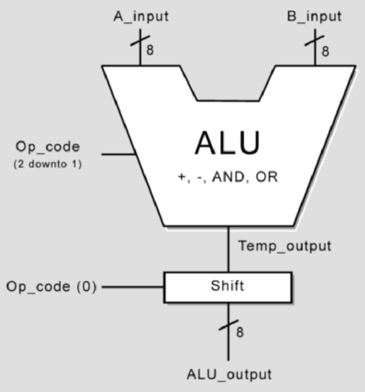
CHAPTER 6

Using VHDL for Synthesis of Digital Hardware

```
ARCHITECTURE behavior OF ALU IS
SIGNAL temp_output: STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN
PROCESS (Op_code, A_input, B_input)
        BEGIN
        -- Select Arithmetic/Logical Operation
        CASE Op_Code ( 2 DOWNTO 1 ) IS
                WHEN "00" =>
                   temp_output <= A_input + B_input;
                WHEN "01" =>
                   temp_output <= A_input - B_input;
                WHEN "10" =>
                   temp_output <= A_input AND B_input;
                WHEN "11" =>
                   temp_output <= A_input OR B_input;
                WHEN OTHERS =>
                   temp_output <= "00000000";
        END CASE;
        -- Select Shift Operation
        IF Op_Code( 0 ) = '1' THEN
                Alu_output <= temp_output( 6 DOWNTO 0 ) & '0';
        ELSE
                Alu_output <= temp_output;
        END IF;
END PROCESS;
END behavior;
```



6 Using VHDL for Synthesis of Digital Hardware

In the past, most digital designs were manually entered into a schematic entry tool. With increasingly large and more complex designs, this is a tedious and time-consuming process. Logic synthesis using hardware description languages is becoming widely used since it greatly reduces development time and cost. It also enables more exploration of design alternatives, more flexibility to changes in the hardware technology, and promotes design reuse.

VHDL is a language widely used to model and design digital hardware. VHDL is the subject of IEEE standards 1076 and 1164 and is supported by numerous CAD tool and programmable logic vendors. VHDL is an acronym for VHSIC Hardware Description Language. VHSIC, Very High Speed Integrated Circuits, was a USA Department of Defense program in the 1980s that sponsored the early development of VHDL. VHDL has syntax similar to ADA and PASCAL.

Conventional programming languages are based on a sequential operation model. Digital hardware devices by their very nature operate in parallel. This means that conventional programming languages cannot accurately describe or model the operation of digital hardware since they are based on the sequential execution of statements. VHDL is designed to model parallel operations.

IT IS **CRUCIAL** TO REMEMBER THAT VHDL MODULES, CONCURRENT STATEMENTS, AND PROCESSES ALL OPERATE IN PARALLEL.

In VHDL, variables change without delay and signals change with a small delay. For VHDL synthesis, signals are normally used instead of variables so that simulation works the same as the synthesized hardware.

A subset of VHDL is used for logic synthesis. In this section, a brief introduction to VHDL for logic synthesis will be presented. It is assumed that the reader is already familiar with basic digital logic devices and PASCAL, ADA, or VHDL.

Whenever you need help with VHDL syntax, VHDL templates of common statements are available in the Quartus II online help. In the text editor, just click the right mouse button and **Insert** ⇨ **Templates** and **select VHDL**.

6.1 VHDL Data Types

In addition to the normal language data types such as Boolean, integer, and real, VHDL contains new types useful in modeling digital hardware. For logic synthesis, the most important type is standard logic. Type standard logic, STD_LOGIC, is normally used to model a logic bit. To accurately model the operation of digital circuits, more values than "0" or "1" are needed for a logic bit. In the logic simulator, a standard logic bit can have nine values, U, X, 0, 1, Z, W, L, H, and "-". U is uninitialized and X is forced unknown. Z is tri-state or high impedance. L and H are weak "0" and weak "1". "-" is don’t care. Type STD_LOGIC_VECTOR contains a one-dimensional array of STD_LOGIC bits. Using these types normally requires the inclusion of special standard logic libraries at the beginning of each VHDL module. The value of a standard logic bit can be set to ‘0’ or ‘1’ using single quotes. A standard logic vector constant, such as the 2-bit zero value, "00" must be enclosed in double quotes. X”F” is the four bit hexadecimal value F.

6.2 VHDL Operators

Table 6.1 lists the VHDL operators and their common function in VHDL synthesis tools.

Table 6.1 VHDL Operators.

VHDL Operator	Operation
+	Addition
-	Subtraction
*	Multiplication*
/	Division*
MOD	Modulus*
REM	Remainder*
&	Concatenation – used to combine bits
SLL**	logical shift left
SRL**	logical shift right
SLA**	arithmetic shift left
SRA**	arithmetic shift right
ROL**	rotate left
ROR**	rotate right
=	equality
/=	Inequality
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
NOT	logical NOT
AND	logical AND
OR	logical OR
NAND	logical NAND
NOR	logical NOR
XOR	logical XOR
XNOR*	logical XNOR

*Not supported in many VHDL synthesis tools. In the Quartus II tools, only multiply and divide by integers are supported. Mod and Rem are not supported in Quartus II. Efficient design of multiply or divide hardware may require the user to specify the arithmetic algorithm and design in VHDL.

** Supported only in 1076-1993 VHDL only.

Table 6.2 illustrates two useful VHDL conversion functions for type STD_LOGIC and integer.

Table 6.2 STD_LOGIC conversion functions.

Function	Example:
CONV_STD_LOGIC_VECTOR(*integer, bits*)	**CONV_STD_LOGIC_VECTOR**(7, 4)
Converts an integer to a standard logic vector. Useful to enter constants. **CONV_SIGNED** and **CONV_UNSIGNED** work in a similar way to produce signed and unsigned values.	Produces a standard logic vector of "0111".
CONV_INTEGER(*std_logic_vector*)	**CONV_INTEGER**("0111")
Converts a standard logic vector to an integer. Useful for array indexing when using a std_logic_vector signal for the array index.	Produces an integer value of 7.

6.3 VHDL Based Synthesis of Digital Hardware

VHDL can be used to construct models at a variety of levels such as structural, behavioral, register transfer level (RTL), and timing. An RTL model of a circuit described in VHDL describes the input/output relationship in terms of dataflow operations on signal and register values. If registers are required, a synchronous clocking scheme is normally used. Sometimes an RTL model is also referred to as a dataflow-style model.

