg];
      read_address_reg <= read_address;
   end
endmodule
```

Example 13–16. VHDL Simple Dual-Port, Dual-Clock Synchronous RAM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY dual_clock_ram 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 dual_clock_ram;
ARCHITECTURE rtl OF dual_clock_ram 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;
```

True Dual-Port Synchronous RAM

The code examples in this section show Verilog HDL and VHDL code that infers true dual-port synchronous RAM. Different synthesis tools may differ in their support for these types of memories. This section describes the inference rules for Quartus II integrated synthesis. This type of RAM inference is supported for the Arria GX, Stratix, and Cyclone series of devices.

Altera synchronous memory blocks have two independent address ports, allowing for operations on two unique addresses simultaneously. A read operation and a write operation can share the same port if they share the same address. The Quartus II software infers true dual-port RAMs in Verilog HDL and VHDL with any combination of independent read or write operations in the same clock cycle, with at most two unique port addresses, performing two reads and one write, two writes and one read, or two writes and two reads in one clock cycle with one or two unique addresses.

In the synchronous RAM block architecture, there is no priority between the two ports. Therefore, if you write to the same location on both ports at the same time, the result is indeterminate in the device architecture. You must ensure your HDL code does not imply priority for writes to the memory block, if you want the design to be implemented in a dedicated hardware memory block. For example, if both ports are defined in the same process block, the code is synthesized and simulated sequentially so that there is a priority between the two ports. If your code does imply a priority, the logic cannot be implemented in the device RAM blocks and is implemented in regular logic cells.

You must also consider the read-during-write behavior of the RAM block to ensure that it can be mapped directly to the device RAM architecture. Refer to "Check Read-During-Write Behavior" on page 13–17 for details.

When a read and write operation occurs on the same port for the same address, the read operation may behave as follows:

- **Read new data—**This mode matches the behavior of synchronous memory blocks.
- **Read old data—**This mode is supported only in device families that support M144, M9k, and MLAB memory blocks.

When a read and write operation occurs on different ports for the same address (also known as mixed port), the read operation may behave as follows:

- **Read new data—**Quartus II integrated synthesis supports this mode by creating bypass logic around the synchronous memory block.
- **Read old data—**Synchronous memory blocks support this behavior.
- **Read don't care—**This behavior is supported on different ports in simple dual-port mode by synchronous memory blocks for all device families except Arria, Arria GX, Cyclone, Cyclone II, MAX, Stratix, and Stratix II device families.

The Verilog HDL single-clock code sample in Example 13–17 maps directly into Altera synchronous memory. When a read and write operation occurs on the same port for the same address, the new data being written to the memory is read. When a read and write operation occurs on different ports for the same address, the old data in the memory is read. Simultaneous writes to the same location on both ports results in indeterminate behavior.

A dual-clock version of this design describes the same behavior, but the memory in the target device will have undefined mixed port read-during-write behavior because it depends on the relationship between the clocks.

Example 13–17. Verilog HDL True Dual-Port RAM with Single Clock

```
module true_dual_port_ram_single_clock
(
    input [(DATA_WIDTH-1):0] data_a, data_b,
    input [(ADDR_WIDTH-1):0] addr_a, addr_b,
    input we_a, we_b, clk,
    output reg [(DATA_WIDTH-1):0] q_a, q_b
);

    parameter DATA_WIDTH = 8;
    parameter ADDR_WIDTH = 6;

    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

    always @ (posedge clk)
    begin // Port A
        if (we_a)
        begin
            ram[addr_a] <= data_a;
            q_a <= data_a;
        end
        else
            q_a <= ram[addr_a];
    end
    always @ (posedge clk)
    begin // Port b
        if (we_b)
        begin
            ram[addr_b] <= data_b;
            q_b <= data_b;
        end
        else
            q_b <= ram[addr_b];
    end

endmodule
```

If you use the following Verilog HDL read statements instead of the `if-else` statements in Example 13–17, the HDL code specifies that the read results in old data when a read operation and write operation occurs at the same time for the same address on the same port or mixed ports. This mode is supported only in device families that support M144, M9k, and MLAB memory blocks.

```
    always @ (posedge clk)
    begin // Port A
      if (we_a)
        ram[addr_a] <= data_a;

      q_a <= ram[addr_a];
    end

    always @ (posedge clk)
    begin // Port B
```

```
    if (we_b)
      ram[addr_b] <= data_b;

    q_b <= ram[addr_b];
  end
```

The VHDL single-clock code sample in Example 13–18 maps directly into Altera synchronous memory. When a read and write operation occurs on the same port for the same address, the new data being written to the memory is read. When a read and write operation occurs on different ports for the same address, the old data in the memory is read. Simultaneous write operations to the same location on both ports results in indeterminate behavior.

A dual-clock version of this design describes the same behavior, but the memory in the target device will have undefined mixed port read-during-write behavior because it depends on the relationship between the clocks.

Example 13–18. VHDL True Dual-Port RAM with Single Clock (Part 1 of 2)

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

entity true_dual_port_ram_single_clock is
   generic (
      DATA_WIDTH : natural := 8;
      ADDR_WIDTH : natural := 6
   );
   port (
      clk : in std_logic;
      addr_a: in natural range 0 to 2**ADDR_WIDTH - 1;
      addr_b: in natural range 0 to 2**ADDR_WIDTH - 1;
      data_a: in std_logic_vector((DATA_WIDTH-1) downto 0);
      data_b: in std_logic_vector((DATA_WIDTH-1) downto 0);
      we_a: in std_logic := '1';
      we_b: in std_logic := '1';
      q_a : out std_logic_vector((DATA_WIDTH -1) downto 0);
      q_b : out std_logic_vector((DATA_WIDTH -1) downto 0)
   );
end true_dual_port_ram_single_clock;

architecture rtl of true_dual_port_ram_single_clock is
   -- Build a 2-D array type for the RAM
   subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
   type memory_t is array((2**ADDR_WIDTH - 1) downto 0) of word_t;
   -- Declare the RAM signal.
   shared variable ram : memory_t;
```

Example 13–19. VHDL True Dual-Port RAM with Single Clock (Part 2 of 2)

```
begin
    process(clk)
    begin
    if(rising_edge(clk)) then -- Port A
        if(we_a = '1') then
            ram(addr_a) <= data_a;

            -- Read-during-write on the same port returns NEW data
            q_a <= data_a;
        else
            -- Read-during-write on the mixed port returns OLD data
            q_a <= ram(addr_a);
        end if;
    end if;
    end process;

    process(clk)
    begin
    if(rising_edge(clk)) then -- Port B
        if(we_b = '1') then
            ram(addr_b) <= data_b;
            -- Read-during-write on the same port returns NEW data
            q_b <= data_b;
        else
            -- Read-during-write on the mixed port returns OLD data
            q_b <= ram(addr_b);
        end if;
    end if;
    end process;

end rtl;
```

Mixed-Width Dual-Port RAM

The RAM code examples in Example 13–20 through Example 13–23 show SystemVerilog and VHDL code that infers RAM with data ports with different widths. This type of logic is not supported in Verilog-1995 or Verilog-2001 because of the requirement for a multi-dimensional array to model the different read width, write width, or both. Different synthesis tools may differ in their support for these memories. This section describes the inference rules for Quartus II integrated synthesis.

The first dimension of the multi-dimensional packed array represents the ratio of the wider port to the narrower port, and the second dimension represents the narrower port width. The read and write port widths must specify a read or write ratio supported by the memory blocks in the target device, or the synthesis tool does not infer a RAM.

Refer to the Quartus II Templates for parameterized examples that you can use for supported combinations of read and write widths, and true dual port RAM examples with two read ports and two write ports for mixed-width writes and reads.

Example 13–20. SystemVerilog Mixed-Width RAM with Read Width Smaller than Write Width

```
module mixed_width_ram    // 256x32 write and 1024x8 read
(
       input [7:0] waddr,
       input [31:0] wdata,
       input we, clk,
       input [9:0] raddr,
       output [7:0] q
);
    logic [3:0][7:0] ram[0:255];
    always_ff@(posedge clk)
       begin
          if(we) ram[waddr] <= wdata;
          q <= ram[raddr / 4][raddr % 4];
       end
endmodule : mixed_width_ram
```

Example 13–21. SystemVerilog Mixed-Width RAM with Read Width Larger than Write Width

```
module mixed_width_ram     // 1024x8 write and 256x32 read
(
       input [9:0] waddr,
       input [31:0] wdata,
       input we, clk,
       input [7:0] raddr,
       output [9:0] q
);
    logic [3:0][7:0] ram[0:255];
    always_ff@(posedge clk)
        begin
          if(we) ram[waddr / 4][waddr % 4] <= wdata;
          q <= ram[raddr];
        end
endmodule : mixed_width_ram
```

Example 13–22. VHDL Mixed-Width RAM with Read Width Smaller than Write Width

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

package ram_types is
    type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
    type ram_t is array (0 to 255) of word_t;
end ram_types;

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.ram_types.all;

entity mixed_width_ram is
    port (
        we, clk : in  std_logic;
        waddr   : in  integer range 0 to 255;
        wdata   : in  word_t;
        raddr   : in  integer range 0 to 1023;
        q       : out std_logic_vector(7 downto 0));
end mixed_width_ram;

architecture rtl of mixed_width_ram is
    signal ram : ram_t;
begin  -- rtl
    process(clk, we)
    begin
        if(rising_edge(clk)) then
            if(we = '1') then
                ram(waddr) <= wdata;
            end if;
            q <= ram(raddr / 4 )(raddr mod 4);
        end if;
    end process;
end rtl;
```

Example 13–23. VHDL Mixed-Width RAM with Read Width Larger than Write Width

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

package ram_types is
   type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
   type ram_t is array (0 to 255) of word_t;
end ram_types;

