Improving Performance in FLEX 10K Devices with the Synplify Software

October 1998, ver. 1.0 Application Note 101

Introduction

As the demand for improved performance increases, you must construct your designs for maximum logic optimization. Achieving better performance in FLEX® 10K devices is possible with VHDL and Verilog HDL coding techniques, Synplify software constraints, and MAX+PLUS® II software options. Using these tools can help you streamline your design, optimize it for FLEX 10K devices, and increase the speed of an already high-performance programmable logic family. This application note documents techniques for improving FLEX 10K device performance using the Altera®/Synplicity design flow.

☞ You should have a basic understanding of the FLEX 10K family architecture, the Altera MAX+PLUS II software, and the Synplicity Synplify software to use this document effectively.

This application note provides information on the following topics:

A-AN-101-01

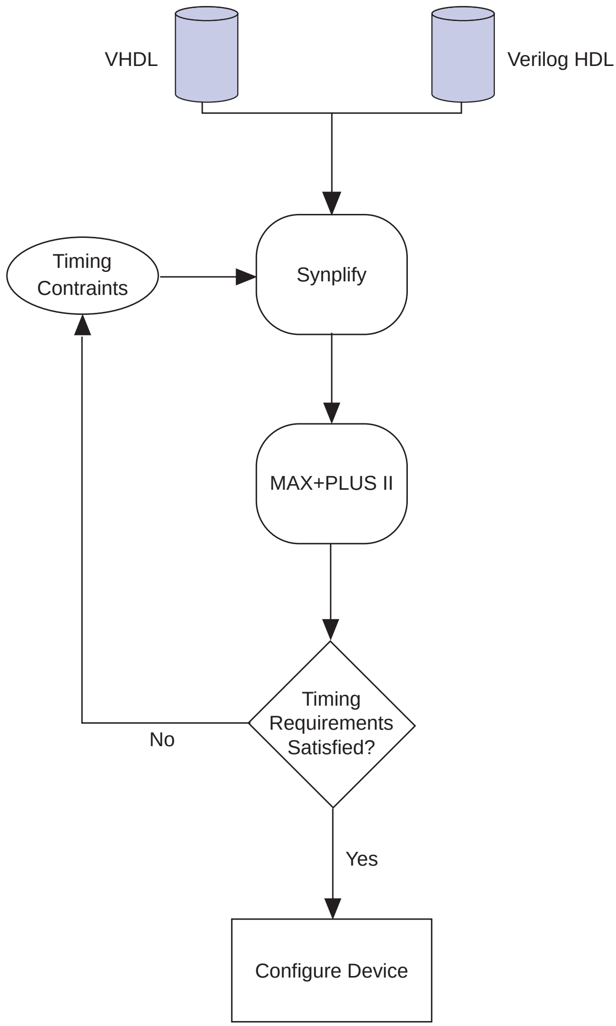
Design Flow

The Altera/Synplicity design flow begins by creating a hardware description language (HDL) design for synthesis in the Synplify software. The guidelines for creating an HDL design are covered in "Effective HDL Design Techniques in the Synplify Software" and "Synplify Design Techniques for FLEX 10K Devices." After successful creation of the HDL description of the design, the design is synthesized in the Synplify software, and an electronic design interchange format (EDIF) file (**.edf**) is generated to be imported into the MAX+PLUS II software. The design flow ends with performance evaluation and, if appropriate, implementation in a FLEX 10K device.

☞ Most design techniques have a timing versus area trade-off. Thus, techniques that yield an improvement in area may reduce performance. Additionally, techniques such as logic replication can improve the overall performance of a design, but may add logic.

Figure 1 shows the recommended design flow between the Synplify software and the MAX+PLUS II software.

Figure 1. Recommended Design Flow



Effective HDL Design Techniques in the Synplify Software

By using effective HDL design techniques in the Synplify software, you can streamline your designs, reduce and optimize logic, reduce logic delay, and improve overall performance. The following sections describe how to achieve these results including:

- Hierarchy
- Combinatorial Logic
- Priority-Encoded If Statements
- "Don't Care" Conditions for Logic Optimization
- Sequential Logic
- Gated Clocks
- State Machines
- Implementing State Machines in Embedded Array Blocks (EABs)

Hierarchy

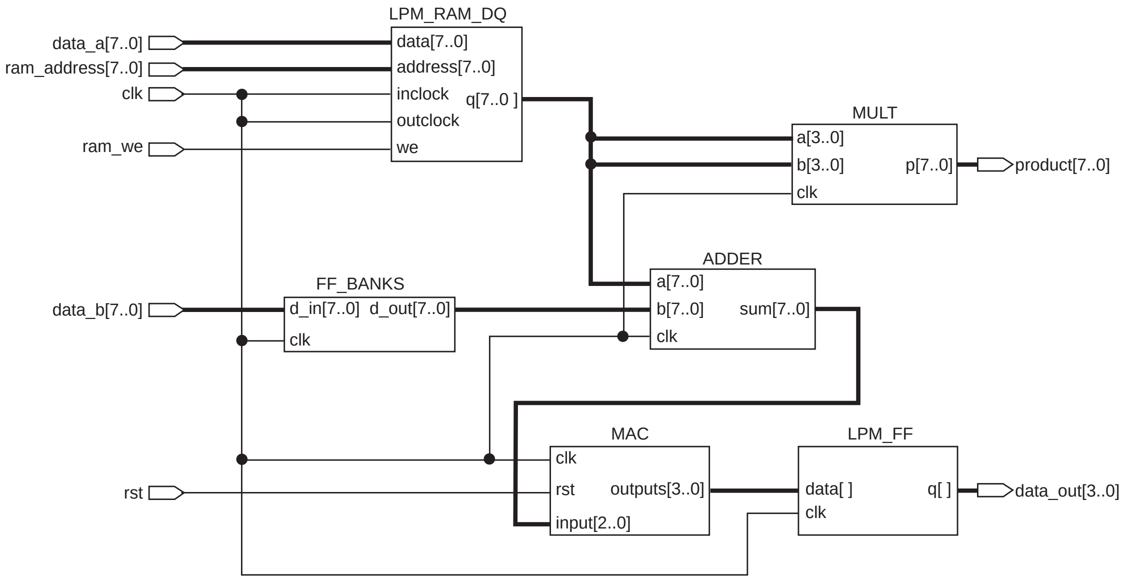
The functionality of many designs is too complex to implement in a single design file. The Synplify software allows you to create multiple design files and link the files into a hierarchy. You can build up the design through standard VHDL or Verilog HDL instantiations, and simulate and optimize subdesigns individually rather than optimizing the entire design. When partitioning your design, use the following guidelines:

- Partition the design at functional boundaries. Block diagrams or high-level schematics help create natural boundaries, as shown in Figure 2. Data paths, tri-state signals, state machines, register blocks, large macrofunctions, memory elements, control blocks, and reusable megafunctions all form natural boundaries.

- Partition the design to minimize I/O connections. Too many I/O ports complicate the design and debug processes.

- Register the block outputs, if possible.

Figure 2. Hierarchical Design



Combinatorial Logic

Logic is described as combinatorial if outputs at a specified time are a function of the inputs at that time only, regardless of the previous state of the circuit. Examples of combinatorial logic functions are decoders, multiplexers, and adders.

When applied to combinatorial logic, the techniques described in the following sections help optimize the performance results obtained during Synplify synthesis.

Latches

FLEX devices have registers, rather than latches, built into the silicon. Therefore, designing with latches generates more logic and lower performance than designing with registers. For example, the MAX+PLUS II software uses two logic elements (LEs) in a FLEX device to create a latch.

When designing combinatorial logic, you should avoid unintentionally creating a latch due to your HDL design style. For example, when Case or If Statements do not cover all possible conditions of the inputs, combinatorial feedback can generate latches.

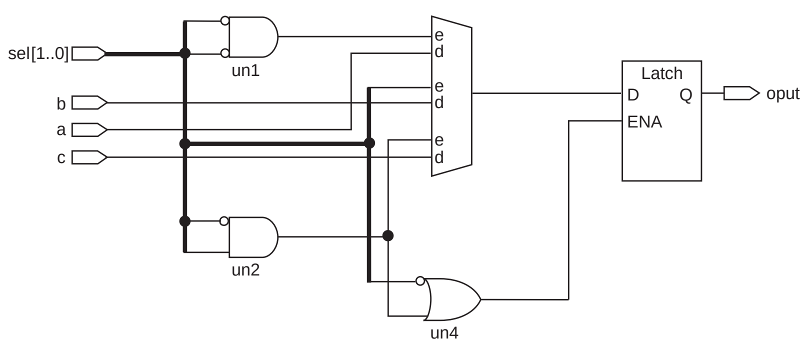
Figure 3 shows sample VHDL code that generates a latch.

Figure 3. Sample VHDL Code Unintentionally Generating a Latch

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bad IS
PORT    (a,b,c  : IN STD_LOGIC;
         sel    : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
         oput   : OUT STD_LOGIC);
END bad;

ARCHITECTURE behave OF bad IS
BEGIN
     PROCESS (a,b,c,sel)
     BEGIN
          IF sel = "00" THEN
               oput <= a;
          ELSIF sel = "01" THEN
               oput <= b;
          ELSIF sel = "10" THEN
               oput <= c;
          END IF;
     END PROCESS;
END behave;
```

Figure 4 shows the schematic representation of the VHDL code from Figure 3.

Figure 4. Schematic Representation of Code Unintentionally Generating a Latch



A latch is generated when the final ELSE clause or WHEN OTHERS clause is omitted from an If or Case Statement, respectively.

"Don't care" assignments on the default condition tend to generate the best logic optimization in the Synplify software.

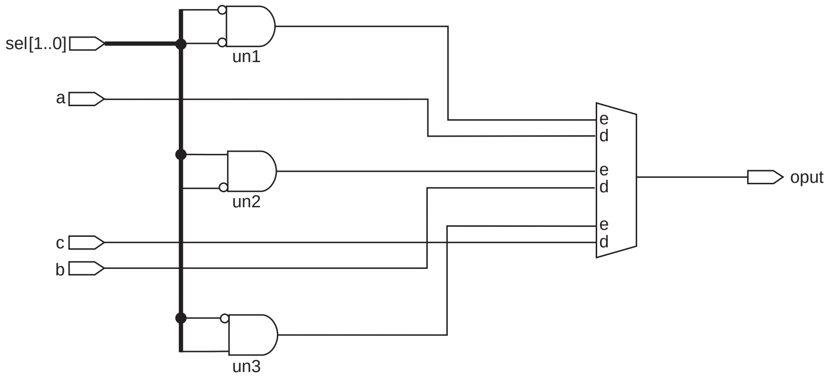
Figure 5 shows sample VHDL code that prevents the unintentional creation of a latch.

Figure 5. Sample VHDL Code Preventing Unintentional Latch Creation

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY good IS
PORT   (a,b,c  : IN STD_LOGIC;
        sel    : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        oput   : OUT STD_LOGIC);
END good;

ARCHITECTURE behave OF good IS
BEGIN
     PROCESS (a,b,c,sel)
     BEGIN
          IF sel = "00" THEN
               oput <= a;
          ELSIF sel = "01" THEN
               oput <= b;
          ELSIF sel = "10" THEN
               oput <= c;
          ELSE
               oput <= 'x';       -- removes latch
          END IF;
     END PROCESS;
END behave;
```

Figure 6 shows the schematic representation of the VHDL code from Figure 5.