VHDL simulation models often include physical device time delays. In VHDL models written for logic synthesis, timing information should not be provided.

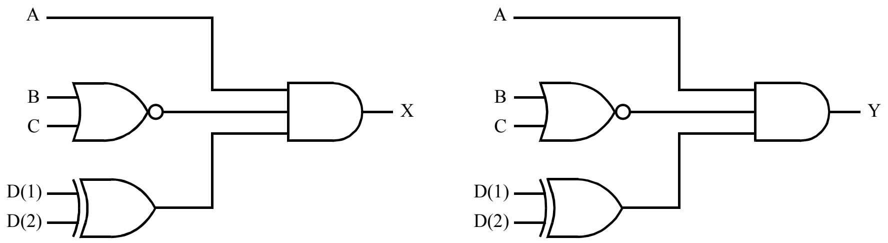
For timing simulations, the CAD tools automatically include the actual timing delays for the synthesized logic circuit. A FPGA timing model supplied by the CAD tool vendor is used to automatically generate the physical device time delays inside the FPGA. Sometimes this timing model is also written in VHDL. For a quick overview of VHDL, several constructs that can be used to synthesize common digital hardware devices will be presented.

6.4 VHDL Synthesis Models of Gate Networks

The first example consists of a simple gate network. In this model, both a concurrent assignment statement and a sequential process are shown which generate the same gate network. X is the output on one network and Y is the output on the other gate network. The two gate networks operate in parallel.

In VHDL synthesis, inputs and outputs from the port declaration in the module will become I/O pins on the programmable logic device. Comment lines begin with "--". The Quartus II editor performs syntax coloring and is useful to quickly find major problems with VHDL syntax.

Inside a process, statements are executed in sequential order, and all processes are executed in parallel. If multiple assignments are made to a signal inside a process, the last assignment is taken as the new signal value.

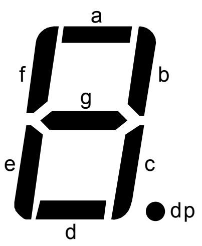


```
LIBRARY IEEE;                        -- Include Libraries for standard logic data types
USE  IEEE.STD_LOGIC_1164.ALL;
                                     -- Entity name normally the same as file name
ENTITY gate_network IS               -- Ports: Declares module inputs and outputs
        PORT( A, B, C  : IN     STD_LOGIC;
                                     -- Standard Logic Vector ( Array of 4 Bits )
                D      : IN     STD_LOGIC_VECTOR( 3 DOWNTO 0 );
                                     -- Output Signals
                X, Y   : OUT    STD_LOGIC );
END gate_network;

                                -- Defines internal module architecture
ARCHITECTURE behavior OF gate_network IS
BEGIN                           -- Concurrent assignment statements operate in parallel
                                -- D(1) selects bit 1 of standard logic vector D
       X <= A AND NOT( B OR C ) AND ( D( 1 ) XOR D( 2 ) );

                                -- Process must declare a sensitivity list,
                                -- In this case it is  ( A, B, C, D )
                                -- List includes all signals that can change the outputs
   PROCESS ( A, B, C, D )
        BEGIN                   -- Statements inside process execute sequentially
                Y <= A AND NOT( B OR C) AND ( D( 1) XOR D( 2 ) );
   END PROCESS;
END behavior;
```

6.5 VHDL Synthesis Model of a Seven-segment LED Decoder

The following VHDL code implements a seven-segment decoder for seven-segment LED displays. A 7-bit standard logic vector is used to assign the value of all seven bits in a single case statement. In the logic vector, the most-significant bit is segment ‘a’ and the least-significant bit is segment ‘g’. The logic synthesis CAD tool automatically minimizes the logic required for implementation. The signal MSD contains the 4-bit binary value to be displayed in hexadecimal. MSD is the left or most-significant digit. Another identical process with a different input variable is needed for the second display digit.

```
LED_MSD_DISPLAY:                  -- BCD to 7 Segment Decoder for LED Displays

PROCESS  (MSD)
BEGIN
                                  -- Case statement implements a logic truth table
  CASE MSD IS
       WHEN "0000" =>
                 MSD_7SEG <= "1111110";
       WHEN "0001" =>
                 MSD_7SEG <= "0110000";
       WHEN "0010" =>
                 MSD_7SEG <= "1101101";
       WHEN "0011" =>
                 MSD_7SEG <= "1111001";
       WHEN "0100" =>
                 MSD_7SEG <= "0110011";
       WHEN "0101" =>
                 MSD_7SEG <= "1011011";
       WHEN "0110" =>
                 MSD_7SEG <= "1011111";
       WHEN "0111" =>
                 MSD_7SEG <= "1110000";
       WHEN "1000" =>
                 MSD_7SEG <= "1111111";
       WHEN "1001" =>
                 MSD_7SEG <= "1111011";
       WHEN OTHERS =>
                 MSD_7SEG <= "0111110";
  END CASE;

END PROCESS LED_MSD_DISPLAY;
```



The following VHDL concurrent assignment statements provide the value to be displayed and connect the individual segments. NOT is used since a logic zero actually turns on the LED. Automatic minimization in the synthesis process will eliminate the extra inverter in the logic circuit. Pin assignments for the seven-segment display must be included in the project's *.qsf file or in the top-level schematic.

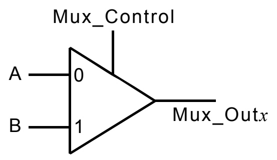
```
                                  -- Provide 4-bit value to display
  MSD <= PC ( 7 DOWNTO 4 );
                                  -- Drive the seven-segments (LEDs are active low)
  MSD_a <= NOT MSD_7SEG( 6 );
  MSD_b <= NOT MSD_7SEG( 5 );
  MSD_c <= NOT MSD_7SEG( 4 );
  MSD_d <= NOT MSD_7SEG( 3 );
  MSD_e <= NOT MSD_7SEG( 2 );
```

```
        MSD_f  <= NOT MSD_7SEG( 1 );
        MSD_g <= NOT MSD_7SEG( 0 );
```

6.6 VHDL Synthesis Model of a Multiplexer

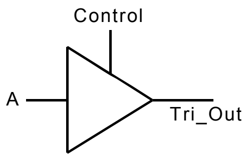
The next example shows several alternative ways to synthesize a 2-to-1 multiplexer in VHDL. Four identical multiplexers that operate in parallel are synthesized by this example. In VHDL, IF and CASE statements must be inside a process. The inputs and outputs from the multiplexers could be changed to standard logic vectors if an entire bus is multiplexed. Multiplexers with more than two inputs can also be easily constructed. Nested IF-THEN-ELSE statements generate priority-encoded logic that requires more hardware and produce a slower circuit than a CASE statement.