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.ram_types.all;

entity mixed_width_ram is
   port (
      we, clk : in  std_logic;
      waddr   : in  integer range 0 to 1023;
      wdata   : in  std_logic_vector(7 downto 0);
      raddr   : in  integer range 0 to 255;
      q       : out word_t);
end mixed_width_ram;

architecture rtl of mixed_width_ram is
   signal ram : ram_t;
begin  -- rtl
   process(clk, we)
   begin
      if(rising_edge(clk)) then
         if(we = '1') then
            ram(waddr / 4)(waddr mod 4) <= wdata;
         end if;
         q <= ram(raddr);
      end if;
   end process;
end rtl;
```

RAM with Byte-Enable Signals

The RAM code examples in Example 13–24 and Example 13–25 show SystemVerilog and VHDL code that infers RAM with controls for writing single bytes into the memory word, or byte-enable signals. Byte enables are modeled by creating write expressions with two indices and writing part of a RAM "word." With these implementations, you can also write more than one byte at once by enabling the appropriate byte enables.

This type of logic is not supported in Verilog-1995 or Verilog-2001 because of the requirement for a multidimensional array. Different synthesis tools may differ in their support for these memories. This section describes the inference rules for Quartus II integrated synthesis.

Refer to the Quartus II Templates for parameterized examples that you can use for different address widths, and true dual port RAM examples with two read ports and two write ports.

Example 13–24. SystemVerilog Simple Dual-Port Synchronous RAM with Byte Enable

```
module byte_enabled_simple_dual_port_ram
(
    input we, clk,
    input [5:0] waddr, raddr, // address width = 6
    input [3:0] be,     // 4 bytes per word
    input [31:0] wdata,  // byte width = 8, 4 bytes per word
    output reg [31:0] q  // byte width = 8, 4 bytes per word
);
   // use a multi-dimensional packed array
   //to model individual bytes within the word
   logic [3:0][7:0] ram[0:63];// # words = 1 << address width

   always_ff@(posedge clk)
   begin
       if(we) begin
           if(be[0]) ram[waddr][0] <= wdata[7:0];
           if(be[1]) ram[waddr][1] <= wdata[15:8];
           if(be[2]) ram[waddr][2] <= wdata[23:16];
      if(be[3]) ram[waddr][3] <= wdata[31:24];
       end
   q <= ram[raddr];
   end
endmodule
```

Example 13–25. VHDL Simple Dual-Port Synchronous RAM with Byte Enable

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

entity byte_enabled_simple_dual_port_ram is
port (
    we, clk : in  std_logic;
    waddr, raddr : in  integer range 0 to 63 ;     -- address width = 6
    be      : in  std_logic_vector (3 downto 0);   -- 4 bytes per word
    wdata   : in  std_logic_vector(31 downto 0);   -- byte width = 8
    q       : out std_logic_vector(31 downto 0) ); -- byte width = 8
end byte_enabled_simple_dual_port_ram;

architecture rtl of byte_enabled_simple_dual_port_ram is
  --  build up 2D array to hold the memory
    type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
    type ram_t is array (0 to 63) of word_t;

    signal ram : ram_t;
    signal q_local : word_t;

    begin  -- Re-organize the read data from the RAM to match the output
        unpack: for i in 0 to 3 generate
            q(8*(i+1) - 1 downto 8*i) <= q_local(i);
    end generate unpack;

    process(clk)
    begin
        if(rising_edge(clk)) then
            if(we = '1') then
                if(be(0) = '1') then
                    ram(waddr)(0) <= wdata(7 downto 0);
                end if;
                if be(1) = '1' then
                    ram(waddr)(1) <= wdata(15 downto 8);
                end if;
                if be(2) = '1' then
                    ram(waddr)(2) <= wdata(23 downto 16);
                end if;
                if be(3) = '1' then
                    ram(waddr)(3) <= wdata(31 downto 24);
                end if;
            end if;
            q_local <= ram(raddr);
        end if;
    end process;
end rtl;
```

Specifying Initial Memory Contents at Power-Up

Your synthesis tool may offer various ways to specify the initial contents of an inferred memory.

Certain device memory types do not support initialized memory, such as the M-RAM blocks in Stratix and Stratix II devices.

There are slight power-up and initialization differences between dedicated RAM blocks and the MLAB memory due to the continuous read of the MLAB. Altera dedicated RAM block outputs always power-up to zero and are set to the initial value on the first read. For example, if address `0` is pre-initialized to FF, the RAM block powers up with the output at `0`. A subsequent read after power-up from address `0` outputs the pre-initialized value of FF. Therefore, if a RAM is powered up and an enable (read enable or clock enable) is held low, the power-up output of `0` is maintained until the first valid read cycle. The MLAB is implemented using registers that power-up to `0`, but are initialized to their initial value immediately at power-up or reset. Therefore, the initial value is seen, regardless of the enable status. The Quartus II software maps inferred memory to MLABs when the HDL code specifies an appropriate `ramstyle` attribute.

Quartus II integrated synthesis supports the `ram_init_file` synthesis attribute that allows you to specify a Memory Initialization File (**.mif**) for an inferred RAM block.

For information about the `ram_init_file` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the tool vendor's documentation.

In Verilog HDL, you can use an initial block to initialize the contents of an inferred memory. Quartus II integrated synthesis automatically converts the initial block into a **.mif** file for the inferred RAM. Example 13–26 shows Verilog HDL code that infers a simple dual-port RAM block and corresponding **.mif** file.

Example 13–26. Verilog HDL RAM with Initialized Contents

```
module ram_with_init(
   output reg [7:0] q,
   input [7:0] d,
   input [4:0] write_address, read_address,
   input we, clk
);
   reg [7:0] mem [0:31];
   integer i;

   initial begin
      for (i = 0; i < 32; i = i + 1)
         mem[i] = i[7:0];
   end

   always @ (posedge clk) begin
      if (we)
         mem[write_address] <= d;
      q <= mem[read_address];
   end
endmodule
```

Quartus II integrated synthesis and other synthesis tools also support the `$readmemb` and `$readmemh` commands so that RAM initialization and ROM initialization work identically in synthesis and simulation. Example 13–27 shows an initial block that initializes an inferred RAM block using the `$readmemb` command.

Refer to the *Verilog Language Reference Manual (LRM)* 1364-2001 Section 17.2.8 or the example in the Templates for the Quartus II software for details about the format of the **ram.txt** file.

Example 13–27. Verilog HDL RAM Initialized with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
    $readmemb("ram.txt", ram);
end
```

In VHDL, you can initialize the contents of an inferred memory by specifying a default value for the corresponding signal. Quartus II integrated synthesis automatically converts the default value into a **.mif** file for the inferred RAM. Example 13–28 shows VHDL code that infers a simple dual-port RAM block and corresponding **.mif** file.

Example 13–28. VHDL RAM with Initialized Contents

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY ram_with_init IS
    PORT(
            clock: IN STD_LOGIC;
            data: IN UNSIGNED (7 DOWNTO 0);
            write_address: IN integer RANGE 0 to 31;
            read_address: IN integer RANGE 0 to 31;
            we: IN std_logic;
            q: OUT UNSIGNED (7 DOWNTO 0));
END;

ARCHITECTURE rtl OF ram_with_init IS

    TYPE MEM IS ARRAY(31 DOWNTO 0) OF unsigned(7 DOWNTO 0);
    FUNCTION initialize_ram
        return MEM is
        variable result : MEM;
    BEGIN
        FOR i IN 31 DOWNTO 0 LOOP
            result(i) := to_unsigned(natural(i), natural'(8));
        END LOOP;
        RETURN result;
    END initialize_ram;

    SIGNAL ram_block : MEM := initialize_ram;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
            ram_block(write_address) <= data;
            END IF;
            q <= ram_block(read_address);
        END IF;
    END PROCESS;
END rtl;
```

Inferring ROM Functions from HDL Code

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, and only for device families that have dedicated memory blocks.

ROMs are inferred when a `CASE` statement exists in which a value is set to a constant for every choice in the `case` statement. Because small ROMs typically achieve the best performance when they are implemented using the registers in regular logic, each ROM function must meet a minimum size requirement to be inferred and placed into memory.

If you use Quartus II integrated synthesis, you can direct the software to infer ROM blocks for all sizes with the **Allow Any ROM Size for Recognition** option in the **More Analysis & Synthesis Settings** dialog box.

Some synthesis tools provide options to control the implementation of inferred ROM blocks for Altera devices with synchronous memory blocks. For example, Quartus II integrated synthesis provides the `romstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block.

For details about using the `romstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the appropriate chapter in the Synthesis section in volume 1 of the *Quartus II Handbook*.

Because formal verification tools do not support ROM megafunctions, Quartus II integrated synthesis does not infer ROM megafunctions when a formal verification tool is selected. When you are using a formal verification flow, Altera recommends that you instantiate ROM megafunction blocks in separate entities or modules that contain only the ROM logic, because you may need to treat the entity or module as a black box during formal verification.

The Verilog HDL and VHDL code samples in Example 13–29 through Example 13–32 on pages 13–37 through 13–39 infer synchronous ROM blocks. Depending on the device family's dedicated RAM architecture, the ROM logic may have to be synchronous; refer to the device family handbook for details.

For device architectures with synchronous RAM blocks, such as the Arria series, Cyclone series, or Stratix series devices and newer device families, either the address or the output must be registered for synthesis software to infer a ROM block. When your design uses output registers, the synthesis software implements registers from the input registers of the RAM block without affecting the functionality of the ROM. If you register the address, the power-up state of the inferred ROM can be different from the HDL design. In this scenario, the synthesis software issues a warning. The Quartus II Help explains the condition under which the functionality changes when you use Quartus II integrated synthesis.

The ROM code examples in Example 13–29 through Example 13–32 on pages 13–37 through 13–39 map directly to the Altera memory architecture.