Figure 6. Schematic Representation of Code Preventing Latch Creation



Priority-Encoded If Statements

To reduce the propagation delay of critical-path signals in a design, you can use If Statements to perform priority encoding. The Verilog HDL code in Figure 7 shows priority encoding using If Statements. This example illustrates good design practice if `sel1` is a late-arriving signal in the critical path. In this case, `sel1` has the highest priority.

Figure 7. Verilog HDL for Priority-Encoded If Statement

```
module priority (a,b,c,d,sel1,sel2,sel3,sel4,oput);
input a,b,c,d,sel1,sel2,sel3,sel4;
output oput;
always @(a or b or c or d or sel1 or sel2 or sel3 or sel4)
begin
    oput = 1'b0;
    if (sel1)
          oput = a;
    else if (sel2)
          oput = b;
    else if (sel3)
          oput = c;
    else if (sel4)
          oput = d;
end
endmodule
```

“Don’t Care” Conditions for Logic Optimization

The Synplify software generally treats unknowns as “don't care” conditions to optimize logic. Within a design, you can assign the default case value to “don't care” instead of to a logic value to give the best logic optimization.

☞ All “don’t care” conditions should be verified in simulation.

Figure 8 shows VHDL code that assigns a logic value for the default case, thereby creating a non-optional implementation.

Figure 8. Assigning a Logic Value for the Default Case in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY value IS
   PORT (     data : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
              sel  : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
              dout : OUT STD_LOGIC);
END value;

ARCHITECTURE behave OF value IS
BEGIN
   PROCESS(sel,data)
   BEGIN
      CASE sel IS
              WHEN "00000" => dout <= data(0);
              WHEN "00001" => dout <= data(1);
              WHEN "00010" => dout <= data(2);
              WHEN "00011" => dout <= data(3);
              WHEN "00100" => dout <= data(4);
              WHEN "00101" => dout <= data(5);
              WHEN "00110" => dout <= data(6);
              WHEN "00111" => dout <= data(7);
              WHEN "01000" => dout <= data(8);
              WHEN "01001" => dout <= data(9);
              WHEN "01010" => dout <= data(10);
              WHEN "01011" => dout <= data(11);
              WHEN "01100" => dout <= data(12);
              WHEN "01101" => dout <= data(13);
              WHEN "01110" => dout <= data(14);
              WHEN "01111" => dout <= data(15);
              WHEN OTHERS  => dout <= '0';       -- assigned a logic value.
      END CASE;
   END PROCESS;
END behave;
```

The design style shown in Figure 8 fails to produce the optimum implementation, and has a propagation delay (t_{PD}) of 19.6 ns. By implementing a "don't care" condition for the default case, a t_{PD} of 18.3 ns is achieved (see Figure 9).

Figure 9. Assigning a "Don't Care" Value for the Default Case in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY no_care IS
   PORT (     data  :IN STD_LOGIC_VECTOR (15 DOWNTO 0);
              sel   :IN STD_LOGIC_VECTOR(4 DOWNTO 0);
              dout  :OUT STD_LOGIC);
END no_care;

ARCHITECTURE behave no_care IS
BEGIN
   PROCESS(sel,data)
   BEGIN
      CASE sel IS
             WHEN "00000" => dout <= data(0);
             WHEN "00001" => dout <= data(1);
             WHEN "00010" => dout <= data(2);
             WHEN "00011" => dout <= data(3);
             WHEN "00100" => dout <= data(4);
             WHEN "00101" => dout <= data(5);
             WHEN "00110" => dout <= data(6);
             WHEN "00111" => dout <= data(7);
             WHEN "01000" => dout <= data(8);
             WHEN "01001" => dout <= data(9);
             WHEN "01010" => dout <= data(10);
             WHEN "01011" => dout <= data(11);
             WHEN "01100" => dout <= data(12);
             WHEN "01101" => dout <= data(13);
             WHEN "01110" => dout <= data(14);
             WHEN "01111" => dout <= data(15);
             WHEN OTHERS  => dout <= 'x';   -- assigned a "don't care" value.
      END CASE;
   END PROCESS;
END behave;
```

Sequential Logic

Logic is sequential if the outputs at a specified time are a function of the inputs at that time and at all preceding times. All sequential circuits must include one or more registers (i.e., flipflops). Each FLEX LE contains a D-type flipflop; thus, pipelining allocates no additional resources. If you want to create a break in the logic or to store a value within your design, extra LEs or routing resources are not allocated.

You should avoid unintentionally creating a feedback multiplexer. A feedback multiplexer is created if all possible input conditions are not assigned when using If Statements. Figure 10 shows VHDL code that generates a feedback multiplexer.

Figure 10. VHDL Code that Generates a Feedback Multiplexer

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY seq IS
  PORT( a,b,c,d,clk,rst : IN STD_LOGIC;
        sel             : STD_LOGIC_VECTOR(3 DOWNTO 0);
        oput            : OUT STD_LOGIC);
END seq;

ARCHITECTURE behave OF seq IS
BEGIN
    PROCESS(clk, rst)
    BEGIN
         IF rst = '1' THEN
              oput <= '1';
         ELSIF clk='1' AND clk’EVENT THEN
              IF sel(0) = '1' THEN
                   oput <= a AND b;
              ELSIF sel(1) = '1' THEN
                   oput <= b;
              ELSIF sel(2) = '1' THEN
                   oput <= c;
              ELSIF sel(3) = '1' THEN
                   oput <= d;
              END IF;
         END IF;
    END PROCESS;
END behave;
```

When a feedback multiplexer is generated, the function shown in Figure 10 requires five LEs. To reduce the number of LEs, use the final ELSE clause to assign all states, thus removing the feedback. The implementation shown in Figure 11 requires only two LEs.

Figure 11. VHDL Code that Prevents Feedback Multiplexer Generation

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

ENTITY seq2 IS
  PORT ( a,b,c,d,clk,rst : IN STD_LOGIC;
         sel             : STD_LOGIC_VECTOR(3 DOWNTO 0);
         oput            : OUT STD_LOGIC);
END seq2;

ARCHITECTURE behave OF seq2 IS
BEGIN
    PROCESS(clk, rst)
    BEGIN
         IF rst = '1' THEN
              oput <= '1';
         ELSIF clk='1' AND clk'EVENT THEN
              IF sel(0) = '1' THEN
                   oput <= a AND b;
              ELSIF sel(1) = '1' THEN
                   oput <= b;
              ELSIF sel(2) = '1' THEN
                   oput <= c;
              ELSIF sel(3) = '1' THEN
                   oput <= d;
              ELSE
                   oput<= 'x'; -- removes feedback
                                  multiplexer
              END IF;
         END IF;
    END PROCESS;
END behave;
```

Gated Clocks

Gated clocks create logic delays and clock skew, and use additional routing resources within FLEX devices. Additionally, a limited number of clocks are available per logic array block (LAB). If possible, you should avoid using gated clocks.

If you must implement a gated clock in your design, use the `GLOBAL` primitive to place the gated clock on one of the high-fan-out internal global signals. Figure 12 shows an example that implements a gated clock using the `GLOBAL` primitive in a VHDL design.

Figure 12. Implementing a Gated Clock in VHDL

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

ENTITY gate IS
  PORT ( a,b      : IN STD_LOGIC;
         c,d      : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         oput     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END gate;

ARCHITECTURE behave OF gate IS
SIGNAL clock        : STD_LOGIC;
SIGNAL gclk         : STD_LOGIC;
SIGNAL count        : STD_LOGIC_VECTOR(3 DOWNTO 0);
ATTRIBUTE black_box : BOOLEAN;

COMPONENT GLOBAL
  PORT ( a_in  : IN STD_LOGIC;
         a_out : OUT STD_LOGIC);
END COMPONENT;
ATTRIBUTE black_box OF GLOBAL:  COMPONENT IS TRUE;

BEGIN
  clock <= a AND b;
  clk_buf:  GLOBAL PORT MAP (clock, gclk);