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;

ENTITY multiplexer IS                          -- Input Signals and Mux Control
        PORT( A, B, Mux_Control        : IN    STD_LOGIC;
              Mux_Out1, Mux_Out2,
              Mux_Out3, Mux_Out4       : OUT   STD_LOGIC  );
END multiplexer;

ARCHITECTURE behavior OF multiplexer IS
BEGIN                                          -- selected signal assignment statement…

        Mux_Out1 <= A WHEN Mux_Control = '0' ELSE B;
                                               -- … with Select Statement
        WITH Mux_control SELECT

        Mux_Out2 <=    A WHEN    '0',
                       B WHEN    '1',
                       A WHEN OTHERS;          -- OTHERS case required since STD_LOGIC
                                               --     has values other than "0" or "1"
        PROCESS ( A, B, Mux_Control)
        BEGIN                                  -- Statements inside a process
                IF Mux_Control = '0' THEN      --     execute sequentially.
                  Mux_Out3 <= A;
                ELSE
                  Mux_out3 <= B;
                END IF;

                CASE Mux_Control IS
                        WHEN '0' =>
                                Mux_Out4 <= A;
                        WHEN '1' =>
                                Mux_Out4 <= B;
                        WHEN OTHERS =>
                                Mux_Out4 <= A;
                END CASE;
        END PROCESS;
END behavior;
```



6.7 VHDL Synthesis Model of Tri-State Output

Tri-state gates are supported in VHDL synthesis tools and are supported in many programmable logic devices. Most programmable logic devices have tri-state output pins. Some programmable logic devices do not support internal tri-state logic. Here is a VHDL example of a tri-state output. In VHDL, the assignment of the value "Z" to a signal produces a tri-state output.



```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;

ENTITY tristate IS
   PORT( A, Control      : IN       STD_LOGIC;
         Tri_out         : INOUT    STD_LOGIC); -- Use Inout for bi-directional tri-state
                                                --     signals or out for output only
END tristate;

ARCHITECTURE behavior OF tristate IS              -- defines internal module architecture
BEGIN
   Tri_out <= A WHEN Control = '0' ELSE 'Z';      -- Assignment of 'Z' value generates
END behavior;                                     --    tri-state output
```

6.8 VHDL Synthesis Models of Flip-flops and Registers

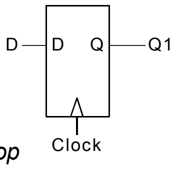
In the next example, several flip-flops will be generated. Unlike earlier combinational hardware devices, a flip-flop can only be synthesized inside a process. In VHDL, Clock’EVENT is true whenever the clock signal changes. The positive clock edge is selected by (clock’EVENT AND clock = ‘1’) and positive edge triggered D flip-flops will be used for synthesis.

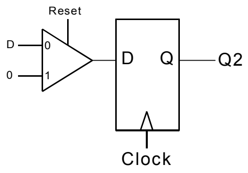
The following module contains a variety of Reset and Enable options on positive edge-triggered D flip-flops. Processes with a wait statement do not need a process sensitivity list. A process can only have one clock or reset type.

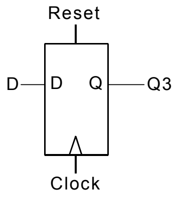
The negative clock edge is selected by (clock’EVENT AND clock = ‘0’) and negative edge-triggered D flip-flops will be generated during synthesis. If (Clock = ’1’) is substituted for (clock’EVENT AND clock = ‘1’) level-triggered latches will be selected for logic synthesis. Rising_edge(clock) can also be used instead of clock’EVENT AND clock = ‘1’. Falling_edge(clock) is also supported for negative clock edges.

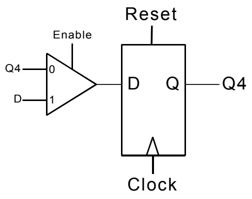
```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
ENTITY DFFs IS
      PORT( D, Clock, Reset, Enable   : IN    STD_LOGIC;
                Q1, Q2, Q3, Q4        : OUT  STD_LOGIC  );
END DFFs;
```

```
ARCHITECTURE behavior OF DFFs IS
BEGIN


    PROCESS                              -- Positive edge triggered D flip-flop
    BEGIN                                -- If WAIT is used no sensitivity list is used
        WAIT UNTIL ( Clock 'EVENT AND Clock = '1' );
            Q1 <= D;
    END PROCESS;


    PROCESS                              -- Positive edge triggered D flip-flop
    BEGIN                                --     with synchronous reset
        WAIT UNTIL ( Clock 'EVENT AND Clock = '1' );
                IF reset = '1'  THEN
                    Q2 <= '0';
                ELSE
                    Q2 <= D;
                END IF;
    END PROCESS;


    PROCESS (Reset,Clock)                -- Positive edge triggered D flip-flop
    BEGIN                                --     with asynchronous reset
        IF reset = '1' THEN
            Q3 <= '0';
        ELSIF ( clock 'EVENT AND clock = '1' ) THEN
            Q3 <= D;
        END IF;
    END PROCESS;