Example 13–29. Verilog HDL Synchronous ROM

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

    reg [5:0] data_out;

    always @ (posedge clock)
    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
```

Example 13–30. VHDL Synchronous ROM

```
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
BEGIN
PROCESS (clock)
    BEGIN
    IF rising_edge (clock) THEN
        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 IF;
    END PROCESS;
END rtl;
```

Example 13–31. Verilog HDL Dual-Port Synchronous ROM Using readmemb

```
module dual_port_rom (
    input [(addr_width-1):0] addr_a, addr_b,
    input clk,
    output reg [(data_width-1):0] q_a, q_b
);
    parameter data_width = 8;
    parameter addr_width = 8;

    reg [data_width-1:0] rom[2**addr_width-1:0];

    initial // Read the memory contents in the file
            //dual_port_rom_init.txt.
    begin
        $readmemb("dual_port_rom_init.txt", rom);
    end

    always @ (posedge clk)
    begin
        q_a <= rom[addr_a];
        q_b <= rom[addr_b];
    end
endmodule
```

Example 13–32. VHDL Dual-Port Synchronous ROM Using Initialization Function

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

entity dual_port_rom is
    generic (
        DATA_WIDTH : natural := 8;
        ADDR_WIDTH : natural := 8
    );
    port (
        clk    : in std_logic;
        addr_a: in natural range 0 to 2**ADDR_WIDTH - 1;
        addr_b: in natural range 0 to 2**ADDR_WIDTH - 1;
        q_a    : out std_logic_vector((DATA_WIDTH -1) downto 0);
        q_b    : out std_logic_vector((DATA_WIDTH -1) downto 0)
    );
end entity;

architecture rtl of dual_port_rom is
    -- Build a 2-D array type for the ROM
    subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
    type memory_t is array(addr_a'high downto 0) of word_t;

    function init_rom
        return memory_t is
        variable tmp : memory_t := (others => (others => '0'));
    begin
        for addr_pos in 0 to 2**ADDR_WIDTH - 1 loop
            -- Initialize each address with the address itself
            tmp(addr_pos) := std_logic_vector(to_unsigned(addr_pos,
DATA_WIDTH));
        end loop;
        return tmp;
    end init_rom;

    -- Declare the ROM signal and specify a default initialization value.
    signal rom : memory_t := init_rom;
begin
    process(clk)
    begin
    if (rising_edge(clk)) then
        q_a <= rom(addr_a);
        q_b <= rom(addr_b);
    end if;
    end process;
end rtl;
```

Shift Registers—Inferring the ALTSHIFT_TAPS Megafunction from HDL Code

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 the shift registers must have the following characteristics:

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

When you use a formal verification flow, Altera recommends that you create shift register blocks in separate entities or modules containing only the shift register logic, because you might have to treat the entity or module as a black box during formal verification.

Because formal verification tools do not support shift register megafunctions, Quartus II integrated synthesis does not infer the ALTSHIFT_TAPS megafunction when a formal verification tool is selected. You can select EDA tools for use with your design on the **EDA Tool Settings** page of the **Settings** dialog box in the Quartus II software.

For more information about the ALTSHIFT_TAPS megafunction, refer to the *ALTSHIFT_TAPS Megafunction User Guide*.

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

Quartus II integrated synthesis uses the following guidelines which are common in other EDA tools. The Quartus II software determines whether to infer the ALTSHIFT_TAPS megafunction based on the width of the registered bus (W), the length between each tap (L), and the number of taps (N). If the **Auto Shift Register Recognition** setting is set to **Auto**, Quartus II integrated synthesis uses the **Optimization Technique** setting, logic and RAM utilization information about the design, and timing information from **Timing-Driven Synthesis** to determine which shift registers are implemented in RAM blocks for logic.

- If the registered bus width is one ($W = 1$), 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 ($W > 1$), 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 logic to decode the read and write counters. This situation occurs because for different sizes of shift registers, external decode logic that uses logic elements (LEs) or ALMs is required to implement the function. This decode logic eliminates the performance and utilization advantages 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 a Verilog HDL or VHDL output file for simulation tools because their node names do not exist after synthesis.

If your design uses a shift enable signal to infer a shift register, the shift register will not be implemented into MLAB memory, but can use only dedicated RAM blocks.

Simple Shift Register

The code samples in Example 13–33 and Example 13–34 show a simple, single-bit wide, 64-bit long shift register. The synthesis software implements the register (`W` = 1 and `M` = 64) in an ALTSHIFT_TAPS megafunction for supported devices and maps it to RAM in supported devices, which may be placed in dedicated RAM blocks or MLAB memory. If the length of the register is less than 64 bits, the software implements the shift register in logic.

Example 13–33. 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
```

Example 13–34. VHDL Single-Bit Wide, 64-Bit Long Shift Register

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
ENTITY shift_1x64 IS
    PORT (
        clk: IN STD_LOGIC;
        shift: IN STD_LOGIC;
        sr_in: IN STD_LOGIC;
        sr_out: OUT STD_LOGIC
    );
END shift_1x64;

ARCHITECTURE arch OF shift_1x64 IS
    TYPE sr_length IS ARRAY (63 DOWNTO 0) OF STD_LOGIC;
    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_out <= sr(63);
END arch;
```

Shift Register with Evenly Spaced Taps

The code samples in Example 13–35 and Example 13–36 show a Verilog HDL and VHDL 8-bit wide, 64-bit long shift register (`W > 1` and `M = 64`) with evenly spaced taps at 15, 31, and 47. The synthesis software implements this function in a single ALTSHIFT_TAPS megafunction and maps it to RAM in supported devices, which is allowed placement in dedicated RAM blocks or MLAB memory.

Example 13–35. 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
```

Example 13–36. 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;
```

Coding Guidelines for Registers and Latches

This section provides device-specific coding recommendations for Altera registers and latches. Understanding the architecture of the target Altera device helps ensure that your code produces the expected results and achieves the optimal quality of results.

This section provides guidelines in the following areas:

- "Register Power-Up Values in Altera Devices"
- "Secondary Register Control Signals Such as Clear and Clock Enable" on page 13–46
- "Latches" on page 13–50

Register Power-Up Values in Altera Devices

Registers in the device core always power up to a low (0) logic level on all Altera devices. If your design specifies a power-up level other than 0, synthesis tools can implement logic that causes registers to behave as if they were powering up to a high (1) logic level.

If your design uses a preset signal on a device that does not support presets in the register architecture, your synthesis tool may convert the preset signal to a clear signal, which requires synthesis to perform an optimization referred to as NOT gate push-back. NOT gate push-back adds an inverter to the input and the output of the register so that the reset and power-up conditions will appear to be high, and the device operates as expected. In this case, your synthesis tool may issue a message informing you about the power-up condition. The register itself powers up low, but the register output is inverted, so the signal that arrives at all destinations is high.

Due to these effects, if you specify a non-zero reset value, you may cause your synthesis tool to use the asynchronous clear (`aclr`) signals available on the registers to implement the high bits with NOT gate push-back. In that case, the registers look as though they power up to the specified reset value.

When an asynchronous load (`aload`) signal is available in the device registers, your synthesis tools can implement a reset of `1` or `0` value by using an asynchronous load of `1` or `0`. When the synthesis tool uses a load signal, it is not performing NOT gate push-back, so the registers power up to a `0` logic level.

For additional details, refer to the appropriate device family handbook or the appropriate handbook on the Altera website.

Designers typically use an explicit reset signal for the design, which forces all registers into their appropriate values after reset. Altera recommends this practice to reset the device after power-up to restore the proper state.

You can make your design more stable and avoid potential glitches by synchronizing external or combinational logic of the device architecture before you drive the asynchronous control ports of registers.

For additional information about good synchronous design practices, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Specifying a Power-Up Value

If you want to force a particular power-up condition for your design, you can use the synthesis options available in your synthesis tool. With Quartus II integrated synthesis, you can apply the **Power-Up Level** logic option. You can also apply the option with an `altera_attribute` assignment in your source code. Using this option forces synthesis to perform NOT gate push-back because synthesis tools cannot actually change the power-up states of core registers.

You can apply the Quartus II integrated synthesis **Power-Up Level** logic option to a specific register or to a design entity, module, or subdesign. If you do so, every register in that block receives the value. Registers power up to `0` by default; therefore, you can use this assignment to force all registers to power up to `1` using NOT gate push-back.

Setting the **Power-Up Level** to a logic level of high for a large design entity could degrade the quality of results due to the number of inverters that are required. In some situations, issues are caused by enable signal inference or secondary control logic inference. It may also be more difficult to migrate such a design to an ASIC.

You can simulate the power-up behavior in a functional simulation if you use initialization.

The **Power-Up Level** option and the `altera_attribute` assignment are described in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Some synthesis tools can also read the default or initial values for registered signals and implement this behavior in the device. For example, Quartus II integrated synthesis converts default values for registered signals into **Power-Up Level** settings. When the Quartus II software reads the default values, the synthesized behavior matches the power-up state of the HDL code during a functional simulation.

For example, the code samples in Example 13–37 and Example 13–38 both infer a register for `q` and set its power-up level to high.

Example 13–37. Verilog Register with High Power-Up Value

```
reg q = 1'b1; //q has a default value of '1'

always @ (posedge clk)
begin
    q <= d;
end
```

Example 13–38. VHDL Register with High Power-Up Level

```
SIGNAL q : STD_LOGIC := '1'; -- q has a default value of '1'

PROCESS (clk, reset)
BEGIN
    IF (rising_edge(clk)) THEN
        q <= d;
    END IF;
END PROCESS;
```

There may also be undeclared default power-up conditions based on signal type. If you declare a VHDL register signal as an integer, Quartus II synthesis attempts to use the left end of the integer range as the power-up value. For the default signed integer type, the default power-up value is the highest magnitude negative integer (100…001). For an unsigned integer type, the default power-up value is `0`.

If the target device architecture does not support two asynchronous control signals, such as `aclr` and `aload`, you cannot set a different power-up state and reset state. If the NOT gate push-back algorithm creates logic to set a register to `1`, that register will power-up high. If you set a different power-up condition through a synthesis assignment or initial value, the power-up level is ignored during synthesis.

Secondary Register Control Signals Such as Clear and Clock Enable

The registers in Altera FPGAs provide a number of secondary control signals (such as clear and enable signals) that you can use to implement control logic for each register without using extra logic cells. Device families vary in their support for secondary signals, so consult the device family data sheet to verify which signals are available in your target device.

To make the most efficient use of the signals in the device, your HDL code should match the device architecture as closely as possible. The control signals have a certain priority due to the nature of the architecture, so your HDL code should follow that priority where possible.

Your synthesis tool can emulate any control signals using regular logic, so achieving functionally correct results is always possible. However, if your design requirements are flexible in terms of which control signals are used and in what priority, match your design to the target device architecture to achieve the most efficient results. If the priority of the signals in your design is not the same as that of the target architecture, extra logic may be required to implement the control signals. This extra logic uses additional device resources and can cause additional delays for the control signals.

In addition, there are certain cases where using logic other than the dedicated control logic in the device architecture can have a larger impact. For example, the clock enable signal has priority over the synchronous reset or clear signal in the device architecture. The clock enable turns off the clock line in the LAB, and the clear signal is synchronous. Therefore, in the device architecture, the synchronous clear takes effect only when a clock edge occurs.

If you code a register with a synchronous clear signal that has priority over the clock enable signal, the software must emulate the clock enable functionality using data inputs to the registers. Because the signal does not use the clock enable port of a register, you cannot apply a Clock Enable Multicycle constraint. In this case, following the priority of signals available in the device is clearly the best choice for the priority of these control signals, and using a different priority causes unexpected results with an assignment to the clock enable signal.

