Using Library Modules in VHDL Designs

This tutorial explains how Altera’s library modules can be included in VHDL-based designs, which are implemented by using the Quartus II software.

Contents:

Example Circuit
Library of Parameterized Modules
Augmented Circuit with an LPM
Results for the Augmented Design
Practical designs often include commonly used circuit blocks such as adders, subtractors, multipliers, decoders, counters, and shifters. Altera provides efficient implementations of such blocks in the form of library modules that can be instantiated in VHDL designs. The compiler may recognize that a standard function specified in VHDL code can be realized using a library module, in which case it may automatically infer this module. However, many library modules provide functionality that is too complex to be recognized automatically by the compiler. These modules have to be instantiated in the design explicitly by the user.

Quartus® II software includes a library of parameterized modules (LPM). The modules are general in structure and they are tailored to a specific application by specifying the values of general parameters.

Doing this tutorial, the reader will learn about:

- Library of parameterizes modules (LPMs)
- Configuring an LPM for use in a circuit
- Instantiating an LPM in a designed circuit

The detailed examples in the tutorial were obtained using the Quartus II version 7.1, but other versions of the software can also be used.

1 Example Circuit

As an example, we will use the adder/subtractor circuit shown in Figure 1. It can add, subtract, and accumulate \( n \)-bit numbers using the 2's complement number representation. The two primary inputs are numbers \( A = a_{n-1}a_{n-2}\cdots a_0 \) and \( B = b_{n-1}b_{n-2}\cdots b_0 \), and the primary output is \( Z = z_{n-1}z_{n-2}\cdots z_0 \). Another input is the \( AddSub \) control signal which causes \( Z = A + B \) to be performed when \( AddSub = 0 \) and \( Z = A - B \) when \( AddSub = 1 \). A second control input, \( Sel \), is used to select the accumulator mode of operation. If \( Sel = 0 \), the operation \( Z = A \pm B \) is performed, but if \( Sel = 1 \), then \( B \) is added to or subtracted from the current value of \( Z \). If the addition or subtraction operations result in arithmetic overflow, an output signal, \( Overflow \), is asserted.

To make it easier to deal with asynchronous input signals, they are loaded into flip-flops on a positive edge of the clock. Thus, inputs \( A \) and \( B \) will be loaded into registers \( Areg \) and \( Breg \), while \( Sel \) and \( AddSub \) will be loaded into flip-flops \( SelR \) and \( AddSubR \), respectively. The adder/subtractor circuit places the result into register \( Zreg \).
The required circuit is described by the VHDL code in Figure 2. For our example, we use a 16-bit circuit as specified by \( n = 16 \). Implement this circuit as follows:

- Create a project `addersubtractor`.
- Include a file `addersubtractor.vhd`, which corresponds to Figure 2, in the project. For convenience, this file is provided in the directory `DE2_tutorials/design_files`, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera’s DE2 web pages.
- Choose the Cyclone II EP2C35F672C6 device, which is the FPGA chip on Altera’s DE2 board.
- Compile the design.
- Simulate the design by applying some typical inputs.
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Top-level entity
ENTITY addersubtractor IS
    GENERIC ( n : INTEGER := 16 );
    PORT ( A, B : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
            Clock, Reset, Sel, AddSub : IN STD_LOGIC;
            Z : BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0);
            Overflow : OUT STD_LOGIC );
END addersubtractor ;

ARCHITECTURE Behavior OF addersubtractor IS
    SIGNAL G, H, M, Areg, Breg, Zreg, AddSubR_n : STD_LOGIC_VECTOR(n-1 DOWNTO 0);
    SIGNAL SelR, AddSubR, carryout, over_flow : STD_LOGIC;
    COMPONENT mux2to1
        GENERIC ( k : INTEGER := 8 ) ;
        PORT ( V, W : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0);
               Selm : IN STD_LOGIC ;
               F : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
    END COMPONENT ;
    COMPONENT adderk
        GENERIC ( k : INTEGER := 8 ) ;
        PORT ( carryin : IN STD_LOGIC ;
                X, Y : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0);
                S : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0);
                carryout : OUT STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    PROCESS ( Reset, Clock )
    BEGIN
        IF Reset = '1' THEN
            Areg <= (OTHERS => '0');
            Breg <= (OTHERS => '0');
            Zreg <= (OTHERS => '0');
            SelR <= '0';
            AddSubR <= '0';
            Overflow <= '0';
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Areg <= A;
            Breg <= B;
            Zreg <= M;
            SelR <= Sel;
            AddSubR <= AddSub;
            Overflow <= over_flow;
        END IF ;
    END PROCESS ;