    PROCESS (Reset,Clock)                -- Positive edge triggered D flip-flop
    BEGIN                                --     with asynchronous reset and
                                         --     enable
        IF reset = '1' THEN
            Q4 <= '0';
        ELSIF ( clock 'EVENT AND clock = '1' ) THEN
             IF Enable = '1' THEN
                Q4 <= D;
             END IF;
        END IF;
    END PROCESS;
END behavior;
```









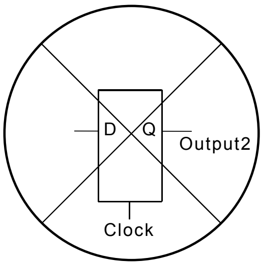
In VHDL, as in any digital logic designs, it is not good design practice to AND or gate other signals with the clock. Use a flip-flop with a clock enable instead to avoid timing and clock skew problems. In some limited cases, such as power management, a single level of clock gating can be used. This works only when a small amount of clock skew can be tolerated and the signal gated with the clock is known to be hazard or glitch free. A particular programmable logic device may not support every flip-flop or latch type and Set/Reset and Enable option.

If D and Q are replaced by standard logic vectors in these examples, registers with the correct number of bits will be generated instead of individual flip-flops.

6.9 Accidental Synthesis of Inferred Latches

Here is a very common problem to be aware of when coding VHDL for synthesis. If a non-clocked process has any path that does not assign a value to an output, VHDL assumes you want to use the previous value. A level triggered latch is automatically generated or inferred by the synthesis tool to save the previous value. In many cases, this can cause serious errors in the design. Edge-triggered flip-flops should not be mixed with level-triggered latches in a design, or serious timing problems will result. Typically this can happen in CASE statements or nested IF statements. In the following example, the signal OUTPUT2 infers a latch when synthesized. Assigning a value to OUTPUT2 in the last ELSE clause will eliminate the inferred latch.

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
ENTITY ilatch IS
        PORT( A, B                    : IN     STD_LOGIC;
              Output1, Output2        : OUT    STD_LOGIC  );
END ilatch;

ARCHITECTURE behavior OF ilatch IS
BEGIN
        PROCESS ( A, B )
        BEGIN
                IF A = '0' THEN
                   Output1 <= '0';
                   Output2 <= '0';
                ELSE
                   IF B = '1' THEN
                      Output1 <= '1';
                      Output2 <= '1';
                   ELSE                   -- Latch inferred since no value is assigned
                      Output1 <= '0';     --    to output2 in the else clause!
                   END IF;
                END IF;
        END PROCESS;
END behavior;
```



6.10 VHDL Synthesis Model of a Counter

Here is an 8-bit counter design. This design performs arithmetic operations on standard logic vectors. Since this example includes arithmetic operations, two new libraries must be included at the beginning of the module. Either signed or unsigned libraries can be selected, but not both. Since the unsigned library was used, an 8-bit magnitude comparator is automatically synthesized for the internal_count < max_count comparison.

Compare operations between standard logic and integer types are supported. The assignment internal_count <= internal_count + 1 synthesizes an 8-bit incrementer. An incrementer circuit requires less hardware than an adder that adds one. The operation, "+1", is treated as a special incrementer case by synthesis tools.

VHDL does not allow reading of an "OUT" signal so an internal_count signal is used which is always the same as count. This is the first example that includes an internal signal. Note its declaration at the beginning of the architecture section.

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY Counter IS
    PORT( Clock, Reset   :  IN    STD_LOGIC;
          Max_count      :  IN    STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          Count          :  OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 )  );
END Counter;

ARCHITECTURE behavior OF Counter IS              -- Declare signal(s) internal to module
    SIGNAL internal_count:         STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN
    count <= internal_count;

    PROCESS ( Reset,Clock )
        BEGIN                                    -- Reset counter
            IF reset = '1' THEN
              internal_count <= "00000000";
            ELSIF ( clock 'EVENT AND clock = '1' ) THEN
              IF internal_count < Max_count THEN     -- Check for maximum count
                internal_count <= internal_count + 1;   -- Increment Counter
              ELSE                                   -- Count >=  Max_Count
                internal_count <= "00000000";          --      reset Counter
              END IF;
            END IF;
    END PROCESS;
END behavior;
```

6.11 VHDL Synthesis Model of a State Machine

The next example is a Moore state machine with three states, two inputs and a single output. A state diagram of the example state machine is shown in Figure 6.1. In VHDL, an enumerated data type is specified for the current state using the TYPE statement. This allows the synthesis tool to assign the actual "0" or "1" values to the states. In many cases, this will produce a smaller hardware design than direct assignment of the state values in VHDL.

Depending on the synthesis tool settings, the states may be encoded or constructed using the one-hot technique. Outputs are defined in the last WITH… SELECT statement. This statement lists the output for each state and eliminates possible problems with inferred latches. To avoid possible timing problems, unsynchronized external inputs to a state machine should be synchronized by passing them through one or two D flip-flops that are clocked by the state machine's clock.

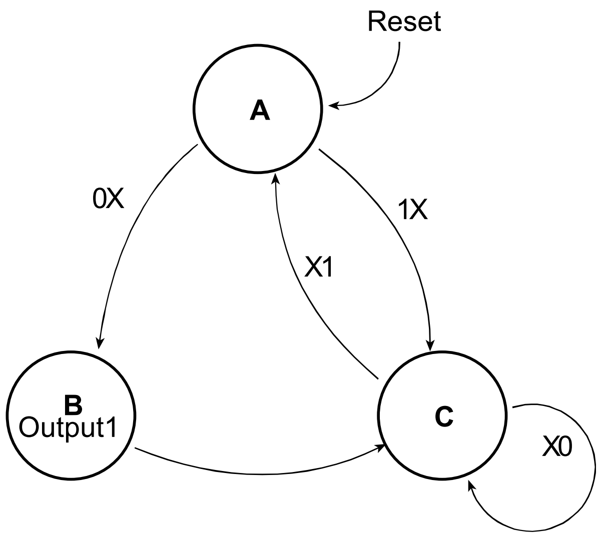


Figure 6.1 State Diagram for st_mach VHDL example

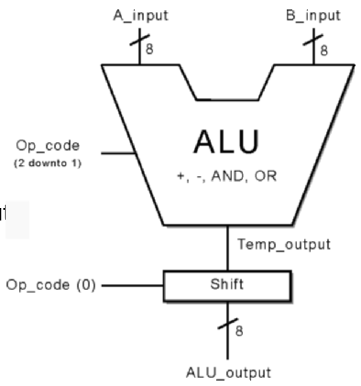
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY st_mach IS
    PORT( clk, reset        : IN     STD_LOGIC;
          Input1, Input2    : IN     STD_LOGIC;
          Output1           : OUT    STD_LOGIC);
END st_mach;

ARCHITECTURE A OF st_mach IS
                                          -- Enumerated Data Type for State
    TYPE STATE_TYPE IS ( state_A, state_B, state_C );
    SIGNAL state: STATE_TYPE;

BEGIN
    PROCESS ( reset, clk )
    BEGIN
        IF reset = '1' THEN               -- Reset State
          state <= state_A;
        ELSIF clk 'EVENT AND clk = '1' THEN
        CASE state IS                       -- Define Next State Transitions using a Case
                                            --      Statement based on the Current State
            WHEN state_A =>
                IF Input1 = '0' THEN
                    state <= state_B;
                ELSE
                    state <= state_C;
                END IF;

            WHEN state_B =>
                state <= state_C;

            WHEN state_C =>
                IF Input2 = '1' THEN
                    state <= state_A;
                END IF;

            WHEN OTHERS =>
                state <= state_A;
        END CASE;
    END IF;
  END PROCESS;

  WITH state SELECT                         -- Define State Machine Outputs
    Output1   <=   '0' WHEN state_A,
                   '1' WHEN state_B,
                   '0' WHEN state_C;
END a;
```