The priority order for secondary control signals in Altera devices differs from the order for other vendors' devices. If your design requirements are flexible regarding priority, verify that the secondary control signals meet design performance requirements when migrating designs between FPGA vendors and try to match your target device architecture to achieve the best results.

The signal order is the same for all Altera device families, although, as noted previously, not all device families provide every signal. The following priority order is observed:

1. Asynchronous Clear, `aclr`—highest priority
2. Asynchronous Load, `aload`
3. Enable, `ena`
4. Synchronous Clear, `sclr`
5. Synchronous Load, `sload`
6. Data In, `data`—lowest priority

The following examples provide Verilog HDL and VHDL code that creates a register with the `aclr`, `aload`, and `ena` control signals.

MAX® 3000 and MAX 7000 devices include a dedicated preset signal, which has second priority after `aclr`, and is not included in the following examples.

The Verilog HDL example (Example 13–39) does not have adata on the sensitivity list, but the VHDL example (Example 13–40) does. This is a limitation of the Verilog HDL language—there is no way to describe an asynchronous load signal (in which q toggles if adata toggles while aload is high). All synthesis tools should infer an aload signal from this construct despite this limitation. When they perform such inference, you may see information or warning messages from the synthesis tool.

Example 13–39. Verilog HDL D-Type Flipflop (Register) with ena, aclr, and aload Control Signals

```
module dff_control(clk, aclr, aload, ena, data, adata, q);
    input clk, aclr, aload, ena, data, adata;
    output q;

    reg q;

    always @ (posedge clk or posedge aclr or posedge aload)
    begin
        if (aclr)
            q <= 1'b0;
        else if (aload)
            q <= adata;
        else if (ena)
            q <= data;
    end
endmodule
```

Example 13–40. VHDL D-Type Flipflop (Register) with ena, aclr, and aload Control Signals (Part 1 of 2)

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

ENTITY dff_control IS
    PORT (
       clk: IN STD_LOGIC;
       aclr: IN STD_LOGIC;
       aload: IN STD_LOGIC;
       adata: IN STD_LOGIC;
       ena: IN STD_LOGIC;
      data: IN STD_LOGIC;
q: OUT STD_LOGIC
    );
END dff_control;
```

Example 13–40. VHDL D-Type Flipflop (Register) with ena, aclr, and aload Control Signals (Part 2 of 2)

```
ARCHITECTURE rtl OF dff_control IS
BEGIN
    PROCESS (clk, aclr, aload, adata)
    BEGIN
IF (aclr = '1') THEN
q <= '0';
ELSIF (aload = '1') THEN
q <= adata;
ELSE
          IF (clk = '1' AND clk'event) THEN
             IF (ena ='1') THEN
q <= data;
             END IF;
          END IF;
       END IF;
    END PROCESS;
END rtl;
```

Creating many registers with different `sload` and `sclr` signals can make packing the registers into LABs difficult for the Quartus II Fitter because the `sclr` and `sload` signals are LAB-wide signals. In addition, using the LAB-wide `sload` signal prevents the Fitter from packing registers using the quick feedback path in the device architecture, which means that some registers cannot be packed with other logic.

Synthesis tools typically restrict use of `sload` and `sclr` signals to cases in which there are enough registers with common signals to allow good LAB packing. Using the look-up table (LUT) to implement the signals is always more flexible if it is available. Because different device families offer different numbers of control signals, inference of these signals is also device-specific. For example, because Stratix II devices have more flexibility than Stratix devices with respect to secondary control signals, synthesis tools might infer more `sload` and `sclr` signals for Stratix II devices.

If you use these additional control signals, use them in the priority order that matches the device architecture. To achieve the most efficient results, ensure the `sclr` signal has a higher priority than the `sload` signal in the same way that `aclr` has higher priority than `aload` in the previous examples. Remember that the register signals are not inferred unless the design meets the conditions described previously. However, if your HDL described the desired behavior, the software always implements logic with the correct functionality.

In Verilog HDL, the following code for `sload` and `sclr` could replace the `if (ena) q <= data;` statements in the Verilog HDL in Example 13–39 (after adding the control signals to the module declaration).

Example 13–41. Verilog HDL sload and sclr Control Signals

```
if  (ena) begin
   if  (sclr)
     q <= 1'b0;
   else if  (sload)
     q <= sdata;
   else
     q <= data;
end
```

In VHDL, the following code for sload and sclr could replace the IF (ena ='1') THEN q <= data; END IF; statements in the VHDL in Example 13–40 on page 13–48 (after adding the control signals to the entity declaration).

Example 13–42. VHDL sload and sclr Control Signals

```
IF (ena ='1') THEN
   IF (sclr = '1') THEN
     q <= '0';
   ELSIF (sload = '1') THEN
     q <= sdata;
   ELSE
     q <= data;
   END IF;
END IF;
```

Latches

A latch is a small combinational loop that holds the value of a signal until a new value is assigned.

Altera recommends that you design without the use of latches whenever possible.

For additional information about the issues involved in designing with latches and combinational loops, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Latches can be inferred from HDL code when you did not intend to use a latch, as described in "Unintentional Latch Generation". If you do intend to infer a latch, it is important to infer it correctly to guarantee correct device operation as detailed in "Inferring Latches Correctly" on page 13–51.

Unintentional Latch Generation

When you are 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 inferred latches. If your code unintentionally creates a latch, make code changes to remove the latch.

A latch is required if a signal is assigned a value outside of a clock edge (for example, with an asynchronous reset), but is not assigned a value in an edge-triggered design block. An unintentional latch may be generated if your HDL code assigns a value to a signal in an edge-triggered design block, but that logic is removed during synthesis. For example, when a CASE or IF statement tests the value of a condition with a parameter or generic that evaluates to FALSE, any logic or signal assignment in that statement is not required and is optimized away during synthesis. This optimization may result in a latch being generated for the signal.

Latches have limited support in formal verification tools. Therefore, ensure that you do not infer latches unintentionally.

The `full_case` attribute can be used in Verilog HDL designs to treat unspecified cases as don't care values (`X`). However, using the `full_case` attribute can cause simulation mismatches because this attribute is a synthesis-only attribute, so simulation tools still treat the unspecified cases as latches.

For more information about using attributes in your synthesis tool, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*. The *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook* provides an example explaining possible simulation mismatches.

Omitting the final `else` or `when others` clause in an `if` or `case` statement can also generate a latch. Don't care (`X`) assignments on the default conditions are useful in preventing latch generation. For the best logic optimization, assign the default `case` or final `else` value to don't care (`X`) instead of a logic value.

The VHDL code sample in Example 13–43 prevents unintentional latches. Without the final `else` clause, this code creates unintentional latches to cover the remaining combinations of the `sel` inputs. When you are targeting a Stratix device with this code, omitting the final `else` condition can cause the synthesis software to use up to six LEs, instead of the three it uses with the `else` statement. Additionally, assigning the final `else` clause to `1` instead of `X` can result in slightly more LEs, because the synthesis software cannot perform as much optimization when you specify a constant value compared to a don't care value.

Example 13–43. 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;
```

Inferring Latches Correctly

Synthesis tools can infer a latch that does not exhibit the glitch and timing hazard problems typically associated with combinational loops.

Timing analysis does not completely model latch timing in some cases. Do not use latches unless required by your design, and you fully understand the impact of using the latches.

When using Quartus II integrated synthesis, latches that are inferred by the software are reported in the **User-Specified and Inferred Latches** section of the Compilation Report. This report indicates whether the latch is considered safe and free of timing hazards.

If a latch or combinational loop in your design is not listed in the **User-Specified and Inferred Latches** section, it means that it was not inferred as a safe latch by the software and is not considered glitch-free.

All combinational loops listed in the **Analysis & Synthesis Logic Cells Representing Combinational Loops** table in the Compilation Report are at risk of timing hazards. These entries indicate possible problems with your design that you should investigate. However, it is possible to have a correct design that includes combinational loops. For example, it is possible that the combinational loop cannot be sensitized. This can occur in cases where there is an electrical path in the hardware, but either the designer knows that the circuit never encounters data that causes that path to be activated, or the surrounding logic is set up in a mutually exclusive manner that prevents that path from ever being sensitized, independent of the data input.

For macrocell-based devices, such as MAX 7000 and MAX 3000, all data (D-type) latches and set-reset (S-R) latches listed in the **Analysis & Synthesis User-Specified and Inferred Latches** table have an implementation free of timing hazards, such as glitches. The implementation includes both a cover term to ensure there is no glitching and a single macrocell in the feedback loop.

For 4-input LUT-based devices, such as Stratix devices, the Cyclone series, and MAX II devices, all latches in the **User-Specified and Inferred Latches** table with a single LUT in the feedback loop are free of timing hazards when a single input changes. Because of the hardware behavior of the LUT, the output does not glitch when a single input toggles between two values that are supposed to produce the same output value, such as a D-type input toggling when the enable input is inactive or a set input toggling when a reset input with higher priority is active. This hardware behavior of the LUT means that no cover term is required for a loop around a single LUT. The Quartus II software uses a single LUT in the feedback loop whenever possible. A latch that has data, enable, set, and reset inputs in addition to the output fed back to the input cannot be implemented in a single 4-input LUT. If the Quartus II software cannot implement the latch with a single-LUT loop because there are too many inputs, the **User-Specified and Inferred Latches** table indicates that the latch is not free of timing hazards.

For 6-input LUT-based devices, the software can implement all latch inputs with a single adaptive look-up table (ALUT) in the combinational loop. Therefore, all latches in the **User-Specified and Inferred Latches** table are free of timing hazards when a single input changes.

If a latch is listed as a safe latch, other optimizations performed by the Quartus II software, such as physical synthesis netlist optimizations in the Fitter, maintain the hazard-free performance.

To ensure hazard-free behavior, only one control input can change at a time. Changing two inputs simultaneously, such as deasserting set and reset at the same time, or changing data and enable at the same time, can produce incorrect behavior in any latch.

Quartus II integrated synthesis infers latches from `always` blocks in Verilog HDL and `process` statements in VHDL, but not from continuous assignments in Verilog HDL or concurrent signal assignments in VHDL. These rules are the same as for register inference. The software infers registers or flipflops only from `always` blocks and `process` statements.

The Verilog HDL code sample in Example 13–44 infers a S-R latch correctly in the Quartus II software.