  PROCESS(gclk)
  BEGIN
       IF gclk='1' AND gclk'EVENT THEN
                     count <= c + d;
       END IF;
  END PROCESS;
  oput <= count;
END behave;
```

State Machines

In designs containing state machines, you should separate the state machine logic from all arithmetic functions and data paths to improve performance. Use a state machine purely as control logic.

Figure 13 shows a block diagram of a sample state machine consisting of a 32-bit counter, an 8-bit 4-to-1 multiplexer, and a state machine that provides the control logic for the other two elements in the design.

Figure 13. Sample State Machine

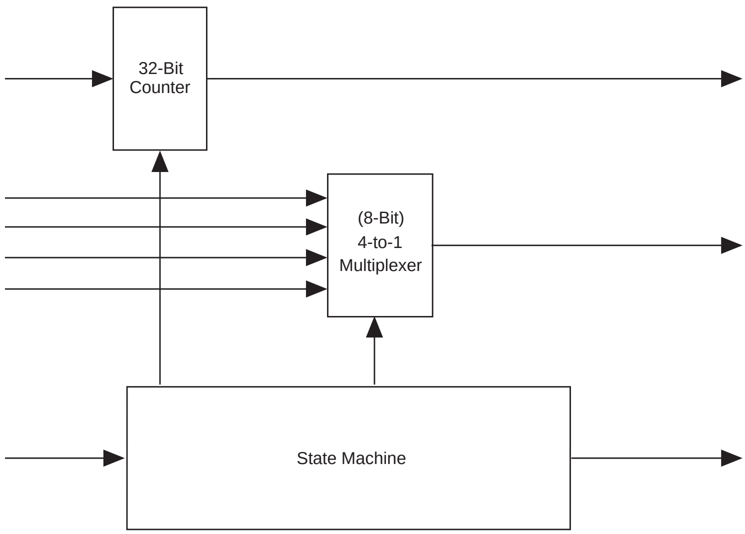


Figure 14 shows an inefficient design style because the counter and the multiplexer are incorporated into the state machine description. As a result, two counters may be inferred instead of one up/down counter. In addition, the multiplexer may have additional logic associated with its control signals.

Figure 14. Inefficient VHDL Code for a Sample State Machine (Part 1 of 4)

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

ENTITY vhdl_bad IS
 PORT (
    clk    : IN  STD_LOGIC;
```

Figure 14. Inefficient VHDL Code for a Sample State Machine (Part 2 of 4)

```
    rst    : IN  STD_LOGIC;
    muxa   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
    muxb   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
    muxc   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
    muxd   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
    c_in   : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
    sm_in  : IN  STD_LOGIC_VECTOR(4 DOWNTO 0);
    sm_out : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
    c_out  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    muxout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END vhdl_bad ;

ARCHITECTURE sm OF vhdl_bad IS

    TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, s6, s7);
    SIGNAL state: STATE_TYPE;
    SIGNAL count     : STD_LOGIC_VECTOR(31 DOWNTO 0);

BEGIN
    c_out <= count;
    PROCESS (clk, rst)
    BEGIN
      IF (rst = '1') THEN
          state <= s0;
          sm_out <= "00000";
          count  <= (OTHERS => '0');
          muxout <= (OTHERS => '0');
      ELSIF (clk’EVENT AND clk = '1') THEN
          muxout <= muxa;
          CASE state IS
            WHEN s0 =>
              IF (sm_in(4) = '1' AND sm_in(3) = '1'AND
              sm_in(1)='1' AND sm_in(0) = '1')THEN
                  state  <= s6;
                  sm_out <= "00110";
              ELSIF (sm_in(1) = '1') THEN
                  state  <= s5;
                  sm_out <= "01110";
                  count <= count + "1";
              ELSE
                  state  <= s0;
                  sm_out <= "11110";
                  muxout <= muxb;
              END IF;
            WHEN s1 =>
```

Figure 14. Inefficient VHDL Code for a Sample State Machine (Part 3 of 4)

```
  IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
  sm_in(1) ='1' AND sm_in(0) = '1') THEN
      state  <= s7;
      sm_out <= "11111";
  ELSIF (sm_in(4) = '1' AND sm_in(1) = '1') THEN
      state  <= s4;
      sm_out <= "10110";
      count <= c_in;
      muxout <= muxc;
  ELSIF (sm_in(3) = '1') THEN
      state  <= s3;
      sm_out <= "00011";
  ELSIF (sm_in(4) = '1') THEN
      state  <= s2;
      sm_out <= "01101";
      muxout <= muxd;
  ELSE
      state  <= s1;
      sm_out <= "00101";
  END IF;
WHEN s2 =>
  IF (sm_in(4) = '1' AND sm_in(2) = '1' AND
  sm_in(0) = '1') THEN
      state  <= s3;
      sm_out <= "11111";
  ELSIF (sm_in(0) = '1') THEN
      state  <= s2;
      sm_out <= "11010";
      muxout <= muxb;
  ELSIF (sm_in(4) = '1' AND sm_in(3) = '1' AND
  sm_in(1) = '1') THEN
      state  <= s1;
      sm_out <= "01010";
      count <= count - "1";
  ELSIF (sm_in(4) = '1' AND sm_in(1) = '1') THEN
      state  <= s4;
      sm_out <= "10011";
  ELSE
      state  <= s2;
      sm_out <= "10100";
  END IF;
WHEN s3 =>
  IF (sm_in(3) = '1') THEN
      state  <= s2;
      sm_out <= "01011";
```

Figure 14. Inefficient VHDL Code for a Sample State Machine (Part 4 of 4)

```
                muxout <= muxc;
            ELSE
                state  <= s3;
                sm_out <= "10100";
            END IF;
          WHEN s4 =>
            IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
            sm_in(2) ='1' AND sm_in(0) = '1') THEN
                state  <= s2;
                sm_out <= "11101";
            ELSE
                state  <= s4;
                sm_out <= "00111";
                count <= c_in;
            END IF;
          WHEN s5 =>
            IF (sm_in(3) = '1' AND sm_in(1) = '1' AND
            sm_in(0) ='1') THEN
                state  <= s2;
                sm_out <= "00110";
            ELSIF (sm_in(4) = '1' AND sm_in(3) = '1') THEN
                state  <= s7;
                sm_out <= "00001";
            ELSIF (sm_in(4) = '1' AND sm_in(3) = '1' AND
            sm_in(2) = '1' AND sm_in(1) = '1' AND
            sm_in(0) = '1')THEN
                state  <= s3;
                sm_out <= "10100";
                muxout <= muxd;
            ELSE
                state  <= s5;
                sm_out <= "00101";
                count <= count + "1";
            END IF;
          WHEN s6 =>
                state  <= s1;
                sm_out <= "10011";
          WHEN s7 =>
                state  <= s7;
                sm_out <= "10011";
          END CASE;
      END IF;
   END PROCESS;
END sm;
```

Figure 15 shows a more efficient description of the state machine design shown in Figure 14. In this example, the functionality of the counter and multiplexer are implemented in separate processes from the state machine.

Figure 15. Efficient VHDL Code for a Sample State Machine (Part 1 of 5)

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

ENTITY vhdl_good2 IS
 PORT (
        clk    : IN  STD_LOGIC;
        rst    : IN  STD_LOGIC;
        muxa   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        muxb   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        muxc   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        muxd   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        c_in   : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        sm_in  : IN  STD_LOGIC_VECTOR(4 DOWNTO 0);
        sm_out : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
        c_out  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        muxout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
       );
END vhdl_good2;

ARCHITECTURE sm OF vhdl_good2 IS

   TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, s6, s7);
   SIGNAL state STATE_TYPE;
   SIGNAL count    : STD_LOGIC_VECTOR(31 DOWNTO 0);
   SIGNAL count_en : STD_LOGIC;
   SIGNAL count_id : STD_LOGIC;
   SIGNAL count_ud : STD_LOGIC;
   SIGNAL muxel    : STD_LOGIC_VECTOR(1 DOWNTO0);

BEGIN
multiplexer:PROCESS(muxsel,muxa,muxb,muxc,muxd)

BEGIN
    CASE muxsel IS
        WHEN "00" =>
          muxout <= muxa;
        WHEN "01" =>
          muxout <= muxb;
        WHEN "10" =>
          muxout <= muxc;
```

Figure 15. Efficient VHDL Code for a Sample State Machine (Part 2 of 5)

```
        WHEN "11" =>
          muxout <= muxd;
        WHEN OTHERS =>
          muxout <= muxa;
    END CASE;
END PROCESS;

count: PROCESS (clk)
    VARIABLE direction : INTEGER;
    BEGIN
        IF (count_ud = '1') THEN
          direction := 1;
        ELSE
          direction := -1;
        END IF;
        IF rst = '1' THEN
          count <= "00000000000000000000000000000000";

        ELSIF (clk’EVENT AND clk = '1') THEN
             IF count_ld = '0' THEN
                 count <= c_in;
             ELSE
                 IF count_en = '1' THEN
                       count <= count + direction;
                 END IF;
             END IF;
          END IF;

          c_out <= count;
    END PROCESS;

    PROCESS (clk, rst)
    BEGIN
      IF (rst = '1') THEN
          state <= s0;
          sm_out <= "00000";
          muxsel <= "00";
          count_ld  <= '0';
          count_en  <= '0';
          count_ud  <= '0';
      ELSIF (clk’EVENT AND clk = '1') THEN
          count_ld  <= '0';
          count_en  <= '0';
          count_ud  <= '1';
```

Figure 15. Efficient VHDL Code for a Sample State Machine (Part 3 of 5)

```
muxsel    <= "00";
CASE state IS
  WHEN s0 =>
    IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
    sm_in(1) ='1' AND sm_in(0) = '1') THEN
      state  <= s6;
      sm_out <= "00110";
    ELSIF (sm_in(1) = '1') THEN
      state  <= s5;
      sm_out <= "01110";
      count_en <= '1';
    ELSE
      state  <= s0;
      sm_out <= "11110";
      muxsel <= "01";
    END IF;
  WHEN s1 =>
    IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
    sm_in(1) ='1' AND sm_in(0) = '1') THEN
      state  <= s7;
      sm_out <= "11111";
    ELSIF (sm_in(4) = '1' AND sm_in(1) = '1') THEN
      state  <= s4;
      sm_out <= "10110";
      count_ld <= '1';
      muxsel <= "10";
    ELSIF (sm_in(3) = '1') THEN
      state  <= s3;
      sm_out <= "00011";
    ELSIF (sm_in(4) = '1') THEN
      state  <= s2;
      sm_out <= "01101";
      muxsel <= "11";
    ELSE
      state  <= s1;
      sm_out <= "00101";
    END IF;
  WHEN s2 =>
    IF (sm_in(4) = '1' AND sm_in(2) = '1' AND
    sm_in(0) ='1') THEN
      state  <= s3;
      sm_out <= "11111";
    ELSIF (sm_in(0) = '1') THEN
      state  <= s2;
      sm_out <= "11010";
```

Figure 15. Efficient VHDL Code for a Sample State Machine (Part 4 of 5)