6.12 VHDL Synthesis Model of an ALU with an Adder/Subtractor and a Shifter

Here is an 8-bit arithmetic logic unit (ALU) that adds, subtracts, bitwise ANDs, or bitwise ORs, two operands and then performs an optional shift on the output. The most-significant two bits of the Op-code select the arithmetic logical operation. If the least-significant bit of the op_code equals '1' a 1-bit left-shift operation is performed. An addition and subtraction circuit is synthesized for the "+" and "-" operator. Depending on the number of bits and the speed versus area settings in the synthesis tool, ripple carry or carry-lookahead circuits will be used. Several "+" and "-" operations in multiple assignment statements may generate multiple ALUs and increase the hardware size, depending on the VHDL CAD tool and compiler settings used. If a single ALU is desired, muxes can be placed at the inputs and the "+" operator would be used only in a single assignment statement.

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY ALU IS
```

```
    PORT( Op_code              : IN    STD_LOGIC_VECTOR( 2 DOWNTO 0 );
          A_input, B_input     : IN    STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          ALU_output           : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0 ) );
END ALU;

ARCHITECTURE behavior OF ALU IS
                    -- Declare signal(s) internal to module here
    SIGNAL temp_output              :      STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN

    PROCESS ( Op_code, A_input, B_input )
    BEGIN
                    --Select Arithmetic/Logical Operation
        CASE Op_Code (  2 DOWNTO 1 ) IS
            WHEN "00" =>
                    temp_output <= A_input   +  B_input;
            WHEN "01" =>
                    temp_output <= A_input  -  B_input;
            WHEN "10" =>
                    temp_output <= A_input   AND B_inpu
            WHEN "11" =>
                    temp_output <= A_input   OR B_input;
            WHEN OTHERS =>
                    temp_output <= "00000000";
        END CASE;

        -- Select Shift Operation: Shift bits left with zero fill using concatenation operator
        --    Can also use VHDL 1076-1993 shift operator such as SLL

        IF Op_Code( 0 ) = '1' THEN
            Alu_output <= temp_output( 6 DOWNTO 0 ) & '0';
        ELSE
            Alu_output <= temp_output;
        END IF;
    END PROCESS;
END behavior;
```



6.13 VHDL Synthesis of Multiply and Divide Hardware

In the Quartus II tool, integer multiply and divide is supported using VHDL's "*" and "/" operators. Mod and Rem are not supported in Quartus II. In current generation tools, efficient design of multiply or divide hardware typically requires the use of a vendor-specific library function or even the specification of the arithmetic algorithm and hardware implementation in VHDL.

A wide variety of multiply and divide algorithms that trade off time versus hardware size can be found in most computer arithmetic texts. Several such references are listed at the end of this chapter. These algorithms require a sequence of add/subtract and shift operations that can be easily synthesized in VHDL using the standard operators. The LPM_MULT function in Quartus II can be used to synthesize integer multipliers. LPM_DIVIDE, is also available. When using LPM functions, **Tools ⇨ MegaWizard Plug-in Manager** can be used to help generate VHDL code. The LPM functions also support pipeline options. Array multiply and divide hardware for more than a few bits requires extensive hardware and a large FPGA.

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY mult IS
      PORT( A, B       : IN   STD_LOGIC_VECTOR(  7 DOWNTO 0 );
              Product  : OUT STD_LOGIC_VECTOR( 15 DOWNTO 0 ) );
END mult;

ARCHITECTURE a OF mult IS
BEGIN                                          -- LPM 8x8 multiply function P = A * B
      multiply: lpm_mult
      GENERIC MAP(  LPM_WIDTHA               =>  8,
                    LPM_WIDTHB               =>  8,
                    LPM_WIDTHS               => 16,
                    LPM_WIDTHP               => 16,
                    LPM_REPRESENTATION       =>   "UNSIGNED" )

      PORT MAP (    data   => A,
                    datab => B,
                    result => Product );
END a;
```

Floating-point operations can be implemented on very large FPGAs; however, performance is lower than current floating-point DSP and microprocessor chips. The floating-point algorithms must be coded by the user in VHDL using integer add, multiply, divide, and shift operations. The LPM_CLSHIFT function is useful for the barrel shifter needed in a floating-point ALU. Some floating point IP cores are starting to appear. Many FPGA vendors also have optimized arithmetic packages for DSP applications such as FIR filters.

6.14 VHDL Synthesis Models for Memory

Typically, it is more efficient to call a vendor-specific function to synthesize RAM. These functions typically use the FPGA's internal RAM blocks rather than building a RAM using FPGA logic elements. The memory function in the Altera toolset is the ALTSYNCRAM function. On the UP2 board's older FPGA, the LPM_RAM_DQ memory function can also be used. The memory can be set to an initial value using a separate memory initialization file with the extension *.mif. A similar call, LPM_ROM, can be used to synthesize ROM.

If small blocks of multi-ported or other special-purpose RAM are needed, they can be synthesized using registers with address decoders for the write operation and multiplexers for the read operation. Additional read or write ports can be added to synthesized RAM. An example of this approach is a dual-ported register file for a computer processor core. Most RISC processors need to read two registers on each clock cycle and write to a third register.