    nbit_adder: adderk
        GENERIC MAP ( k => n ) ;
        PORT MAP ( AddSubR, G, H, M, carryout );
    multiplexer: mux2to1
        GENERIC MAP ( k => n ) ;
        PORT MAP ( Areg, Z, SelR, G ) ;
    AddSubR_n <= (OTHERS => AddSubR);
    H <= Breg XOR AddSubR_n ;
    over_flow <= carryout XOR G(n-1) XOR H(n-1) XOR M(n-1) ;
    Z <= Zreg ;
END Behavior;
    . . . continued in Part b

Figure 2. VHDL code for the circuit in Figure 1 (Part a).
-- k-bit 2-to-1 multiplexer
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux2to1 IS
    GENERIC ( k : INTEGER := 8 );
    PORT ( V, W : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0);
       Selm : IN STD_LOGIC;
       F : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) );
END mux2to1;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( V, W, Selm )
    BEGIN
    IF Selm = '0' THEN
        F <= V;
    ELSE
        F <= W;
    END IF;
    END PROCESS;
END Behavior;

-- k-bit adder
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY adderk IS
    GENERIC ( k : INTEGER := 8 );
    PORT ( carryin : IN STD_LOGIC;
       X, Y : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0);
       S : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0);
       carryout : OUT STD_LOGIC );
END adderk;

ARCHITECTURE Behavior OF adderk IS
    SIGNAL Sum : STD_LOGIC_VECTOR(k DOWNTO 0);
BEGIN
    Sum <= ('0' & X) + ('0' & Y) + carryin;
    S <= Sum(k-1 DOWNTO 0);
    carryout <= Sum(k);
END Behavior;

Figure 2. VHDL code for the circuit in Figure 1 (Part b).
The LPMs in the library of parameterized modules are general in structure and they can be configured to suit a specific application by specifying the values of various parameters. Select Help > Megafunctions/LPM to see a listing of the available LPMs. One of them is an adder/subtractor module called \textit{lpm\_add\_sub megafunction}. Select this module to see its description. The module has a number of inputs and outputs, some of which may be omitted in a given application. Several parameters can be defined to specify a particular mode of operation. For example, the number of bits in the operands is specified in the parameter \textit{LPM\_WIDTH}. The \textit{LPM\_REPRESENTATION} parameter specifies whether the operands are to be interpreted as signed or unsigned numbers, and so on. Templates on how an LPM can be instantiated in a hardware description language are given in the description of the module. Using these templates is somewhat cumbersome, so Quartus II software provides a wizard that makes the instantiation of LPMs easy.

We will use the \textit{lpm\_add\_sub} module to simplify our adder/subtractor circuit defined in Figures 1 and 2. The augmented circuit is given in Figure 3. The \textit{lpm\_add\_sub} module, instantiated under the name \textit{megaddsub}, replaces the adder circuit as well as the XOR gates that provide the input \textit{H} to the adder. Since arithmetic overflow is one of the outputs that the LPM provides, it is not necessary to generate this output with a separate XOR gate.

To implement this adder/subtractor circuit, create a new directory named \textit{tutorial\_lpm}, and then create a project \textit{addersubtractor2}. Choose the same Cyclone II EP2C35F672C6 device, to allow a direct comparison of implemented designs.
The new design will include the desired LPM subcircuit specified as a VHDL component that will be instantiated in the top-level VHDL design entity. The VHDL component for the LPM subcircuit is generated by using a wizard as follows:

1. Select Tools > MegaWizard Plug-in Manager, which leads to a sequence of seven pop-up boxes in which the user can specify the details of the desired LPM.

2. In the box shown in Figure 4 indicate Create a new custom megafuction variation and click Next.

![Figure 4. Choose to define an LPM.](image)

3. The box in Figure 5 provides a list of the available LPMs. Expand the “arithmetic” sublist and select LPM_ADD_SUB. Choose VHDL as the type of output file that should be created. The output file must be given a name; choose the name megaddsub.vhd and indicate that the file should be placed in the directory tutorial_lpm as shown in the figure. Press Next.