```
        muxsel <= "01";
      ELSIF (sm_in(4) = '1' AND sm_in(3) = '1' AND
      sm_in(1) = '1') THEN
        state  <= s1;
        sm_out <= "01010";
        count_en <= '1';
        count_ud <= '0';
      ELSIF (sm_in(4) = '1' AND sm_in(1) = '1') THEN
        state  <= s4;
        sm_out <= "10011";
      ELSE
        state  <= s2;
        sm_out <= "10100";
      END IF;
    WHEN s3 =>
      IF (sm_in(3) = '1') THEN
        state  <= s2;
        sm_out <= "01011";
        muxsel <= "10";
      ELSE
        state  <= s3;
        sm_out <= "10100";
      END IF;
    WHEN s4 =>
      IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
      sm_in(2) ='1' AND sm_in(0) = '1') THEN
        state  <= s2;
        sm_out <= "11101";
      ELSE
        state  <= s4;
        sm_out <= "00111";
        count_ld <= '1';
      END IF;
    WHEN s5 =>
      IF (sm_in(3) = '1' AND sm_in(1) = '1' AND
      sm_in(0) ='1') THEN
        state  <= s2;
        sm_out <= "00110";
      ELSIF (sm_in(4) = '1' AND sm_in(3) = '1') THEN
        state  <= s7;
        sm_out <= "00001";
      ELSIF (sm_in(4) = '1' AND sm_in(3) = '1' AND
      sm_in(2) = '1' AND sm_in(1) = '1' ANDD
      sm_in(0) = '1')THEN
        state  <= s3;
        sm_out <= "10100";
```

Figure 15. Efficient VHDL Code for a Sample State Machine (Part 5 of 5)

```
                muxsel <= "11";
              ELSE
                state  <= s5;
                sm_out <= "00101";
                count_en <= '1';
              END IF;
            WHEN s6 =>
                state  <= s1;
                sm_out <= "10011";
            WHEN s7 =>
                state  <= s7;
                sm_out <= "10011";
          END CASE;
       END IF;
    END PROCESS;
END sm;
```

One-hot encoding uses one bit per state, which uses more state registers but reduces the decoding logic required. To optimize a state machine, use one-hot encoding when targeting FLEX 10K devices. FLEX architectures are register-rich, look-up table (LUT)-based architectures that work well with low-fan-in decoding logic. Consequently, one-hot encoding increases performance and efficiency for these devices.

Implementing State Machines in EABs

Another design technique that increases state machine performance for FLEX 10K devices is placing state machines in EABs. Follow these guidelines when placing a state machine in EABs:

- Extremely complex state machines with limited I/O are ideal for implementing in EABs.
- Ensure that the state machine does not contain any latches.
- Place the state machine in a lower-level file and provide the feedback on an upper level.
- Break up large state machines into smaller state machines.
- Register all or none of the inputs and all or none of the outputs.
- Use registers in EABs for best results.
- Base the number of inputs to a state machine in an EAB on the number of states and outputs.

State machines can be implemented in EABs by assigning MAX+PLUS II software options or by using synthesis attributes during Synplify synthesis.

The Synplify software uses the following code to implement state machines in EABs:

VHDL:

```
ATTRIBUTE altera_implement_in_eab : BOOLEAN;
ATTRIBUTE altera_implement_in_eab OF U1: LABEL IS TRUE;
```

where `U1` is the function to be implemented in the EAB.

Verilog HDL:

```
sqrtb sq (.z(sqa), .a(a)) /* synthesis
altera_implement_in_eab=1 */;
```

where `sqrtb` is the module to be implemented in the EAB.

Figures 16 through 19 show a VHDL state machine that is implemented in an EAB and has the following attributes:

- Registers are implemented in LEs.
- A state machine contains the state machine control logic.
- A top-level module instantiates the register. The state machine provides feedback, allowing the state machine to be implemented in a FLEX 10K EAB.

Figure 16 defines the states of the state machine.

Figure 16. VHDL Code for my_pack.vhd

```
PACKAGE my_pack IS
   TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, serror);
END my_pack;
```

Figure 17 shows the description of registers implemented in LEs.

Figure 17. VHDL Code for register.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.my_pack.ALL;

ENTITY registers IS
PORT ( next_state   : IN STATE_TYPE;
       temp_out     : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       clock, reset : IN STD_LOGIC;
       out_state    : OUT STATE_TYPE;
       result       : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END registers;

ARCHITECTURE behave OF registers IS
BEGIN
   PROCESS (clock, reset)
   BEGIN
      IF (reset = '0') THEN
             result <= "0000";
      ELSIF (clock = '1' AND clock'EVENT) THEN
             result <= temp_out;
             out_state <= next_state;
      END IF;
   END PROCESS;
END behave;
```

Figure 18 shows the VHDL code for **maclogic.vhd**, which is the lower-level state machine design without feedback.

Figure 18. VHDL Code for maclogic.vhd (Part 1 of 3)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.my_pack.ALL;

ENTITY maclogic IS
PORT ( input        : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
       in_state     : IN STATE_TYPE;
       next_state   : OUT STATE_TYPE;
       temp_out     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
```

Figure 18. VHDL Code for maclogic.vhd (Part 2 of 3)

```
       clock, reset : IN STD_LOGIC);
END maclogic;

ARCHITECTURE behave OF maclogic IS

BEGIN
       PROCESS(in_state,input)
       BEGIN
           temp_out <="0000";

           CASE in_state IS
               WHEN s0 =>
                   IF (input = "000") THEN
                           next_state <= s1;
                   ELSE
                           next_state <= s0;
                   END IF;
                   temp_out <= "0000";
               WHEN s1 =>
                   IF (input = "001") THEN
                           next_state <= s2;
                   ELSE
                           next_state <= s1;
                   END IF;
                   temp_out <= "0001";
               WHEN s2 =>
                   IF (input = "010") THEN
                           next_state <= s3;
                   ELSE
                           next_state <= s2;
                   END IF;
                   temp_out <= "0010";
               WHEN s3 =>
                   IF (input = "011") THEN
                           next_state <= s4;
                   ELSE
                           next_state <= s3;
                   END IF;
                   temp_out <= "0011";
               WHEN s4 =>
                   IF (input = "100") THEN
                           next_state <= s5;
                   ELSE
                           next_state <= s4;
                   END IF;
                   temp_out <= "0100";
```

Figure 18. VHDL Code for maclogic.vhd (Part 3 of 3)

```
              WHEN s5 =>
                  IF (input = "101") THEN
                         next_state <= s0;
                  ELSE
                         next_state <= s5;
                  END IF;
                  temp_out <= "0101";
              WHEN serror =>
                  IF (input = "111") THEN
                         next_state <= s0;
                  ELSE
                         next_state <= serror;
                  END IF;
                  temp_out <= "1111";
              WHEN OTHERS =>
                         next_state <= serror;
                  temp_out <= "1111";
       END CASE;
    END PROCESS;

END behave;
```

Figure 19 shows the VHDL code for **mac.vhd**, which is the top-level file providing feedback on the state machine. The **mac.vhd** file links the two design elements structurally.

Figure 19. VHDL Code for mac.vhd (Part 1 of 2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.my_pack.ALL;

ENTITY mac IS
PORT(  clock,reset : IN STD_LOGIC;
       input       : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
       outputs     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END mac;

ARCHITECTURE behave OF mac IS
       COMPONENT maclogic
       PORT( input        : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
             temp_out     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
             in_state     : IN STATE_TYPE;
             next_state   : OUT STATE_TYPE;
             clock, reset : STD_LOGIC);
```

Figure 19. VHDL Code for mac.vhd (Part 2 of 2)

```
       END COMPONENT;
       ATTRIBUTE altera_implement_in_eab      : BOOLEAN;
       ATTRIBUTE altera_implement_in_eab OF U1: LABEL IS TRUE;

       COMPONENT registers
       PORT( next_state     : IN  STATE_TYPE;
             temp_out       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
             clock, reset   : IN  STD_LOGIC;
             out_state      : OUT STATE_TYPE;
             result         : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
       END COMPONENT;
       SIGNAL temp         : STATE_TYPE;
       SIGNAL tempstate    : STATE_TYPE;
       SIGNAL temp_result  : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
       u1 : maclogic
       PORT MAP (input => input, temp_out => temp_result, next_state =>
                 tempstate, in_state => temp, clock => clock, reset => reset);
       u2 : registers
       PORT MAP (next_state => tempstate, temp_out => temp_result, clock =>
                 clock, reset => reset, out_state => temp, result => outputs);
END behave;
```

Synplify Design Techniques for FLEX 10K Devices

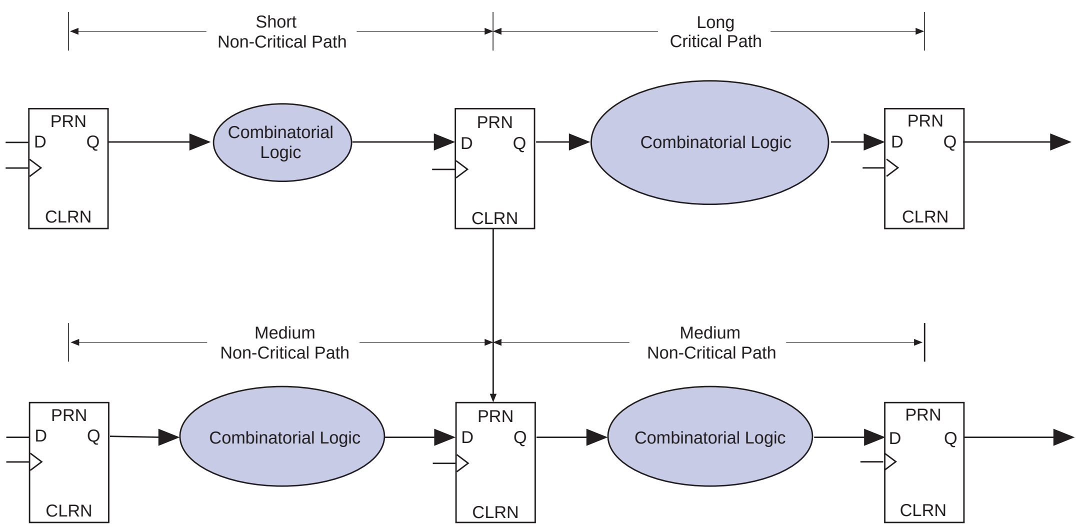
You can optimize FLEX 10K device performance by using the following design techniques:

- Register Balancing
- Pipelining
- Logic Duplication
- LPM Functions

Register Balancing

Register balancing is a technique used to reduce long delays and to increase shorter delays. This technique is beneficial in applications where registers are used purely for latency purposes. See Figure 20.