VHDL Memory Model - Example One

The first memory example synthesizes a memory that can perform a read and a write operation every clock cycle. Memory is built using arrays of positive edge-triggered D flip-flops. Memory write, memwrite, is gated with an address decoder output and used as an enable to load each memory location during a write operation. A synchronous write operation is more reliable. Asynchronous write operations respond to any logic hazards or momentary level changes on the write signal. As in any synchronous memory, the write address must be stable before the rising edge of the clock signal. A non-clocked mux is used for the read operation. If desired, memory can be initialized by a reset signal.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY memory IS
    PORT( read_data     :   OUT  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          read_address  :   IN   STD_LOGIC_VECTOR( 2 DOWNTO 0 );
          write_data    :   IN   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          write_address :   IN   STD_LOGIC_VECTOR( 2 DOWNTO 0 );
          Memwrite      :   IN   STD_LOGIC;
          clock,reset   :   IN   STD_LOGIC  );
END memory;

ARCHITECTURE behavior OF memory IS
    SIGNAL mem0, mem1  :         STD_LOGIC_VECTOR( 7 DOWNTO 0 );

BEGIN
    PROCESS (read_address, mem0, mem1)  -- Process for memory read operation
    BEGIN
        CASE read_address IS
            WHEN "000" =>
                    read_data <= mem0;
            WHEN "001" =>
                    read_data <= mem1;
            WHEN OTHERS =>              -- Unimplemented memory locations
                    read_data <= X"FF" ;
        END CASE;
    END PROCESS;

    PROCESS
    BEGIN
        WAIT UNTIL clock 'EVENT AND clock = '1';
            IF ( reset = '1' ) THEN
                mem0 <= X"55" ; -- Initial values for memory (optional)
                mem1 <= X"AA" ;
            ELSE
```

```
                IF memwrite = '1' THEN              -- Write to memory?
                CASE write_address IS               -- Use a flip-flop with
                    WHEN "000" =>                   --    an enable for memory
                        mem0 <= write_data;
                    WHEN "001" =>
                        mem1 <= write_data;
                    WHEN OTHERS =>                  -- unimplemented memory locations
                        NULL;
                END CASE;
                END IF;
            END IF;
    END PROCESS;
END behavior;
```

VHDL Memory Model - Example Two

The second example uses an array of standard logic vectors to implement memory. This approach is easier to write in VHDL since the array index generates the address decoder and multiplexers automatically; however, it is a little more difficult to access the values of individual array elements during simulation. There are a few VHDL synthesis tools that do not support array types. Synthesizing RAM requires a vast amount of programmable logic resources. Only a few hundred bits of RAM can be synthesized, even on large devices. Each bit of RAM requires 10 to 20 logic gates and a large amount of FPGA interconnect resources. Some tools may automatically detect synthesized RAM and use the FPGA's embedded memory blocks.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY memory IS
    PORT( read_data     :   OUT  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          read_address  :   IN   STD_LOGIC_VECTOR( 2 DOWNTO 0 );
          write_data    :   IN   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          write_address :   IN   STD_LOGIC_VECTOR( 2 DOWNTO 0 );
          Memwrite      :   IN   STD_LOGIC;
          Clock         :   IN   STD_LOGIC  );
END memory;
ARCHITECTURE behavior OF memory IS
                                        -- define new data type for memory array
    TYPE memory_type IS ARRAY ( 0 TO 7 ) OF STD_LOGIC_VECTOR( 7 DOWNTO 0 );
    SIGNAL memory          : memory_type;
BEGIN
    -- Read Memory and  convert array index to an integer with CONV_INTEGER
    read_data <= memory( CONV_INTEGER( read_address( 2 DOWNTO 0 ) ) );

    PROCESS                     -- Write Memory?
    BEGIN
        WAIT UNTIL clock 'EVENT AND clock = '1';
        IF ( memwrite = '1' ) THEN
    -- convert array index to an integer with CONV_INTEGER
            memory( CONV_INTEGER( write_address( 2 DOWNTO 0 ) ) ) <= write_data;
        END IF;
    END PROCESS;
END behavior;
```

VHDL Memory Model - Example Three

The third example shows the use of the ALTSYNCRAM megafunction to implement a block of memory. An additional library is needed for the megafunctions. (For more information on the megafunctions see the online help guide in the Quartus II tool.) In single port mode, the ALTSYNCRAM memory can do either a read or a write operation in a single clock cycle since there is only one address bus. In dual port mode, it can do both a read and write. If this is the only memory operation needed, the ALTSYNCRAM function produces a more efficient hardware implementation than synthesis of the memory in VHDL. In the ALTSYNCRAM megafunction, the memory address must be clocked into a dedicated address register located inside the FPGA's synchronous memory block. Asynchronous memory operations without a clock can cause timing problems and are not supported on many FPGAs including the Cyclone.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
LIBRARY Altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY amemory IS
    PORT( read_data        :    OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          memory_address   :    IN    STD_LOGIC_VECTOR( 2 DOWNTO 0 );
          write_data       :    IN    STD_LOGIC_VECTOR( 7 DOWNTO 0 );
          Memwrite         :    IN    STD_LOGIC;
          clock,reset      :    IN    STD_LOGIC );
END amemory;

ARCHITECTURE behavior OF amemory IS
BEGIN
    data_memory: altsyncram             -- Altsyncram memory function
    GENERIC MAP ( operation_mode => "SINGLE_PORT",
                  width_a           => 8,
                  widthad_a         => 3,
                  lpm_type => "altsyncram",
                  outdata_reg_a     => "UNREGISTERED",
                                    -- Reads in mif file for initial data values (optional)
                  init_file         => "memory.mif",
                  intended_device_family => "Cyclone"    )