Example 13–44. Verilog HDL Set-Reset Latch

```
module simple_latch (
   input SetTerm,
   input ResetTerm,
   output reg LatchOut
   );

   always @ (SetTerm or ResetTerm) begin
      if (SetTerm)
         LatchOut = 1'b1
      else if (ResetTerm)
         LatchOut = 1'b0
   end
endmodule
```

The VHDL code sample in Example 13–45 infers a D-type latch correctly in the Quartus II software.

Example 13–45. VHDL Data Type Latch

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

ENTITY simple_latch IS
    PORT (
       enable, data    : IN STD_LOGIC;
       q               : OUT STD_LOGIC
    );
END simple_latch;

ARCHITECTURE rtl OF simple_latch IS
BEGIN

    latch : PROCESS (enable, data)
       BEGIN
       IF (enable = '1') THEN
          q <= data;
       END IF;
    END PROCESS latch;
END rtl;
```

The following example shows a Verilog HDL continuous assignment that does not infer a latch in the Quartus II software:

```
assign latch_out = (~en & latch_out) | (en & data);
```

The behavior of the assignment is similar to a latch, but it may not function correctly as a latch, and its timing is not analyzed as a latch.

Quartus II integrated synthesis also creates safe latches when possible for instantiations of the LPM_LATCH megafunction. You can use this megafunction to create a latch with any combination of data, enable, set, and reset inputs. The same limitations apply for creating safe latches as for inferring latches from HDL code.

Inferring the Altera LPM_LATCH function in another synthesis tool ensures that the implementation is also recognized as a latch in the Quartus II software. If a third-party synthesis tool implements a latch using the LPM_LATCH megafunction, the Quartus II integrated synthesis lists the latch in the **User-Specified and Inferred Latches** table in the same way as it lists latches created in HDL source code. The coding style necessary to produce an LPM_LATCH implementation may depend on your synthesis tool. Some third-party synthesis tools list the number of LPM_LATCH functions that are inferred.

For LUT-based families, the Fitter uses global routing for control signals, including signals that Analysis and Synthesis identifies as latch enables. In some cases the global insertion delay may decrease the timing performance. If necessary, you can turn off the **Quartus II Global Signal** logic option to manually prevent the use of global signals. Global latch enables are listed in the **Global & Other Fast Signals** table in the Compilation Report.

General Coding Guidelines

This section helps you understand how synthesis tools map various types of HDL code into the target Altera device. Following Altera recommended coding styles, and in some cases designing logic structures to match the appropriate device architecture, can provide significant improvements in the design's quality of results.

This section provides coding guidelines for the following logic structures:

- "Tri-State Signals". This section explains how to create tri-state signals for bidirectional I/O pins.
- "Clock Multiplexing" on page 13–56. This section provides recommendations for multiplexing clock signals.
- "Adder Trees" on page 13–59. This section explains the different coding styles that lead to optimal results for devices with 4-input LUTs and 6-input ALUTs.
- "State Machines" on page 13–61. This section helps ensure the best results when you use state machines.
- "Multiplexers" on page 13–68. This section explains how multiplexers can be synthesized, addresses common problems, and provides guidelines to achieve optimal resource utilization.
- "Cyclic Redundancy Check Functions" on page 13–71. This section provides guidelines for getting good results when designing Cyclic Redundancy Check (CRC) functions.
- "Comparators" on page 13–73. This section explains different comparator implementations and provides suggestions for controlling the implementation.

- "Counters" on page 13–74. This section provides guidelines to ensure your counter design targets the device architecture optimally.

Tri-State Signals

When you target Altera devices, you should use tri-state signals only 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.

Synthesis tools implement designs with internal tri-state signals correctly in Altera devices using multiplexer logic, but Altera does not recommend this coding practice.

In hierarchical block-based or incremental design flows, a hierarchical boundary cannot contain any bidirectional ports, unless the lower-level bidirectional port is connected directly through the hierarchy to a top-level output pin without connecting to any other design logic. If you use boundary tri-states in a lower-level block, synthesis software must push the tri-states through the hierarchy to the top level to make use of the tri-state drivers on output pins of Altera devices. Because pushing tri-states requires optimizing through hierarchies, lower-level tri-states are restricted with block-based design methodologies.

The code in Example 13–46 and Example 13–47 show Verilog HDL and VHDL code that creates tri-state bidirectional signals.

Example 13–46. Verilog HDL Tri-State Signal

```
module tristate (myinput, myenable, mybidir);
    input myinput, myenable;
    inout mybidir;
    assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

Example 13–47. VHDL Tri-State Signal

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

ENTITY tristate IS
PORT (
   mybidir : INOUT STD_LOGIC;
   myinput : IN STD_LOGIC;
   myenable : IN STD_LOGIC
   );
END tristate;

ARCHITECTURE rtl OF tristate IS
BEGIN
    mybidir <= 'Z' WHEN (myenable = '0') ELSE myinput;
END rtl;
```

Clock Multiplexing

Clock multiplexing is sometimes used to operate the same logic function with different clock sources. This type of logic can introduce glitches that create functional problems, and the delay inherent in the combinational logic can lead to timing problems. Clock multiplexers trigger warnings from a wide range of design rule check and timing analysis tools.

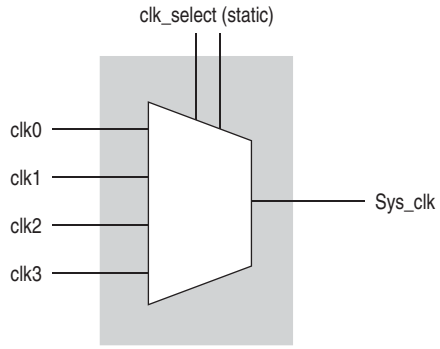
Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature or the Clock Control Block available in certain Altera devices. These dedicated hardware blocks avoid glitches, ensure that you use global low-skew routing lines, and avoid any possible hold time problems on the device due to logic delay on the clock line. Many Altera devices also support dynamic PLL reconfiguration, which is the safest and most robust method of changing clock rates during device operation.

Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures. Also refer to the *ALTCLKCTRL Megafunction User Guide*, the *ALTPLL Megafunction User Guide*, and the *Phase-Locked Loops Reconfiguration (ALTPLL_RECONFIG) Megafunction User Guide*.

If you implement a clock multiplexer in logic cells because the design has too many clocks to use the clock control block, or if dynamic reconfiguration is too complex for your design, it is important to consider simultaneous toggling inputs and ensure glitch-free transitions.

Figure 13–2 shows a simple representation of a clock multiplexer (mux) in a device with 6-input LUTs.

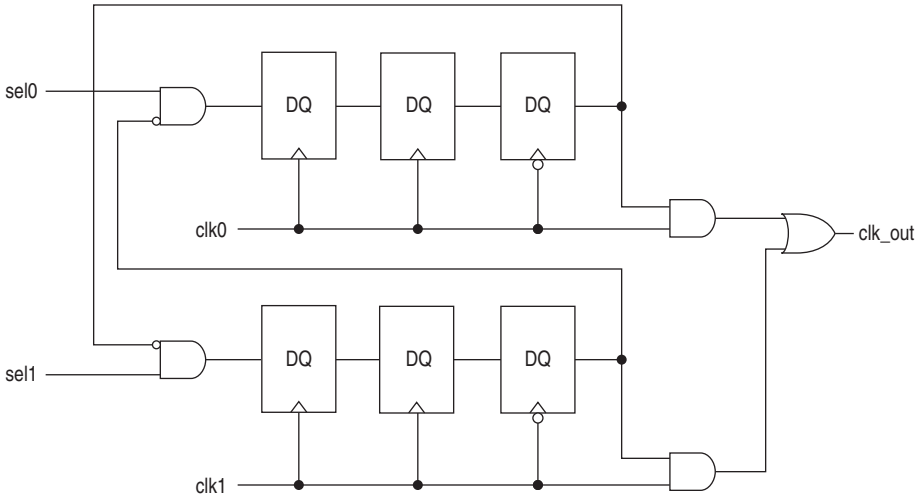
Figure 13–2. Simple Clock Multiplexer in a 6-Input LUT



The data sheet for your target device describes how LUT outputs may glitch during a simultaneous toggle of input signals, independent of the LUT function. Although, in practice, the 4:1 MUX function does not generate detectable glitches during simultaneous data input toggles, it is possible to construct cell implementations that do exhibit significant glitches, so this simple clock mux structure is not recommended. An additional problem with this implementation is that the output behaves erratically during a change in the `clk_select` signals. This behavior could create timing violations on all registers fed by the system clock and result in possible metastability.

A more sophisticated clock select structure can eliminate the simultaneous toggle and switching problems, as in Figure 13–3.

Figure 13–3. Glitch-Free Clock Multiplexer Structure



This structure can be generalized for any number of clock channels. Example 13–48 contains a parameterized version in Verilog HDL. The design enforces that no clock activates until all others have been inactive for at least a few cycles, and that activation occurs while the clock is low. The design applies a `synthesis_keep` directive to the AND gates on the right side of the figure, which ensures there are no simultaneous toggles on the input of the `clk_out` OR gate.

Switching from clock A to clock B requires that clock A continue to operate for at least a few cycles. If the old clock stops immediately, the design sticks. The select signals are implemented as a "one-hot" control in this example, but you can use other encoding if you prefer. The input side logic is asynchronous and is not critical. This design can tolerate extreme glitching during the switch process.

Example 13–48. Verilog HDL Clock Multiplexing Design to Avoid Glitches

```
module clock_mux (clk,clk_select,clk_out);

   parameter num_clocks = 4;

   input [num_clocks-1:0] clk;
   input [num_clocks-1:0] clk_select; // one hot
   output clk_out;

   genvar i;

   reg [num_clocks-1:0] ena_r0;
   reg [num_clocks-1:0] ena_r1;
   reg [num_clocks-1:0] ena_r2;
   wire [num_clocks-1:0] qualified_sel;

   // A look-up-table (LUT) can glitch when multiple inputs
   // change simultaneously. Use the keep attribute to
   // insert a hard logic cell buffer and prevent
   // the unrelated clocks from appearing on the same LUT.