Figure 20. Register Balancing

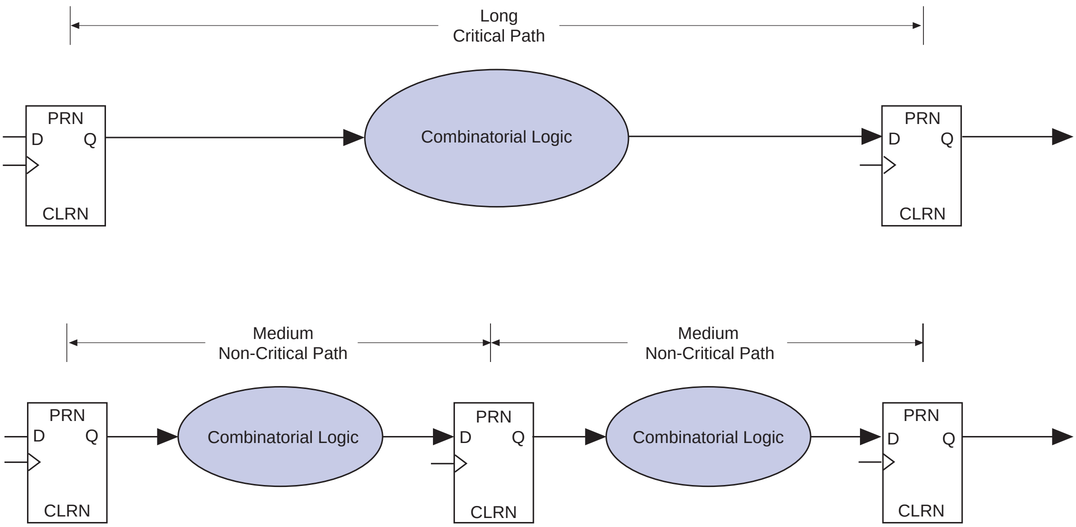


Pipelining

Pipelining is a technique that uses registers rather than combinatorial latches to hold logic. When pipelining a design, you add registers to break up large combinatorial delays. Because the FLEX architecture includes a register with each LUT, pipelining combinatorial logic generally does not require additional device resources. Therefore, using registers for pipelining improves performance without increasing the LE utilization in a device. See Figure 21.

☞ When pipelining, each register adds one degree of latency (e.g., on counters, decode one clock cycle ahead per degree of latency).

Figure 21. Pipelining a Design



Logic Duplication

Logic duplication is a good method for reducing fan-out and improving the design performance on a given path. You should use logic duplication on high-fan-out nodes and flipflops because it reduces the number of loads these signals drive and can potentially ease routing.

Synplicity provides the `syn_preserve` attribute, which forces Synplify synthesis to preserve a register and not optimize across it. This attribute prevents duplicate logic from being optimized out.

The Synplify software uses the following code to implement the `syn_preserve` attribute:

VHDL:

```
ATTRIBUTE syn_preserve : BOOLEAN;
ATTRIBUTE syn_preserve OF signal_name : SIGNAL IS TRUE;
```

Verilog HDL:

```
reg foo /* synthesis syn_preserve=1 */ ;
```

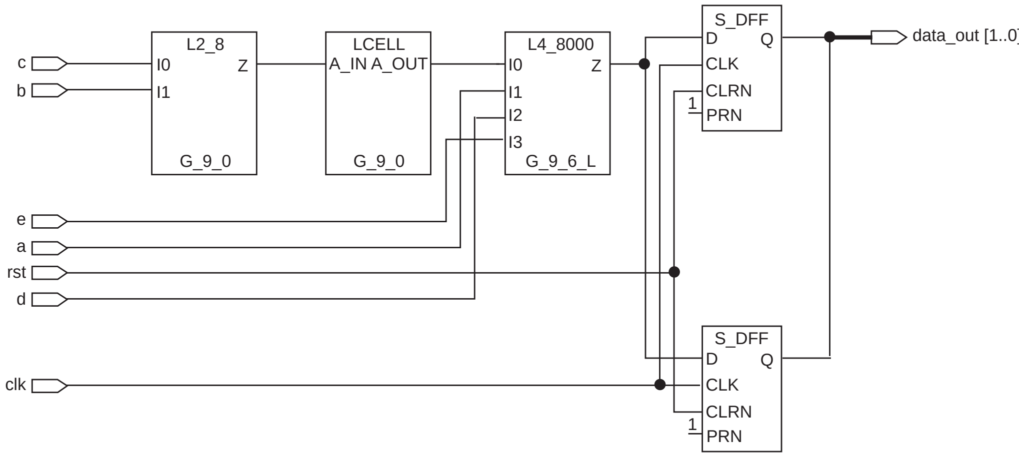
The VHDL code in Figure 22 demonstrates the use of the `syn_preserve` attribute for logic duplication.

Figure 22. Using the syn_preserve Attribute for Logic Duplication

```
ENTITY split IS
   PORT(  clk        : IN STD_LOGIC;
          a,b,c,d,e  : IN STD_LOGIC;
          rst        : IN STD_LOGIC;
          data_out   : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END split;

ARCHITECTURE behave OF split IS
   SIGNAL inter                       : STD_LOGIC_VECTOR(1 DOWNTO 0);
   ATTRIBUTE syn_preserve             : BOOLEAN;
   ATTRIBUTE syn_preserve OF inter : SIGNAL IS TRUE;
BEGIN
   reg: PROCESS(clk,rst)
      BEGIN
                      IF rst = '0' THEN
                           inter <= "00";
                      ELSIF (clk = '1' AND clk'EVENT) THEN
                           inter(0) <= (a AND b AND c AND d AND e);
                           inter(1) <= (a AND b AND c AND d AND e);
                      END IF;
      END PROCESS;
   data_out(0) <= inter(0);  -- the registers inter(0) and inter(1) are
   data_out(1) <= inter(1);     duplicates.

END behave;
```

Figure 23 shows the schematic representation of the VHDL logic from Figure 22.

Figure 23. Schematic Representation of Preserving Logic for Logic Duplication



If the `syn_preserve` attribute is not used, as shown in Figure 24, one of the registers is optimized out.

Figure 24. Non-Use of the syn_preserve Attribute (Part 1 of 2)

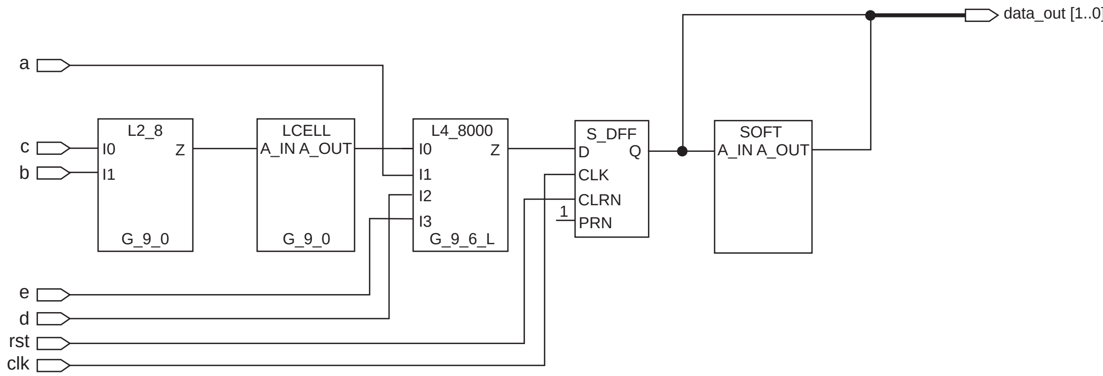
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY split3 IS
    PORT(  clk        : IN STD_LOGIC;
           a,b,c,d,e  : IN STD_LOGIC;
           rst        : IN STD_LOGIC;
           data_out   : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END split3;

ARCHITECTURE behave OF split3 IS
SIGNAL inter: STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN
    reg: process(clk,rst)
        BEGIN
                        IF rst = '0' THEN
                              inter <= "00";
```

Figure 24. Non-Use of the syn_preserve Attribute (Part 2 of 2)

```
                              ELSIF (clk = '1' AND clk'EVENT) THEN
                                   inter(0) <= (a AND b AND c AND d AND e);
                                   inter(1) <= (a AND b AND c AND d AND e);
                              END IF;
         END PROCESS;
    data_out(0) <= inter(0);
    data_out(1) <= inter(1);
END behave;
```

Figure 25 shows the schematic representation without logic duplication.

Figure 25. Schematic Representation Without Logic Duplication



LPM Functions

Functions from the library of parameterized modules (LPM) are large building blocks that can be customized easily for your application by using different ports and by setting different parameters. Altera optimizes LPM functions for Altera device architectures.

Within the Synplify software, LPM functions are compiled as black boxes. The parameter values are expressed as Verilog HDL meta comments or as VHDL attributes, which are passed to the EDIF file or Text Design File (**.tdf**).

Figure 26 shows a VHDL design that uses the `lpm_mult` function to create a pipelined multiplier with two levels of pipelining.

Figure 26. Instantiating an LPM Function in VHDL (Part 1 of 2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mult IS
PORT(  a     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
       b     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
       clock : IN  STD_LOGIC;
       p     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END mult;

ARCHITECTURE a OF mult IS

COMPONENT my_mult

   PORT ( dataa  : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
          datab  : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
          aclr   : IN STD_LOGIC := '0';
          clock  : IN STD_LOGIC := '0';
          sum    : IN STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
          result : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

ATTRIBUTE black_box    : BOOLEAN;
ATTRIBUTE lpm_widtha   : INTEGER;
ATTRIBUTE lpm_widthb   : INTEGER;
ATTRIBUTE lpm_widths   : INTEGER;
ATTRIBUTE lpm_widthp   : INTEGER;
ATTRIBUTE lpm_pipeline : INTEGER;
ATTRIBUTE lpm_type     : STRING;

-- assign the appropriate attribute values

ATTRIBUTE black_box OF my_mult    : COMPONENT IS TRUE;
ATTRIBUTE lpm_widtha OF my_mult   : COMPONENT IS 4;
ATTRIBUTE lpm_widthb OF my_mult   : COMPONENT IS 4;
ATTRIBUTE lpm_widths OF my_mult   : COMPONENT IS 8;
ATTRIBUTE lpm_widthp OF my_mult   : COMPONENT IS 8;
ATTRIBUTE lpm_pipeline OF my_mult : COMPONENT IS 2;
ATTRIBUTE lpm_type OF my_mult     : COMPONENT IS "LPM_MULT";

SIGNAL tmp_result: STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Figure 26. Instantiating an LPM Function in VHDL (Part 2 of 2)