    PORT MAP (wren_a => Memwrite,  clock0  => clock,
              address_a => memory_address( 2 DOWNTO 0 ),
              data_a => write_data,  q_a => read_data );
END behavior;
```

On the Cyclone FPGA chip, the memory can be implemented using the M4K memory blocks, which are separate from the FPGA's logic cells. In the Cyclone EP1C6 chip there are 20 M4K RAM blocks at 4Kbits each for a total of 92,160 bits. In the Cyclone EP1C12 there are 52 M4K blocks for a total of 239,616 bits. Each M4K block can be setup to be 4K by 1, 2K by 2, 1K by 4, 512 by 8, 256 by 16, 256 by 18, 128 by 32 or 128 by 36 bits wide. The **Tools ➪ MegaWizard Plug-in Manager** feature is useful to configure the Altsyncram parameters.

6.15 Hierarchy in VHDL Synthesis Models

Large VHDL models should be split into a hierarchy using a top-level structural model in VHDL or by using the symbol and graphic editor in the Quartus II tool. In the graphical editor, a VHDL file can be used to define the contents of a symbol block. Synthesis tools run faster using a hierarchy on large models and it is easier to write, understand, and maintain a large design when it is broken up into smaller modules.

An example of a hierarchical design with three submodules is seen in the schematic in Figure 6.2. Following the schematic, the same design using a top-level VHDL structural model is shown. This VHDL structural model provides the same connection information as the schematic seen in Figure 6.2.

Debounce, Onepulse, and Clk_div are the names of the VHDL submodules. Each one of these submodules has a separate VHDL source file. In the Quartus II tool, compiling the top-level module will automatically compile the lower-level modules.

In the example, VHDL structural-model example, note the use of a component declaration for each submodule. The component statement declares the module name and the inputs and outputs of the module. Internal signal names used for interconnections of components must also be declared at the beginning of the component list.

In the final section, port mappings are used to specify the module or component interconnections. Port names and their order must be the same in the VHDL submodule file, the component instantiations, and the port mappings. Component instantiations are given unique labels so that a single component can be used several times.

Note that node names in the schematic or signals in VHDL used to interconnect modules need not always have the same names as the signals in the components they connect. Just like signal or wire names in a schematic are not always the same as the pin names on chips that they connect. As an example, pb_debounced on the debounce component connects to an internal signal with a different name, pb1_debounced.

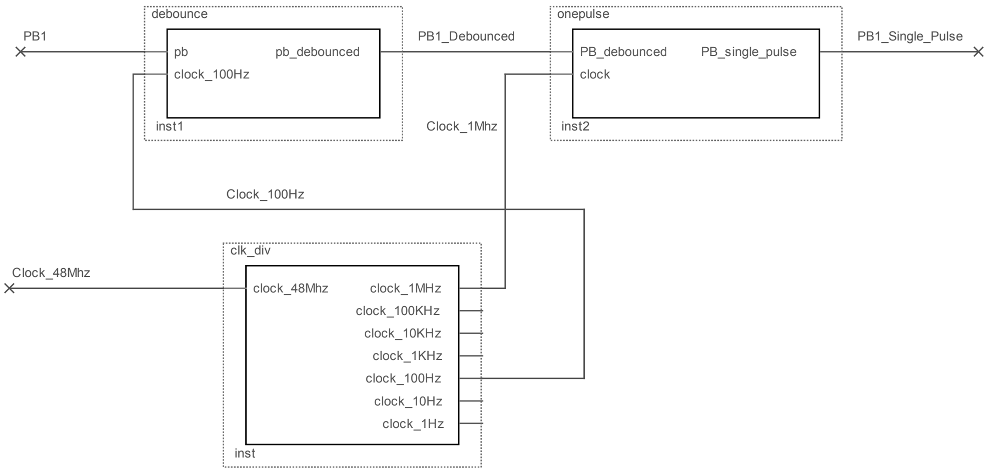


Figure 6.2 Schematic of Hierarchical Design Example

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY hierarch IS
    PORT (    clock_48MHz, pb1      : IN STD_LOGIC;
              pb1_single_pulse      : OUT STD_LOGIC);
END hierarch;
ARCHITECTURE structural OF hierarch IS
-- Declare internal signals needed to connect submodules
SIGNAL clock_1MHz, clock_100Hz, pb1_debounced : STD_LOGIC;
-- Use Components to Define Submodules and Parameters
    COMPONENT debounce
            PORT(    pb, clock_100Hz       : IN       STD_LOGIC;
                     pb_debounced          : OUT      STD_LOGIC);
    END COMPONENT;

    COMPONENT onepulse
            PORT(pb_debounced, clock       : IN       STD_LOGIC;
             pb_single_pulse               : OUT      STD_LOGIC);
    END COMPONENT;

    COMPONENT clk_div
            PORT(    clock_48MHz           : IN       STD_LOGIC;
                     clock_1MHz            : OUT      STD_LOGIC;
                     clock_100kHz          : OUT      STD_LOGIC;
                     clock_10kHz           : OUT      STD_LOGIC;
                     clock_1kHz            : OUT      STD_LOGIC;
                     clock_100Hz           : OUT      STD_LOGIC;
                     clock_10Hz            : OUT      STD_LOGIC;
                     clock_1Hz             : OUT      STD_LOGIC);
    END COMPONENT;
BEGIN
```

```
-- Use Port Map to connect signals between components in the hierarchy
debounce1 : debounce PORT MAP  (pb => pb1, clock_100Hz = >clock_100Hz,
                              pb_debounced = >pb1_debounced);

prescalar : clk_div  PORT MAP (clock_48MHz = >clock_48MHz,
                                    clock_1MHz =>clock_1MHz,
                            clock_100hz = >clock_100hz);

single_pulse : onepulse PORT MAP (pb_debounced = >pb1_debounced,
                              clock => clock_1MHz,
                               pb_single_pulse => pb1_single_pulse);
END structural;
```

6.16 Using a Testbench for Verification

Complex VHDL synthesis models are frequently verified by simulation of the model's behavior in a specially written entity called a testbench. As seen in Figure 6.3, the top–level testbench module contains a component instantiation of the hardware unit under test (UUT). The testbench also contains VHDL code used to automatically generate input stimulus to the UUT and automatically monitor the response of the UUT for correct operation.

The testbench contains test vectors and timing information used in testing the UUT. The testbench's VHDL code is used only for testing, and it is not synthesized. This keeps the test-only code portion of the VHDL model separate from the UUT's hardware synthesis model. Third party simulation tools such as ModelSIM or Active-HDL are typically required for this approach. Unfortunately, full versions of these third party simulation tools are currently very expensive for students or individuals.