![Figure 5. Choose an LPM from the available library.](image)
4. In the box in Figure 6 specify that the width of the data inputs is 16 bits. Also, specify the operating mode in which one of the ports allows performing both addition and subtraction of the input operand, under the control of the $add_{sub}$ input. A symbol for the resulting LPM is shown in the top left corner. Note that if $add_{sub} = 1$ then $result = A + B$; otherwise, $result = A - B$. This interpretation of the control input and the operation performed is different from our original design in Figures 1 and 2, which we have to account for in the modified design. Observe that we have included this change in the circuit in Figure 3. Click Next.

Figure 6. Specify the size of data inputs.

5. In the box in Figure 7, specify that the values of both inputs may vary and click Next.

Figure 7. Further specification of inputs.
6. The box in Figure 8 allows the designer to indicate optional inputs and outputs that may be specified. Since we need the overflow signal, make the Create an overflow output choice and press Next.

Figure 8. Specify the Overflow output.

7. In the box in Figure 9 say No to the pipelining option and click Next.

Figure 9. Refuse the pipelining option.
8. Figure 10 shows the simulation model files needed to simulate the generated design. Press Next to proceed to the final page.

Figure 10. Simulation model files.

9. Figure 11 gives a summary which shows the files that the wizard will create. Press Finish to complete the process.

Figure 11. Files created by the wizard.
3 Augmented Circuit with an LPM

We will use the file `megaddsub.vhd` in our modified design. Figure 12 depicts the VHDL code in this file; note that we have not shown the comments in order to keep the figure small.

```vhdl
// Adder/subtractor module created by the MegaWizard
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY megaddsub IS
  PORT ( add_sub : IN STD_LOGIC ;
         dataa : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         datab : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
         result : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
         overflow : OUT STD_LOGIC );
END megaddsub;

ARCHITECTURE SYN OF megaddsub IS
  BEGIN
    overflow < = sub_wire0;
    result < = sub_wire1(15 DOWNTO 0);
    lpm_add_sub_component : lpm_add_sub
      GENERIC MAP ( lpm_width => 16,
                     lpm_direction => "UNUSED",
                     lpm_type => "LPM_ADD_SUB",
                     lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO" )
      PORT MAP ( dataa => dataa,
                  add_sub => add_sub,
                  datab => datab,
                  overflow => sub_wire0,
                  result => sub_wire1 );
END SYN;

Figure 12. VHDL code for the ADD_SUB LPM.
```
The modified VHDL code for the adder/subtractor design is given in Figure 13. It incorporates the code in Figure 12 as a component. Put the code in Figure 13 into a file tutorial_lpm\addersubtractor2.vhd. For convenience, the required file addersubtractor2.vhd is provided in the directory DE2_tutorials\design_files, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera’s DE2 web pages.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Top-level entity
ENTITY addersubtractor2 IS
    GENERIC ( n : INTEGER := 16 ) ;
    PORT ( A, B : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
    Clock, Reset, Sel, AddSub : IN STD_LOGIC ;
    Z : BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
    Overflow : OUT STD_LOGIC ) ;
END addersubtractor2 ;

ARCHITECTURE Behavior OF addersubtractor2 IS
    SIGNAL G, M, Areg, Breg, Zreg, : STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
    SIGNAL SelR, AddSubR, over_flow : STD_LOGIC ;
    COMPONENT mux2to1
        GENERIC ( k : INTEGER := 8 ) ;
        PORT ( V, W : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
        Selm : IN STD_LOGIC ;
        F : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
    END COMPONENT ;
    COMPONENT megaddsub
        PORT ( add_sub : IN STD_LOGIC ;
        dataa, datab : IN STD_LOGIC_VECTOR(15 DOWNTO 0) ;
        result : OUT STD_LOGIC_VECTOR(15 DOWNTO 0) ;
        overflow : OUT STD_LOGIC ) ;
    END COMPONENT ;

BEGIN
    -- Define flip-flops and registers
    PROCESS ( Reset, Clock )
    BEGIN
        IF Reset = '1' THEN
            Areg <= (OTHERS => '0'); Breg <= (OTHERS => '0');
            Zreg <= (OTHERS => '0'); SelR <= '0'; AddSubR <= '0'; Overflow <= '0';
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Areg <= A; Breg <= B; Zreg <= M;
            SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
        END IF ;
    END PROCESS ;