   wire [num_clocks-1:0] gated_clks /* synthesis keep */;

   initial begin
      ena_r0 = 0;
      ena_r1 = 0;
      ena_r2 = 0;
   end

   generate
      for (i=0; i<num_clocks; i=i+1)
      begin : lp0
         wire [num_clocks-1:0] tmp_mask;
         assign tmp_mask = {num_clocks{1'b1}} ^ (1 << i);

         assign qualified_sel[i] = clk_select[i] & (~|(ena_r2 & tmp_mask));

         always @(posedge clk[i]) begin
            ena_r0[i] <= qualified_sel[i];
            ena_r1[i] <= ena_r0[i];
         end

         always @(negedge clk[i]) begin
            ena_r2[i] <= ena_r1[i];
         end

         assign gated_clks[i] = clk[i] & ena_r2[i];
      end
   endgenerate

   // These will not exhibit simultaneous toggle by construction
   assign clk_out = |gated_clks;

endmodule
```

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 using a large adder tree is a finite impulse response (FIR) correlator. Using a pipelined binary or ternary adder tree appropriately can greatly improve the quality of your results.

This section explains why coding recommendations are different for Altera 4-input LUT devices and 6-input LUT devices.

Architectures with 4-Input LUTs in Logic Elements

Architectures such as Stratix devices and the Cyclone series of devices contain 4-input LUTs as the standard combinational structure in the LE.

If your design can tolerate pipelining, the fastest way to add three numbers `A`, `B`, and `C` in devices that use 4-input lookup tables is to add `A + B`, register the output, and then add the registered output to `C`. Adding `A + B` takes one level of logic (one bit is added in one LE), so this runs at full clock speed. This can be extended to as many numbers as desired.

Example 13–49 shows five numbers `A`, `B`, `C`, `D`, and `E` are added. Adding five numbers in devices that use 4-input lookup tables requires four adders and three levels of registers for a total of 64 LEs (for 16-bit numbers).

Example 13–49. Verilog HDL Pipelined Binary Tree

```
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 6-Input LUTs in Adaptive Logic Modules

High-performance Altera device families use a 6-input LUT in their basic logic structure, so these devices benefit from a different coding style from the previous example presented for 4-input LUTs. Specifically, in these devices, ALMs can simultaneously add three bits. Therefore, the tree in Example 13–49 must be two levels deep and contain just two add-by-three inputs instead of four add-by-two inputs.

Although the code in the previous example compiles successfully for 6-input LUT devices, the code is inefficient and does not take advantage of the 6-input adaptive ALUT. By restructuring the tree as a ternary tree, the design becomes much more efficient, significantly improving density utilization. Therefore, when you are targeting with ALUTs and ALMs, large pipelined binary adder trees designed for 4-input LUT architectures should be rewritten to take advantage of the advanced device architecture.

Example 13–50 uses just 32 ALUTs in a Stratix II device—more than a 4:1 advantage over the number of LUTs in the prior example implemented in a Stratix device.

You cannot pack a LAB full when using this type of coding style because of the number of LAB inputs. However, in a typical design, the Quartus II Fitter can pack other logic into each LAB to take advantage of the unused ALMs.

Example 13–50. Verilog HDL Pipelined Ternary Tree

```
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

   // 3-bit additions
   assign sum1 = a + b + c;
   assign sum2 = sumreg1 + d + e;
   assign out = sumreg2;
endmodule
```

These examples show pipelined adders, but partitioning your addition operations can help you achieve better results in nonpipelined adders as well. If your design is not pipelined, a ternary tree provides much better performance than a binary tree. For example, depending on your synthesis tool, the HDL code `sum = (A + B + C) + (D + E)` is more likely to create the optimal implementation of a 3-input adder for `A + B + C` followed by a 3-input adder for `sum1 + D + E` than the code without the parentheses. If you do not add the parentheses, the synthesis tool may partition the addition in a way that is not optimal for the architecture.

State Machines

Synthesis tools can recognize and encode Verilog HDL and VHDL state machines during synthesis. This section presents guidelines to ensure the best results when you use state machines. Ensuring that your synthesis tool recognizes a piece of code as a state machine allows the tool to recode the state variables to improve the quality of results, and allows the tool to use the known properties of state machines to optimize other parts of the design. When synthesis recognizes a state machine, it is often able to improve the design area and performance.

To achieve the best results on average, synthesis tools often use one-hot encoding for FPGA devices and minimal-bit encoding for CPLD devices, although the choice of implementation can vary for different state machines and different devices. Refer to your synthesis tool documentation for specific ways to control the manner in which state machines are encoded.

For information about state machine encoding in Quartus II integrated synthesis, refer to the *State Machine Processing* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

To ensure proper recognition and inference of state machines and to improve the quality of results, Altera recommends that you observe the following guidelines, which apply to both Verilog HDL and VHDL:

- Assign default values to outputs derived from the state machine so that synthesis does not generate unwanted latches.
- 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 cause the output logic of the state machine to use this value.
- Use a simple asynchronous or synchronous reset to ensure a defined power-up state. If your state machine design contains more elaborate reset logic, such as both an asynchronous reset and an asynchronous load, the Quartus II software generates regular logic rather than inferring a state machine.

If a state machine enters an illegal state due to a problem with the device, the design likely ceases to function correctly until the next reset of the state machine. Synthesis tools do not provide for this situation by default. The same issue applies to any other registers if there is some kind of fault in the system. A `default` or `when others` clause does not affect this operation, assuming that your design never deliberately enters this state. Synthesis tools remove any logic generated by a default state if it is not reachable by normal state machine operation.

Many synthesis tools (including Quartus II integrated synthesis) have an option to implement a safe state machine. The software inserts extra logic to detect an illegal state and force the state machine's transition to the reset state. It is commonly used when the state machine can enter an illegal state. The most common cause of this situation is a state machine that has control inputs that come from another clock domain, such as the control logic for a dual-clock FIFO.

This option protects only state machines by forcing them into the reset state. All other registers in the design are not protected this way. If the design has asynchronous inputs, Altera recommends using a synchronization register chain instead of relying on the safe state machine option.

For additional information about tool-specific options for implementing state machines, refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook.*

The following two sections, "Verilog HDL State Machines" and "VHDL State Machines" on page 13–66, describe additional language-specific guidelines and coding examples.

Verilog HDL State Machines

To ensure proper recognition and inference of Verilog HDL state machines, observe the following additional Verilog HDL guidelines. Some of these guidelines may be specific to Quartus II integrated synthesis. Refer to your synthesis tool documentation for specific coding recommendations.

If the state machine is not recognized and inferred by the synthesis software (such as Quartus II integrated synthesis), the state machine is implemented as regular logic gates and registers, and the state machine is not listed as a state machine in the **Analysis & Synthesis** section of the Quartus II Compilation Report. In this case, the software does not perform any of the optimizations that are specific to state machines.

- If you are using the SystemVerilog standard, use enumerated types to describe state machines. For more information, refer too "SystemVerilog State Machine Coding Example" on page 13–65.
- Represent the states in a state machine with the parameter data types in Verilog-1995 and Verilog-2001, and use the parameters to make state assignments. For more information, refer too"Verilog-2001 State Machine Coding Example" on page 13–63. This parameter implementation makes the state machine easier to read and reduces the risk of errors during coding.

  Altera recommends against the direct use of integer values for state variables, such as `next_state <= 0`. However, using an integer does not prevent inference in the Quartus II software.

- No state machine is inferred in the Quartus II software if the state transition logic uses arithmetic similar to that in the following example:

```
case (state)
   0: begin
      if (ena) next_state <= state + 2;
      else next_state <= state + 1;
   end
   1: begin
   ...
endcase
```

- No state machine is inferred in the Quartus II software if the state variable is an output.
- No state machine is inferred in the Quartus II software for signed variables.

Verilog-2001 State Machine Coding Example

The following module `verilog_fsm` is an example of a typical Verilog HDL state machine implementation (Example 13–51).

This state machine has five states. The asynchronous reset sets the variable `state` to `state_0`. The sum of `in_1` and `in_2` is an output of the state machine in `state_1` and `state_2`. The difference (`in_1` – `in_2`) is also used in `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` store the sum and the difference of `in_1` and `in_2`. Using these temporary variables in the various states of the state machine ensures proper resource sharing between the mutually exclusive states.

Example 13–51. Verilog-2001 State Machine

```
module verilog_fsm (clk, reset, in_1, in_2, out);
    input clk, reset;
    input [3:0] in_1, in_2;
    output [4:0] out;
    parameter state_0 = 3'b000;
    parameter state_1 = 3'b001;
    parameter state_2 = 3'b010;
    parameter state_3 = 3'b011;
    parameter state_4 = 3'b100;

    reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
    reg [2:0] state, next_state;

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            state <= state_0;
        else
            state <= next_state;
    end
    always @ (*)
    begin
        tmp_out_0 = in_1 + in_2;
        tmp_out_1 = in_1 - in_2;
        case (state)
            state_0: begin
               tmp_out_2 = in_1 + 5'b00001;
               next_state = state_1;
            end
            state_1: begin
                if (in_1 < in_2) begin
                    next_state = state_2;
                    tmp_out_2 = tmp_out_0;
                end
                else begin
                    next_state = state_3;
                    tmp_out_2 = tmp_out_1;
                end
            end
            state_2: begin
                tmp_out_2 = tmp_out_0 - 5'b00001;
                next_state = state_3;
            end
            state_3: begin
                tmp_out_2 = tmp_out_1 + 5'b00001;
                next_state = state_0;
            end
            state_4:begin
                tmp_out_2 = in_2 + 5'b00001;
                next_state = state_0;
            end
            default:begin
                tmp_out_2 = 5'b00000;
                next_state = state_0;
            end
        endcase
    end
    assign out = tmp_out_2;
endmodule
```

An equivalent implementation of this state machine can be achieved by using `‘define` instead of the `parameter` data type, as follows:

```
‘define state_0 3'b000
‘define state_1 3'b001
‘define state_2 3'b010
‘define state_3 3'b011
‘define state_4 3'b100
```

In this case, the `state` and `next_state` assignments are assigned a ‘`state_x` instead of a `state_x`, for example:

```
next_state <= ‘state_3;
```

Although the `‘define` construct is supported, Altera strongly recommends the use of the `parameter` data type because doing so preserves the state names throughout synthesis.

SystemVerilog State Machine Coding Example

The module `enum_fsm` in Example 13–52 is an example of a SystemVerilog state machine implementation that uses enumerated types. Altera recommends using this coding style to describe state machines in SystemVerilog.

In Quartus II integrated synthesis, the enumerated type that defines the states for the state machine must be of an unsigned integer type as in Example 13–52. If you do not specify the enumerated type as `int unsigned`, a signed `int` type is used by default. In this case, the Quartus II integrated synthesis synthesizes the design, but does not infer or optimize the logic as a state machine.