```
BEGIN
   u1: my_mult
        PORT MAP (dataa => a(3 DOWNTO 0), datab => b(3 DOWNTO 0), clock
        => clock, result => tmp_result);
   PROCESS(clock)
   BEGIN
        IF (clock'EVENT AND clock ='1') THEN
             p <= tmp_result;
        END IF;
   END PROCESS;
END a;
```

Figure 27 shows a Verilog HDL design that uses the `lpm_ram_dq` function to create a 64 × 16 RAM block.

Figure 27. Instantiating an LPM Function in Verilog HDL

```
// lpm ram example
//
// define the black box here (pick a name, such as myram_64x16),
// note that immediately after the port list, but before the semicolon ';'
// the black_box synthesis directive is included with the lpm_type specified
// as "lpm_ram_dq", and other parameter values are set.

module myram_64x16 (data,address,inclock,
    outclock,we,q) /* synthesis black_box lpm_width=16 lpm_widthad=6
    lpm_type="lpm_ram_dq" */ ;

    input [15:0] data;
    input [5:0] address;
    input inclock, outclock;
    input we;
    output [15:0] q;
    endmodule

// instantiate the lpm function in the
// higher-level module myram,
module myram(clock, we, data, address, q);
    input clock, we;
    input [15:0] data;
    input [5:0] address;
    output [15:0] q;

myram_64x16 inst1 (data,address,clock, clock, we, q);
endmodule
```

Complete details on the LPM functions supported by the MAX+PLUS II software are available in MAX+PLUS II Help.

Synplify Settings

The following sections describe the best settings to use in the Synplify software when you process files targeted for Altera devices.

Map Logic to LCELLs

When you turn on the *Map Logic to LCELLs* option, the Synplify software maps logic to LUTs, carry chains, and cascade chains and optimizes the design for performance.

However, turning this option off may give better device utilization and should be used when the MAX+PLUS II software fails to fit your design.

Perform Cliquing

The *Perform Cliquing* option can be selected only when the *Map Logic to LCELLs* option is turned on and the output netlist type has been set to EDIF. This setting allows the Synplify software to use cliques for critical paths automatically. The cliques are defined in the EDIF netlist output from the Synplify software. This option is not available for TDFs.

Automatic cliquing controls the place-and-route of performance-critical paths. On average, it improves performance by 10% to 15%.

MAX+PLUS II Options For High Performance

After synthesizing your design in the Synplify software, you can import the resulting EDIF netlist file into the MAX+PLUS II software. The following sections describe the recommended synthesis style and other options you can use to achieve optimum performance.

Synthesis Style

When the *Map Logic to LCELLs* option is turned on in the Synplify software, the Synplify software automatically optimizes the design for the FLEX 10K architecture. Therefore, Altera recommends using the *WYSIWYG* logic synthesis style in the MAX+PLUS II software (with the *NOT-Gate Push-back* option turned off). If this style does not give the required performance, try the *FAST* logic synthesis style. Generally, the *WYSIWYG* logic synthesis style gives the best results, but in some cases the *FAST* logic synthesis style produces better results.

Perform the following steps to set your MAX+PLUS II compilation options:

1. Start the MAX+PLUS II software.

2. Choose **Project Name** (File menu). In the **Project Name** dialog box, select the appropriate Synplify software-generated EDIF file *<working directory>*/*<project name>***.edf** and click **OK**.

☞ Altera recommends that you store the Synplify EDIF file in a separate directory from the HDL source files.

3. Choose **Compiler** (MAX+PLUS II menu).

4. Select *Synplicity* as the vendor in the **EDIF Netlist Reader Settings** dialog box (Interfaces menu). Click **OK**.

5. Select the appropriate device in the **Device** dialog box (Assign menu). Click **OK**.

6. Turn on the appropriate netlist writer command (Interfaces menu). For example, if you want to create a VHDL Output File (**.vho**), turn on the **VHDL Netlist Writer** command.

7. Turn on the appropriate netlist writer settings (Interfaces menu). For example, if you want to create a VHDL Output File, select the ***VHDL Output File [.vho]*** option. Click **OK**.

8. Choose **Global Project Logic Synthesis** (Assign menu). In the **Global Project Logic Synthesis** dialog box, set the synthesis style to *WYSIWYG* for FLEX designs that map LEs in the Synplify EDIF file. Click **Define Synthesis Style**. In the **Define Synthesis Style** dialog box, click **Advanced Options**. In the **Advanced Options** dialog box, turn off the *NOT-Gate Push-Back* option. Click **OK** three times to close the dialog boxes.

9. Click the **Start** button to run the MAX+PLUS II Compiler.

For more information on creating designs and compiling them within the MAX+PLUS II software, go to the MAX+PLUS II ACCESS[SM] Key Guidelines for the Synplicity Synplify software.

Using the Fast I/O Logic Option

FLEX 10K devices have built-in registers close to the I/O pins. These input/output element (IOE) registers have faster clock-to-output delays ($\mathbf{t}_{CO}$) than buried registers.

Fast I/O is a logic option that can be applied to registers. This option instructs the MAX+PLUS II Compiler to implement the register in an LE or IOE having a fast, direct connection to an input or I/O pin. Turning the *Fast I/O* option on helps maximize timing performance (e.g., by permitting fast setup times).

You can also apply the *Fast I/O* logic option to pins with the following results:

- On input pins, the MAX+PLUS II Compiler moves the assignment to the IOE or LE fed by the input.
- On output pins, it moves the assignment to the I/O cell or logic cell feeding the output.

Perform the following steps to allow the MAX+PLUS II Compiler to make project-wide *Fast I/O* assignments:

1. Choose **Global Project Logic Synthesis** (Assign menu).

2. Turn on the *Automatic Fast I/O* option in the **Global Project Logic Synthesis** dialog box. Click **OK**.

Perform the following steps to create *Fast I/O* assignments on individual registers:

1. Choose **Logic Options** (Assign menu). In the **Logic Options** dialog box, enter the node name of the register and click **Individual Logic Options**.

2. Turn on the *Fast I/O* option in the **Individual Logic Options** dialog box.

3. Click **OK** twice in the appropriate dialog boxes to save your changes.

4. Click **Start** in the MAX+PLUS II Compiler to compile.

Timing-Driven Compilation

On average, timing-driven compilation improves device performance by 15% to 30%, depending on resource utilization.

☞ Generally, devices with higher resource utilization take longer to compile. When devices are fully utilized, the MAX+PLUS II software works harder to fit the design, while ensuring that all timing requirements are met. This full utilization can increase the compile time by as much as 10 times.

You can specify four different types of timing constraints for a project. See Table 1.

Table 1. Timing Constraints Used in Timing-Driven Compilation

Parameter	Definition	Implementation in MAX+PLUS II
t_{PD}	t_{PD} (input to non-registered output delay) is the time required for a signal from an input pin to propagate through combinatorial logic and appear at an external output pin.	You can specify a required t_{PD} for an entire project and/or for any input pin, output pin, or `TRI` buffer that feeds an output pin.
t_{CO}	t_{CO} (clock-to-output delay) specifies the maximum acceptable clock-to-output delay. The output pin is fed by a register (after a clock signal transition) on an input pin that clocks the register. This time always represents an external pin-to-pin delay.	You can specify a required t_{CO} for an entire project and/or for any input pin, output pin, or `TRI` buffer that feeds an output pin.
t_{SU}	t_{SU} (clock setup time) is the length of time for which data that feeds a register via its data or enable input(s) must be present at an input pin before the clock signal that clocks the register is asserted at the clock pin.	You can specify a required t_{SU} for an entire project and/or for any input pin or bidirectional pin.
f_{MAX}	f_{MAX} (maximum clock frequency) is the maximum clock frequency that can be achieved without violating internal setup and hold time requirements.	You can specify a required f_{MAX} for an entire project and/or for any input pin, bidirectional pin, or a register.

To assign timing requirements globally, choose **Global Project Timing Requirements** (Assign menu). To assign timing requirements to individual critical paths, choose **Timing Requirements** (Assign menu).

Synplify Constraints for Improving Performance

If your design fails to meet performance requirements when you use the options described for the MAX+PLUS II software, you will need to repeat the design process shown in Figure 1 on page 3. You should apply the techniques described in the following sections to reprocess your design within the Synplify software. The following topics are addressed in this section:

- Symbolic Finite State Machine (FSM) Compiler
- Timing Constraints
- Resource Sharing
- HDL Analyst
- Pin Locking
- Using EABs for Memory
- Using EABs for Logic

Symbolic FSM Compiler

The Synplify Symbolic FSM Compiler utility can be used to improve the performance of state machine designs.

The Symbolic FSM Compiler automatically detects and re-encodes state machines. For FLEX devices, the state machine is re-encoded as a one-hot state machine (one-hot encoding uses one bit per state, which uses more state registers but reduces the decoding logic required). Turn on the *Symbolic FSM Compiler* option in the project window.

Figure 28 shows a Verilog HDL state machine design that has been implemented as a binary-encoded state machine.

Figure 28. Binary-Encoded Verilog HDL State Machine (Part 1 of 3)

```
/* prep3 contains a small state machine */

module stmch1(clk, rst, in, out);
input clk, rst;
input [7:0] in;
output [7:0] out;

reg [7:0] out;
reg [3:0] current_state; // holds the current state

parameter // these parameters represent state names
     start = 3'b000,
     sa = 3'b001,
     sb = 3'b010,
     sc = 3'b011,
     sd = 3'b100,
     se = 3'b101,
```

Figure 28. Binary-Encoded Verilog HDL State Machine (Part 2 of 3)