    ... continued in Part b

Figure 13. VHDL code for the circuit in Figure 3 (Part a).
Define combinational circuit
nbit_addsub: megaddsub
PORT MAP ( AddSubR, G, Breg, M, over_flow );
multiplexer: mux2to1
GENERIC MAP ( k => n )
PORT MAP ( Areg, Z, SelR, G );
Z <= Zreg ;
END Behavior;

k-bit 2-to-1 multiplexer
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    GENERIC ( k : INTEGER := 8 );
    PORT ( V , W : IN STD_LOGIC_VECTOR(k−1 DOWNTO 0) ;
        Selm : IN STD_LOGIC ;
        F : OUT STD_LOGIC_VECTOR(k−1 DOWNTO 0) ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( V , W, Selm )
    BEGIN
        IF Selm = '0' THEN
            F <= V ;
        ELSE
            F <= W ;
        END IF ;
    END PROCESS ;
END Behavior ;

16-bit adder/subtractor LPM created by the MegaWizard
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY megaddsub IS
    PORT ( add_sub : IN STD_LOGIC ;
        dataa : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        datab : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        result : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
        overflow : OUT STD_LOGIC );
END megaddsub;
ARCHITECTURE SYN OF megaddsub IS
    SIGNAL sub_wire0 : STD_LOGIC ;
    SIGNAL sub_wire1 : STD_LOGIC_VECTOR (15 DOWNTO 0);

    ... continued in Part c

Figure 13. VHDL code for the circuit in Figure 3 (Part b).
COMPONENT lpm_add_sub
GENERIC ( lpm_width : NATURAL;
        lpm_direction : STRING;
        lpm_type : STRING;
        lpm_hint : STRING );
PORT ( dataa : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        add_sub : IN STD_LOGIC ;
        datab : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        overflow : OUT STD_LOGIC ;
        result : OUT STD_LOGIC_VECTOR (15 DOWNTO 0) );
END COMPONENT;
BEGIN
overflow <= sub_wire0;
result <= sub_wire1(15 DOWNTO 0);
lpma d_add_sub_component : lpm_add_sub
GENERIC MAP ( lpm_width => 16,
                lpm_direction => "UNUSED",
                lpm_type => "LPM_ADD_SUB",
                lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO" )
PORT MAP ( dataa => dataa,
            add_sub => add_sub,
            datab => datab,
            overflow => sub_wire0,
            result => sub_wire1 );
END SYN;

Figure 13. VHDL code for the circuit in Figure 3 (Part c).

The key differences between this code and Figure 2 are:

- The statements that define the over_flow signal and the XOR gates (along with the signal H) are no longer needed.

- The adderk entity, which specifies the adder circuit, is replaced by megaddsub entity. Note that the dataa and datab inputs shown in Figure 6 are driven by the G and Breg vectors, respectively.

- AddSubR signal is specified to be the inverted version of the AddSub signal to conform with the usage of this control signal in the LPM.

If you copied the file addersubtractor2.vhd from the qdesigns directory, you have to include this file in the project. To do so, select Project > Add/Remove Files in Project to reach the window in Figure 14. Browse for the available files by clicking the button File name: ... to reach the window in Figure 15. Select the file addersubtractor2.vhd and click Open, which returns to the window in Figure 14. Click Add to include the file and then click OK. Now, the modified design can be compiled and simulated in the usual way.
Figure 14. Inclusion of the new file in the project.

Figure 15. Specify the *addersubtractor.vhd* file.
4 Results for the Augmented Design

Compile the design and look at the summary, which is depicted in Figure 16. Observe that the modified design is implemented in 52 logic elements, which is the same as when using the code in Figure 2. In very small circuits, which is the case with our example, it is unlikely that using LPMs will result in a significant advantage. However, in more complex designs the advantage of using LPMs is likely to be significant. The reason is that the LPMs implement the required logic more efficiently than what the compiler can do from simple VHDL code, such as the code in Figure 2. The user should consider using an LPM whenever a suitable one exists.

Figure 16. Compilation Results for the Augmented Circuit.

Copyright ©2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera’s standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an “as-is” basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.