Example 13–52. SystemVerilog State Machine Using Enumerated Types

```
module enum_fsm (input clk, reset, input int data[3:0], output int o);

enum int unsigned { S0 = 0, S1 = 2, S2 = 4, S3 = 8 } state, next_state;

always_comb begin : next_state_logic
      next_state = S0;
      case(state)
        S0: next_state = S1;
        S1: next_state = S2;
        S2: next_state = S3;
        S3: next_state = S3;
      endcase
end

always_comb begin
      case(state)
         S0: o = data[3];
         S1: o = data[2];
         S2: o = data[1];
         S3: o = data[0];
      endcase
end

always_ff@(posedge clk or negedge reset) begin
      if(~reset)
         state <= S0;
      else
         state <= next_state;
end
endmodule
```

VHDL State Machines

To ensure proper recognition and inference of VHDL state machines, represent the states in a state machine with enumerated types and use the corresponding types to make state assignments. This implementation makes the state machine easier to read and reduces the risk of errors during coding. If the state is not represented by an enumerated type, synthesis software (such as Quartus II integrated synthesis) does not recognize the state machine. Instead, the state machine is implemented as regular logic gates and registers and the state machine is not listed as a state machine in the **Analysis & Synthesis** section of the Quartus II Compilation Report. In this case, the software does not perform any of the optimizations that are specific to state machines.

VHDL State Machine Coding Example

The following entity, `vhd1_fsm`, is an example of a typical VHDL state machine implementation (Example 13–53).

This state machine has five states. The asynchronous reset sets the variable `state` to `state_0`. The sum of `in1` and `in2` is an output of the state machine in `state_1` and `state_2`. The difference (`in1` - `in2`) is also used in `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` store the sum and the difference of `in1` and `in2`. Using these temporary variables in the various states of the state machine ensures proper resource sharing between the mutually exclusive states.

Example 13–53. VHDL State Machine

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY vhdl_fsm IS
   PORT(
      clk: IN STD_LOGIC;
      reset: IN STD_LOGIC;
      in1: IN UNSIGNED(4 downto 0);
      in2: IN UNSIGNED(4 downto 0);
      out_1: OUT UNSIGNED(4 downto 0)
      );
END vhdl_fsm;
ARCHITECTURE rtl OF vhdl_fsm IS
   TYPE Tstate IS (state_0, state_1, state_2, state_3, state_4);
   SIGNAL state: Tstate;
   SIGNAL next_state: Tstate;
BEGIN
   PROCESS(clk, reset)
   BEGIN
      IF reset = '1' THEN
            state <=state_0;
      ELSIF rising_edge(clk) THEN
            state <= next_state;
      END IF;
   END PROCESS;
PROCESS (state, in1, in2)
      VARIABLE tmp_out_0: UNSIGNED (4 downto 0);
      VARIABLE tmp_out_1: UNSIGNED (4 downto 0);
   BEGIN
      tmp_out_0 := in1 + in2;
      tmp_out_1 := in1 - in2;
      CASE state IS
         WHEN state_0 =>
            out_1 <= in1;
            next_state <= state_1;
         WHEN state_1 =>
            IF (in1 < in2) then
               next_state <= state_2;
               out_1 <= tmp_out_0;
            ELSE
               next_state <= state_3;
               out_1 <= tmp_out_1;
            END IF;
         WHEN state_2 =>
            IF (in1 < "0100") then
               out_1 <= tmp_out_0;
            ELSE
               out_1 <= tmp_out_1;
            END IF;
               next_state <= state_3;
         WHEN state_3 =>
               out_1 <= "11111";
               next_state <= state_4;
         WHEN state_4 =>
               out_1 <= in2;
               next_state <= state_0;
         WHEN OTHERS =>
               out_1 <= "00000";
               next_state <= state_0;
      END CASE;
   END PROCESS;
END rtl;
```

Multiplexers

Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexer logic, you ensure the most efficient implementation in your Altera device. This section addresses common problems and provides design guidelines to achieve optimal resource utilization for multiplexer designs. The section also describes various types of multiplexers, and how they are implemented.

For more information, refer to the *Advanced Synthesis Cookbook*.

Quartus II Software Option for Multiplexer Restructuring

Quartus II integrated synthesis provides the **Restructure Multiplexers** logic option that extracts and optimizes buses of multiplexers during synthesis. The default setting **Auto** for this option uses the optimization when it is most likely to benefit the optimization targets for your design. You can turn the option on or off specifically to have more control over its use.

For details, refer to the *Restructure Multiplexers* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Even with this Quartus II-specific option turned on, it is beneficial to understand how your coding style can be interpreted by your synthesis tool, and avoid the situations that can cause problems in your design.

Multiplexer Types

This section addresses how multiplexers are created from various types of HDL code. `CASE` statements, `IF` statements, and state machines are all common sources of multiplexer logic in designs. These HDL structures create different types of multiplexers, including binary multiplexers, selector multiplexers, and priority multiplexers. Understanding how multiplexers are created from HDL code, and how they might be implemented during synthesis, is the first step toward optimizing multiplexer structures for best results.

Binary Multiplexers

Binary multiplexers select inputs based on binary-encoded selection bits. Example 13–54 shows Verilog HDL code for two ways to describe a simple 4:1 binary multiplexer.

Example 13–54. Verilog HDL Binary-Encoded Multiplexers

```
case (sel)
   2'b00: z = a;
   2'b01: z = b;
   2'b10: z = c;
   2'b11: z = d;
endcase
```

Stratix series devices starting with the Stratix II device family feature 6-input look up tables (LUTs) which are perfectly suited for 4:1 multiplexer building blocks (4 data and 2 select inputs). The extended input mode facilitates implementing 8:1 blocks, and the fractured mode handles residual 2:1 multiplexer pairs. For device families using 4-input LUTs, such as the Cyclone series and Stratix devices, the 4:1 binary multiplexer is efficiently implemented by using two 4-input LUTs. Larger binary multiplexers are decomposed by the synthesis tool into 4:1 multiplexer blocks, possibly with a residual 2:1 multiplexer at the head.

Selector Multiplexers

Selector multiplexers have a separate select line for each data input. The select lines for the multiplexer are one-hot encoded. Example 13–55 shows a simple Verilog HDL code example describing a one-hot selector multiplexer.

Example 13–55. Verilog HDL One-Hot-Encoded Case Statement

```
case (sel)
   4'b0001: z = a;
   4'b0010: z = b;
   4'b0100: z = c;
   4'b1000: z = d;
   default: z = 1'bx;
endcase
```

Selector multiplexers are commonly built as a tree of AND and OR gates. An N-input selector multiplexer of this structure is slightly less efficient in implementation than a binary multiplexer. However, in many cases the select signal is the output of a decoder, in which case Quartus II Synthesis will try to combine the selector and decoder into a binary multiplexer.

Priority Multiplexers

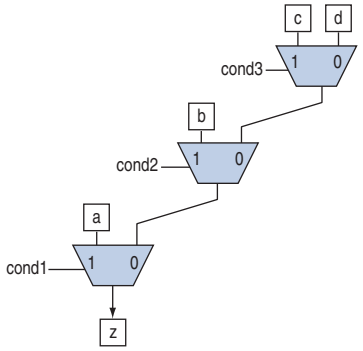
In priority multiplexers, the select logic implies a priority. The options to select the correct item must be checked in a specific order based on signal priority. These structures commonly are created from `IF`, `ELSE`, `WHEN`, `SELECT`, and `?:` statements in VHDL or Verilog HDL. The example VHDL code in Example 13–56 probably results in the schematic implementation illustrated in Figure 13–4.

Example 13–56. VHDL IF Statement Implying Priority