```
     sf = 3'b110,
     sg = 3'b111;

always @ (posedge clk or posedge rst)

begin
   if (rst) begin
        current_state = start;
        out = 8'b0;
   end else begin
        case (current_state)
        start: if (in == 8'h3c) begin
                    current_state = sa;
                    out = 8'h82;
             end else begin
                    out = 8'h00;
                    current_state = start;
             end
        sa: if (in == 8'h2a) begin
                    current_state = sc;
                    out = 8'h40;
             end else if (in == 8'h1f) begin
                    current_state = sb;
                    out = 8'h20;
             end else begin
                    current_state = sa;
                    out = 8'h04;
             end
        sb: if (in == 8'haa) begin
                    current_state = se;
                    out = 8'h11;
             end else begin
                    current_state = sf;
                    out = 8'h30;
             end
        sc: begin
             current_state = sd;
             out = 8'h08;
             end
        sd: begin
             current_state = sg;
             out = 8'h80;
             end
        se: begin
             current_state = start;
```

Figure 28. Binary-Encoded Verilog HDL State Machine (Part 3 of 3)

```
                out = 8'h40;
                end
        sf: begin
                current_state = sg;
                out = 8'h02;
                end
        sg: begin
                current_state = start;
                out = 8'h01;
                end
        default: begin
        /* set current_state to 'bx (don't care) to tell the synplify software
        that all used states have been already been specified */
                current_state = 'bx;
                out = 'bx;
                end
        endcase
     end
end
endmodule
```

When synthesized by the Synplify software and compiled with the MAX+PLUS II software, this design has an $\mathbf{f_{MAX}}$ of 75.18 MHz and uses 27 LEs.

If re-synthesized in the Synplify software with the *Symbolic FSM Compiler* option turned on, the binary encoding is re-encoded by Synplicity as one-hot encoding. This representation produces an $\mathbf{f_{MAX}}$ of 85.47 MHz and uses 27 LEs when recompiled in the MAX+PLUS II software.

If the *Symbolic FSM Compiler* option is turned off, you can still enable state machine optimization on a state register-by-state register basis with the `state_machine` attribute.

The Synplify software uses the following code to implement the `state_machine` attribute:

VHDL:

```
SIGNAL curstate : STATE_TYPE;
ATTRIBUTE state_machine : BOOLEAN;
ATTRIBUTE state_machine OF curstate : SIGNAL IS TRUE;
```

Verilog HDL:

```
reg [3:0] curstate /* synthesis state_machine */ ;
```

Timing Constraints

The Synplify software provides a number of timing constraints that affect Synplify synthesis and can be used to improve design performance.

Use timing constraints in conjunction with the results from the HDL Analyst (in Technology view), as described in "HDL Analyst" on page 49.

Timing constraints are contained in a Synplicity Design Constraint (**.sdc**) file. To include a constraint file in a project, add it to the *Source Files* list box in the Synplify Project Window by clicking the **Add** button and selecting the file.

Alternatively, you can read in a constraint file with the `add_file -constraint` command when running synthesis from a tcl script. A variety of constraint commands are available.

Set the Clock Frequency

You can enter the default clock frequency for a design in the project window. However, if the design has multiple clocks, specify the frequency for each clock using the attribute shown below:

`define_clock` <*clock name*> {`-freq` <*MHz*> | `-period` <*ns*>}

Example: `define_clock sysclk -freq 33 MHz`

Improve the Delay on a Path Feeding a Register Input

You can use the `define_reg_input_delay` timing constraint to speed up paths feeding into a register by a given number of nanoseconds with the following command:

`define_reg_input_delay` <*register name*>[`-improve` <*ns*>]
    [`-route` <*ns*>]

Example: `define_reg_input_delay data_reg -improve 5 ns`

<*register name*> must be a single-bit register and not a bus. For multiple register bits, use multiple `define_reg_input_delay` commands.

If the HDL Analyst or the Log File Timing Report reports that a flipflop has negative slack, your clock frequency goal was not met because of paths to the register. Use `define_reg_input_delay` for that register with the `-improve` option. Using the `-improve` option forces the Synplify software to restructure your design during optimization to try to meet your clock frequency goal. Negative `-improve` values are allowed, and are useful when you have extra time and want to relax the constraints on the paths to some flipflops. By relaxing the constraints in this way, the Synplify software can better improve timing for other paths in your design.

Improve the Delay on a Path from the Output of a Register

The following constraint is similar to `define_reg_input_delay`, but it acts upon the path from the output of a register when that register is the source of a critical path:

```
define_reg_output_delay <register name> [-improve <ns>]
    [-route <ns>]
```

Improve the Delay on the Path Associated with an Input Pin

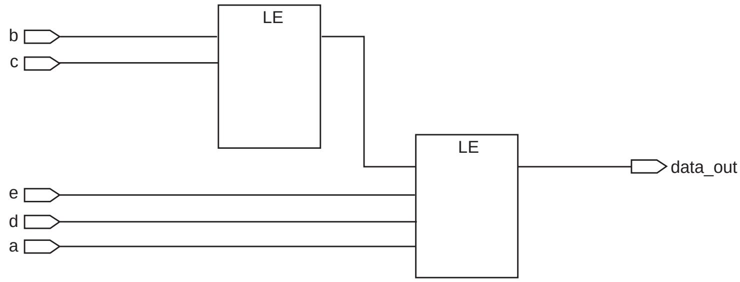
This improvement in delay is best illustrated in the VHDL code in Figure 29, which shows a five-input AND gate.

Figure 29. Five-Input AND Gate

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

ENTITY split IS
    PORT(  a,b,c,d,e  :IN STD_LOGIC;
           data_out   :OUT STD_LOGIC);
END split;

ARCHITECTURE behave OF split IS
BEGIN
         data_out <= (a AND b AND c AND d AND e);
END behave;
```

The function is implemented in a FLEX device using two LEs, as shown in Figure 30.

Figure 30. Implementation of a Five-Input AND Gate Using Two LEs



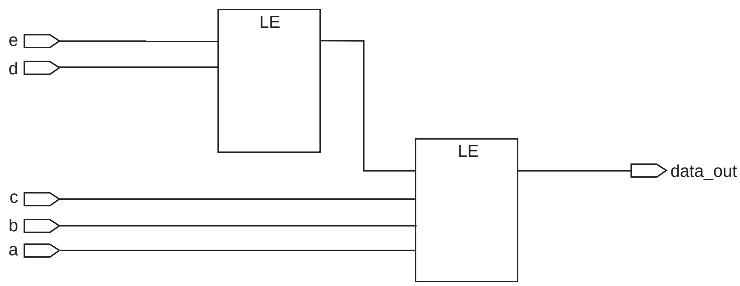
However, if the timing on input `b` is critical, you can use the `define_input_delay` constraint to force input `b` to pass through only one LE with the following command:

```
define_input_delay {<input port name> | -default} <ns>
    [-improve <ns>] [-route <ns>]
```

For example, the following command implements the result shown in Figure 31:

```
define_input_delay b 5
```

Figure 31. Controlling the Path of a Critical Signal



Resource Sharing

The Synplicity software provides the `syn_sharing` attribute, which controls resource sharing within a logic function. By default, the Synplicity software uses resource sharing. Figure 32 shows a Verilog HDL design that uses the default resource sharing.

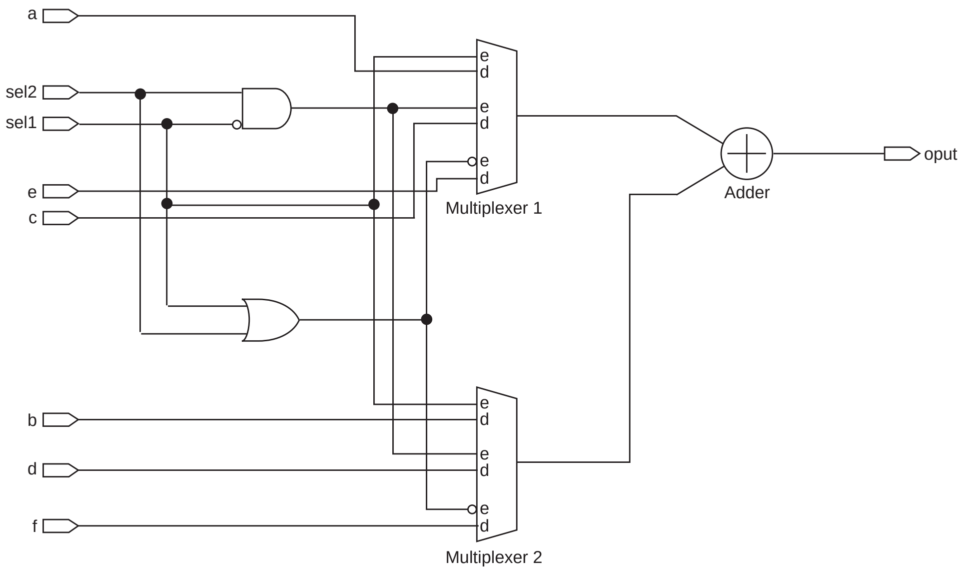
Figure 32. Using the Default Resource Sharing

No special attributes are required to use resource sharing.

```
module share (a,b,c,d,e,f,sel1,sel2,oput);
    input a,b,c,d,e,f;
    input sel1, sel2;
    output oput;
    reg oput;

    always @(a or b or c or d or e or f or sel1 or sel2)
    begin
         if (sel1)
             oput = a + b;
         else if (sel2)
             oput = c + d;
         else
             oput = e + f;
    end
endmodule
```

Figure 33 shows the schematic representation of the Verilog HDL design from Figure 32.

Figure 33. Resource Sharing



Because the Synplify software uses resource sharing by default, this design creates one adder with control logic that multiplexes the appropriate signals to its inputs.