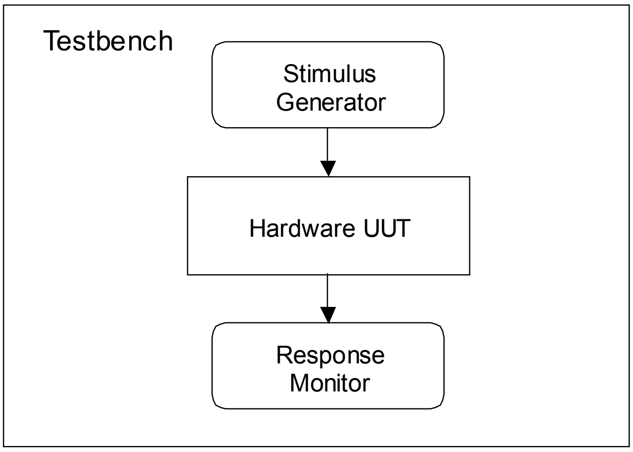


Figure 6.3 Using a testbench for automatic verification during simulation.

The testbench approach is critical in large ASIC designs where all errors are costly. Automatic Test Equipment (ATE) can also use a properly written testbench and its test vector and timing information to physically test each ASIC chip for correct operation after production. In large designs, the testbench can require as much time and effort as the UUT's synthesis model. By performing both a functional simulation and a timing simulation of the UUT with the same test vectors, it is also possible to check for any synthesis-related errors.

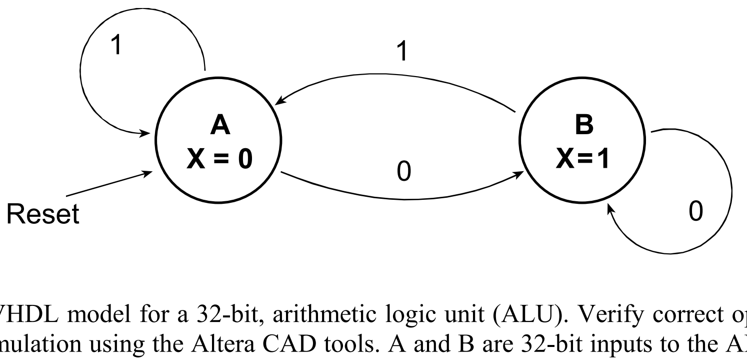
6.17 For additional information

The chapter has introduced the basics of using VHDL for digital synthesis. It has not explored all of the language options available. The Altera online help contains VHDL syntax and templates. A large number of VHDL reference textbooks are also available. Unfortunately, only a few of them currently examine using VHDL models that can be used for digital logic synthesis. One such text is *HDL Chip Design* by Douglas J. Smith, Doone Publications, 1996.

A number of alternative integer multiply, divide, and floating-point algorithms with different speed versus area tradeoffs can be found in computer arithmetic textbooks. Two such examples are *Digital Computer Arithmetic Design and Implementation* by Cavanagh, McGraw Hill, 1984, and *Computer Arithmetic Algorithms* by Israel Koren, Prentice Hall, 1993.

6.18 Laboratory Exercises

1. Rewrite and compile the VHDL model for the seven-segment decoder in Section 6.5 replacing the PROCESS and CASE statements with a WITH…SELECT statement.
2. Write a VHDL model for the state machine shown in the following state diagram and verify correct operation with a simulation using the Altera CAD tools. A and B are the two states, X is the output, and Y is the input. Use the timing analyzer to determine the maximum clock frequency on the Cyclone EP1C6Q240C8 device.



3. Write a VHDL model for a 32-bit, arithmetic logic unit (ALU). Verify correct operation with a simulation using the Altera CAD tools. A and B are 32-bit inputs to the ALU, and Y is the output. A shift operation follows the arithmetic and logical operation. The opcode controls ALU functions as follows:

Opcode	Operation	Function
000XX	ALU_OUT <= A	Pass A
001XX	ALU_OUT <= A + B	Add
010XX	ALU_OUT <= A-B	Subtract
011XX	ALU_OUT <= A AND B	Logical AND
100XX	ALU_OUT <= A OR B	Logical OR
101XX	ALU_OUT <= A + 1	Increment A
110XX	ALU_OUT <= A-1	Decrement A
111XX	ALU_OUT <= B	Pass B
XXX00	Y <= ALU_OUT	Pass ALU_OUT
XXX01	Y<= SHL(ALU_OUT)	Shift Left
XXX10	Y<=SHR(ALU_OUT)	Shift Right (unsigned-zero fill)
XXX11	Y<= 0	Pass 0's

4. Use the Cyclone chip as the target device. Determine the worst case time delay of the ALU using the timing analyzer. Examine the report file and find the device utilization. Use the logic element (LE) device utilization percentage found in the compilation report to compare the size of the designs.

5. Explore different synthesis options for the ALU from problem 3. Change the area and speed synthesis settings in the compiler under **Assignments ⇨Settings ⇨Analysis and Synthesis Settings**, rerun the timing analyzer to determine speed, and examine the report file for hardware size estimates. Include data points for the default, optimized for speed, balanced, and optimized for area settings. Build a plot showing the speed versus area trade-offs possible in the synthesis tool. Use the logic element (LE) device utilization percentage found in the compilation report to compare the size of the designs.

6. Develop a VHDL model of one of the TTL chips listed below. The model should be functionally equivalent, but there will be timing differences. Compare the timing differences between the VHDL FPGA implementation and the TTL chip. Use a data book or find a data sheet using the World Wide Web.

   A. 7400 Quad nand gate

   B. 74LS241 Octal buffer with tri-state output

   C. 74LS273 Octal D flip-flop with Clear

   D. 74163 4-bit binary counter

   E. 74LS181 4-bit ALU

7. Replace the 8count block used in the tutorial in Chapter 4, with a new counter module written in VHDL. Simulate the design and download a test program to the UP3 board.

8. Implement a 128 by 32 RAM using VHDL and the Altsyncram function. Do not use registered output options. Target the design to the Cyclone EP1C6240C8 device. Use the timing analyzer to determine the worst-case read and write access times for the memory.

9. Study the VHDL code in the LCD Display FPGAcore function and draw a state diagram of the initialization and data transfer operations and explain its operation. You may find it helpful to examine the data sheet for the LCD display's microcontroller.