```
IF cond1 THEN z <= a;
ELSIF cond2 THEN z <= b;
ELSIF cond3 THEN z <= c;
ELSE z <= d;
END IF;
```

The multiplexers in Figure 13–4 form a chain, evaluating each condition or select bit sequentially.

Figure 13–4. Priority Multiplexer Implementation of an IF Statement



Depending on the number of multiplexers in the chain, the timing delay through this chain can become large, especially for device families with 4-input LUTs.

To improve the timing delay through the multiplexer, avoid priority multiplexers if priority is not required. If the order of the choices is not important to the design, use a `CASE` statement to implement a binary or selector multiplexer instead of a priority multiplexer. If delay through the structure is important in a multiplexed design requiring priority, consider recoding the design to reduce the number of logic levels to minimize delay, especially along your critical paths.

Implicit Defaults in If Statements

The `IF` statements in Verilog HDL and VHDL can be a convenient way to specify conditions that do not easily lend themselves to a `CASE`-type approach. However, using `IF` statements can result in complicated multiplexer trees that are not easy for synthesis tools to optimize. In particular, every `IF` statement has an implicit `ELSE` condition, even when it is not specified. These implicit defaults can cause additional complexity in a multiplexed design.

There are several ways you can simplify multiplexed logic and remove unneeded defaults. The optimal method may be to recode the design so the logic takes the structure of a 4:1 `CASE` statement. Alternatively, if priority is important, you can restructure the code to reduce default cases and flatten the multiplexer. Examine whether the default "`ELSE IF`" conditions are don't care cases. You may be able to create a default `ELSE` statement to make the behavior explicit. Avoid unnecessary default conditions in your multiplexer logic to reduce the complexity and logic utilization required to implement your design.

Default or Others Case Assignment

To fully specify the cases in a `CASE` statement, include a `default` (Verilog HDL) or `OTHERS` (VHDL) assignment. This assignment is especially important in one-hot encoding schemes where many combinations of the select lines are unused. Specifying a case for the unused select line combinations gives the synthesis tool information about how to synthesize these cases, and is required by the Verilog HDL and VHDL language specifications.

Some designs do not require that the outcome in the unused cases be considered, often because designers assume these cases will not occur. For these types of designs, you can specify any value for the `default` or `OTHERS` assignment. However, be aware that the assignment value you choose can have a large effect on the logic utilization required to implement the design due to the different ways synthesis tools treat different values for the assignment, and how the synthesis tools use different speed and area optimizations.

To obtain best results, explicitly define invalid `CASE` selections with a separate `default` or `OTHERS` statement instead of combining the invalid cases with one of the defined cases.

If the value in the invalid cases is not important, specify those cases explicitly by assigning the `X` (don't care) logic value instead of choosing another value. This assignment allows your synthesis tool to perform the best area optimizations.

Cyclic Redundancy Check Functions

CRC computations are used heavily by communications protocols and storage devices to detect any corruption of data. These functions are highly effective; there is a very low probability that corrupted data can pass a 32-bit CRC check.

CRC functions typically use wide XOR gates to compare the data. The way synthesis tools flatten and factor these XOR gates to implement the logic in FPGA LUTs can greatly impact the area and performance results for the design. XOR gates have a cancellation property that creates an exceptionally large number of reasonable factoring combinations, so synthesis tools cannot always choose the best result by default.

The 6-input ALUT has a significant advantage over 4-input LUTs for these designs. When properly synthesized, CRC processing designs can run at high speeds in devices with 6-input ALUTs.

The following guidelines help you improve the quality of results for CRC designs in Altera devices.

If Performance is Important, Optimize for Speed

Synthesis tools flatten XOR gates to minimize area and depth of levels of logic. Synthesis tools such as Quartus II integrated synthesis target area optimization by default for these logic structures. Therefore, for more focus on depth reduction, set the synthesis optimization technique to speed.

Flattening for depth sometimes causes a significant increase in area.

Use Separate CRC Blocks Instead of Cascaded Stages

Some designers optimize their CRC designs to use cascaded stages (for example, four stages of 8 bits). In such designs, intermediate calculations are used as required (such as the calculations after 8, 24, or 32 bits) depending on the data width. This design is not optimal in FPGA devices. The XOR cancellations that can be performed in CRC designs mean that the function does not require all the intermediate calculations to

determine the final result. Therefore, forcing the use of intermediate calculations increases the area required to implement the function, as well as increasing the logic depth because of the cascading. It is typically better to create full separate CRC blocks for each data width that you require in the design, and then multiplex them together to choose the appropriate mode at a given time

Use Separate CRC Blocks Instead of Allowing Blocks to Merge

Synthesis tools often attempt to optimize CRC designs by sharing resources and extracting duplicates in two different CRC blocks because of the factoring options in the XOR logic. As addressed previously, the CRC logic allows significant reductions, but this works best when each CRC function is optimized separately. Check for duplicate extraction behavior if you have different CRC functions that are driven by common data signals or that feed the same destination signals.

If you are having problems with the quality of results and you see that two CRC functions are sharing logic, ensure that the blocks are synthesized independently using one of the following methods:

- Define each CRC block as a separate design partition in an incremental compilation design flow.

  For details, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

- Synthesize each CRC block as a separate project in your third-party synthesis tool and then write a separate Verilog Quartus Mapping (**.vqm**) or EDIF netlist file for each.

Take Advantage of Latency if Available

If your design can use more than one cycle to implement the CRC functionality, adding registers and retiming the design can help reduce area, improve performance, and reduce power utilization. If your synthesis tool offers a retiming feature (such as the Quartus II software **Perform gate-level register retiming** option), you can insert an extra bank of registers at the input and allow the retiming feature to move the registers for better results. You can also build the CRC unit half as wide and alternate between halves of the data in each clock cycle.

Save Power by Disabling CRC Blocks When Not in Use

CRC designs are heavy consumers of dynamic power because the logic toggles whenever there is a change in the design. To save power, use clock enables to disable the CRC function for every clock cycle that the logic is not required. Some designs don't check the CRC results for a few clock cycles while other logic is performed. It is valuable to disable the CRC function even for this short amount of time.

Use the Device Synchronous Load (sload) Signal to Initialize

The data in many CRC designs must be initialized to 1's before operation. If your target device supports the use of the `sload` signal, you should use it to set all the registers in your design to 1's before operation. To enable use of the `sload` signal, follow the coding guidelines presented in "Secondary Register Control Signals Such as Clear and Clock Enable" on page 13–46. You can check the register equations in the Chip Planner to ensure that the signal was used as expected.

If you must force a register implementation using an `sload` signal, you can use low-level device primitives as described in the *Designing with Low-Level Primitives User Guide*.

Comparators

Synthesis software, including Quartus II integrated synthesis, uses device and context-specific implementation rules for comparators (`<`, `>`, or `==`) and selects the best one for your design. This section provides some information about the different types of implementations available and provides suggestions on how you can code your design to encourage a specific implementation.

The `==` comparator is implemented in general logic cells. The `<` comparison can be implemented using the carry chain or general logic cells. In devices with 6-input ALUTs, the carry chain is capable of comparing up to three bits per cell. In devices with 4-input LUTs, the capacity is one bit of comparison per cell, which is similar to an add/subtract chain. The carry chain implementation tends to be faster than the general logic on standalone benchmark test cases, but can result in lower performance when it is part of a larger design due to the increased restriction on the Fitter. The area requirement is similar for most input patterns. The synthesis software selects an appropriate implementation based on the input pattern.

If you are using Quartus II integrated synthesis, you can guide the synthesis by using specific coding styles. To select a carry chain implementation explicitly, rephrase your comparison in terms of addition. As a simple example, the following coding style allows the synthesis tool to select the implementation, which is most likely using general logic cells in modern device families:

```
wire [6:0] a,b;
wire alb = a<b;
```

In the following coding style, the synthesis tool uses a carry chain (except for a few cases, such as when the chain is very short or the signals `a` and `b` minimize to the same signal):

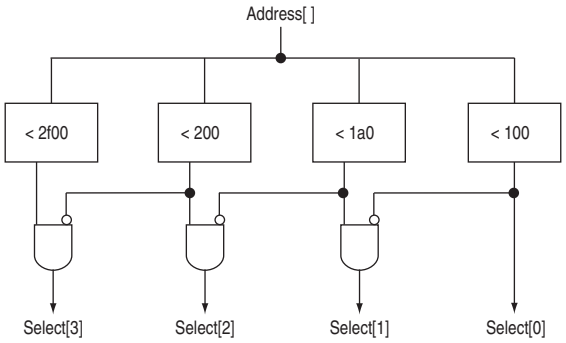
```
wire [6:0] a,b;
wire [7:0] tmp = a - b;
wire alb = tmp[7]
```

This second coding style uses the top bit of the `tmp` signal, which is 1 in twos complement logic if *a* is less than *b*, because the subtraction *a* – *b* results in a negative number.

If you have any information about the range of the input, you have "don't care" values that you can use to optimize the design. Because this information is not available to the synthesis tool, you can often reduce the device area required to implement the comparator with specific hand implementation of the logic.

You can also check whether a bus value is within a constant range with a small amount of logic area by using the logic structure in Figure 13–5. This type of logic occurs frequently in address decoders.

Figure 13–5. Example Logic Structure for Using Comparators to Check a Bus Value Range



Counters

Implementing counters in HDL code is easy; they are implemented with an adder followed by registers. Remember that the register control signals, such as enable (`ena`), synchronous clear (`sclr`), and synchronous load (`sload`), are available. For the best area utilization, ensure that the up/down control or controls are expressed in terms of one addition instead of two separate addition operators.

If you use the following coding style, your synthesis tool may implement two separate carry chains for addition (if it doesn't detect the issue and optimize the logic):

```
out <= count_up ? out + 1 : out - 1;
```

The following coding style requires only one adder along with some other logic:

```
out <= out + (count_up ? 1 : -1);
```

In this case, the coding style better matches the device hardware because there is only one carry chain adder, and the –1 constant logic is implemented in the LUT in front of the adder without adding extra area utilization.

Designing with Low-Level Primitives

Low-level HDL design is the practice of using low-level primitives and assignments to dictate a particular hardware implementation for a piece of logic. Low-level primitives are small architectural building blocks that assist you in creating your design. With the Quartus II software, you can use low-level HDL design techniques to force a specific hardware implementation that can help you achieve better resource utilization or faster timing results.

Using low-level primitives is an advanced technique to help with specific design challenges, and is optional in the Altera design flow. For many designs, synthesizing generic HDL source code and Altera megafunctions gives you the best results.

Low-level primitives allow you to use the following types of coding techniques:

- Instantiate the logic cell or `LCELL` primitive to prevent Quartus II integrated synthesis from performing optimizations across a logic cell

- Create carry and cascade chains using `CARRY`, `CARRY_SUM`, and `CASCADE` primitives
- Instantiate registers with specific control signals using `DFF` primitives
- Specify the creation of LUT functions by identifying the LUT boundaries
- Use I/O buffers to specify I/O standards, current strengths, and other I/O assignments
- Use I/O buffers to specify differential pin names in your HDL code, instead of using the automatically-generated negative pin name for each pair

For details about and examples of using these types of assignments, refer to the *Designing with Low-Level Primitives User Guide*.

Conclusion

Because coding style and megafunction implementation can have such a large effect on your design performance, it is important to match the coding style to the device architecture from the very beginning of the design process. To improve design performance and area utilization, take advantage of advanced device features, such as memory and DSP blocks, as well as the logic architecture of the targeted Altera device by following the coding recommendations presented in this chapter.

For additional optimization recommendations, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Document Revision History

Table 13–3 shows the revision history for this document.

Table 13–3. Document Revision History (Part 1 of 2)

Date	Version	Changes
November 2013	13.1.0	Removed HardCopy device information.
June 2012	12.0.0	■ Revised section on inserting Altera templates. ■ Code update for Example 11-51. ■ Minor corrections and updates.
November 2011	11.1.0	■ Updated document template. ■ Minor updates and corrections.
December 2010	10.1.0	■ Changed to new document template. ■ Updated Unintentional Latch Generation content. ■ Code update for Example 11-18.
July 2010	10.0.0	■ Added support for mixed-width RAM ■ Updated support for no_rw_check for inferring RAM blocks ■ Added support for byte-enable
November 2009	9.1.0	■ Updated support for Controlling Inference and Implementation in Device RAM Blocks ■ Updated support for Shift Registers

Table 13–3. Document Revision History (Part 2 of 2)

Date	Version	Changes
March 2009	9.0.0	■ Corrected and updated several examples ■ Added support for Arria II GX devices ■ Other minor changes to chapter
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	Updates for the Quartus II software version 8.0 release, including: ■ Added information to "RAM Functions—Inferring ALTSYNCRAM and ALTDPRAM Megafunctions from HDL Code" on page 6–13 ■ Added information to "Avoid Unsupported Reset and Control Conditions" on page 6–14 ■ Added information to "Check Read-During-Write Behavior" on page 6–16 ■ Added two new examples to "ROM Functions—Inferring ALTSYNCRAM and LPM_ROM Megafunctions from HDL Code" on page 6–28: Example 6–24 and Example 6–25 ■ Added new section: "Clock Multiplexing" on page 6–46 ■ Added hyperlinks to references within the chapter ■ Minor editorial updates

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.