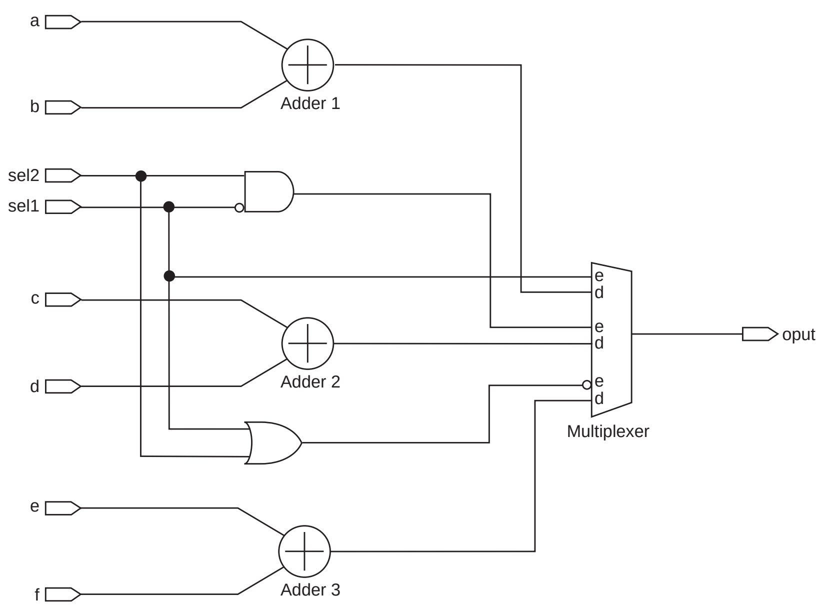
However, the `syn_sharing` attribute can be used to disable resource sharing, as shown in Figure 34.

Figure 34. Using the syn_sharing Attribute to Disable Resource Sharing

```
module no_share (a,b,c,d,e,f,sel1,sel2,oput); /*synthesis syn_sharing="off"*/
   input a,b,c,d,e,f;
   input sel1, sel2;
   output oput;
   reg oput;

   always @(a or b or c or d or e or f or sel1 or sel2)
   begin
     if (sel1)
          oput = a + b;
     else if (sel2)
          oput = c + d;
     else
          oput = e + f;
   end
endmodule
```

Figure 35 shows the schematic representation of the Verilog HDL design from Figure 34.

Figure 35. Disabling Resource Sharing



With this implementation, the outputs of three adders are multiplexed to the output. You must evaluate the results of resource sharing on a case-by-case basis to determine which implementation is best suited for your design.

HDL Analyst

HDL Analyst is a graphical productivity tool for the Synplify synthesis environment. This tool helps you visualize your synthesis results and improve your device's performance and area results.

HDL Analyst automatically generates hierarchical Register Transfer Language (RTL)-level and technology-primitive-level schematics (after the Synplify software technology-maps to LUTs) from your VHDL and Verilog HDL designs. This feature enables you to cross-probe between the RTL-level and technology-primitive-level schematics and your source code. For example, when you highlight a line of text in your code, the corresponding logic in the schematic view is highlighted. Conversely, double-clicking on logic in the schematic view takes you to the corresponding source code.

HDL Analyst's cross-probing capability enables you to analyze your design, and helps you visualize where design changes or adding timing constraints might reduce device area or increase device performance.

HDL Analyst also highlights and isolates critical paths within your design, allowing you to quickly analyze problem areas, add timing constraints, and re-synthesize.

To view your critical paths with HDL Analyst, follow these steps:

1. Synthesize your design.

2. Choose **Technology View** (HDL Analyst menu).

3. Turn on critical path highlighting for the Technology View by choosing **Show Critical Path** (HDL Analyst menu). You can also click the **Show Critical Path** button (stopwatch button) on the Synplify toolbar or the right mouse button pop-up menu. Use the **Show Critical Path** command to turn on and off critical path highlighting.

   The **Show Critical Path** command (HDL Analyst menu) highlights the critical paths in your design and displays timing numbers above every instance. You can analyze your critical paths from this window, or isolate them into their own schematic to make them easier to analyze.

4. Choose the **Filter Schematic** command (HDL Analyst menu) to isolate the critical paths. You can also click the **Filter Schematic** button (buffers button) on the Synplify toolbar or the right mouse button pop-up menu.

   The **Filter Schematic** command (HDL Analyst menu) takes all selected objects (in this case, the highlighted critical path instances) and groups them together in one schematic, removing all other objects from view. If the critical instances were on many schematic sheets, all of the critical instances are gathered together and shown, regardless of which sheet they were on originally.

5. Choose the **Filter Schematic** command (HDL Analyst menu) again to restore your original schematic.

6. Analyze your critical path with the **Show Critical Path** command turned on. Synplify displays two timing numbers above each instance (the delay and the slack time). Based upon this information, see "Synplify Constraints for Improving Performance" on page 40 to improve performance.

For more detailed information on HDL Analyst, go to Synplify on-line help.

Pin Locking

Within the MAX+PLUS II software, you can enter assignments for the current project by choosing Assign menu commands, moving node and pin names in the Floorplan Editor, or manually editing the Assignment & Configuration File (**.acf**). All of these assignment methods edit the ACF. However, only one application can edit the ACF at a time. If you manually edit the ACF in a Text Editor window, you must save your edits whenever you choose an Assign menu command or switch to the Floorplan Editor. Because manually editing the ACF is more likely to generate errors, Altera recommends entering assignments with Assign menu commands and the Floorplan Editor.

Additionally, Altera recommends compiling the project for the first time without any assignments. However, if you want to make particular assignments before an initial compilation, follow these steps:

1. Choose **Pin/Location/Chip** (Assign menu). In the **Pin/Location/Chip** dialog box, enter the node name or pin for which you wish to enter assignments.

2. Select a pin, logic cell, or other resource under *Chip Resources*.

3. Click **Add**, and then click **OK**.

4. Click **Start** in the MAX+PLUS II Compiler window to compile.

Alternatively, the Synplify software allows you to assign logic and signals to specific locations on an Altera device. These assignments are passed to the MAX+PLUS II software via the EDIF netlist produced by the Synplify software.

Figure 36 shows an example of assigning Verilog HDL signals to pins.

Figure 36. Verilog HDL Pin Assignment (Part 1 of 2)

```
module adder(cout, sum, a, b, cin);

   parameter size = 1;  /* declare a parameter. default required */
   output cout;
   output [size-1:0] sum;  // sum uses the size parameter
   input cin;
   input [size-1:0] a, b;  // 'a' and 'b' use the size parameter

   assign {cout, sum} = a + b + cin;
```

Figure 36. Verilog HDL Pin Assignment (Part 2 of 2)

```
endmodule

module adder8(cout, sum, a, b, cin);

    output cout /* synthesis altera_chip_pin_lc="adder8@159" */;

    output [7:0] sum /* synthesis
       altera_chip_pin_lc="@17,@166,@191,@152,@15,@148,@147,@149" */;

    input [7:0] a /* synthesis
       altera_chip_pin_lc="adder8@194,adder8@177,adder8@70,adder8@96,
       adder8@109,adder8@6,adder8@174,adder8@204" */;

    input [7:0]  b;
    input cin;

    adder my_adder (cout, sum, a, b, cin);
    defparam my_adder.size = 8;

endmodule
```

Figure 37 shows an example of assigning VHDL signals to pins.

Figure 37. VHDL Pin Assignment (Part 1 of 2)

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

ENTITY adder8 IS
    GENERIC (num_bits : INTEGER := 8) ;
    PORT ( a,b        : IN STD_LOGIC_VECTOR (NUM_BITS -1 DOWNTO 0);
           result     : OUT STD_LOGIC_VECTOR(NUM_BITS -1 DOWNTO 0));

ATTRIBUTE altera_chip_pin_lc : STRING;

ATTRIBUTE altera_chip_pin_lc OF RESULT : SIGNAL IS
"@17,@166,@191,@152,@15,@148,@147,@149";

ATTRIBUTE altera_chip_pin_lc OF a : SIGNAL IS
"adder8@194,adder8@177,adder8@70,adder8@96,adder8@109,adder8@6,adder8@174,
   adder8@204";
```

Figure 37. VHDL Pin Assignment (Part 2 of 2)

```
END adder8;

ARCHITECTURE behave OF adder8 IS

BEGIN
    result <= a + b;
END;
```

Using EABs for Memory

To implement RAM or ROM functions in an EAB, use the LPM functions shown in Table 2.

Table 2. LPM Functions that can be Implemented in EABs

LPM Functions	Description
`lpm_ram_dq`	Synchronous/asynchronous RAM (separate read/write data ports)
`lpm_ram_io`	Synchronous/asynchronous RAM (bidirectional read/write data ports)
`lpm_rom`	Synchronous/asynchronous ROM
`csdpram`	Dual-port RAM
`csfifo`	FIFO buffer
`altdpram`	Dual-port RAM
`scfifo`	Single-clock FIFO
`dcfifo`	Dual-clock FIFO

EABs are optimized for memory functions, and therefore free up LEs for other logic functions.

For more information on the use of EABs, go to the *FLEX 10K Embedded Programmable Logic Family Data Sheet*.

Figure 26 on page 33 shows a sample instantiation of a memory function.

Using EABs for Logic

The FLEX 10K EAB can be used to implement combinatorial logic. Effectively, the EAB becomes a large LUT.

In FLEX 10K devices, one EAB occupies the same die area and costs as much as 20 LEs. Thus, one EAB can perform, in one logic level, complex functions might otherwise require more than 20 LEs.

Table 3 shows how functions of *X* inputs and *Y* outputs can be implemented in one EAB.

Table 3. X Inputs & Y Outputs Implemented in One EAB

X Inputs	Y Outputs
8	8
9	4
10	2
11	1

Additionally, EABs can be cascaded and combined with LEs for larger functions (e.g., an 8×8 multiplier in an EPF10K20-3 device). Using EABs for logic saves 27 LEs without increasing timing delays (1 EAB = 20 LEs). See Table 4.

Table 4. Using EABs vs. LEs

Implementation	Logic Consumed	Longest Delay
LEs alone	136 LEs	40.0 ns
Using EABs	4 EABS, 29 LEs	39.0 ns

Refer to Figures 16 through 19 for details on a sample instantiation of a state machine in an EAB.

Conclusion

Design time and performance are valuable commodities in the programmable logic industry. This application note demonstrates various techniques to help you achieve performance goals and save design time by streamlining your design. By using VHDL and Verilog HDL coding techniques, Synplify software constraints, and MAX+PLUS II software options, you can improve performance in FLEX 10K devices and ultimately improve your overall design.

Notes:

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 544-7104
Literature Services:
(888) 3-ALTERA
lit_req@altera.com

I.S. EN ISO 9001

Printed on Recycled Paper.