Using Hierarchy in VHDL Design

Introduction

Hierarchical design methodology has been commonly used for quite some time by system designers and software developers. There are two primary advantages to using this methodology. First, it allows commonly-used building blocks to be created separately and saved for later use without having to redesign or reverify them. Second, it allows for more readable design files by keeping the top-level design file as a simple integration of smaller building blocks, either user-defined or from a vendor-supplied library. In system design, these building blocks normally take the form of schematic symbols instantiated into a schematic drawing, while in software they are functions or procedures that are called from the main program.

VHDL includes a set of features specifically designed to make hierarchical design both simple and powerful. This note will first describe these features and then walk through a simple example of how they might be used. It assumes that the reader is familiar with how to create a VHDL design unit consisting of an entity-architecture pair.

Key Concepts

In order to construct a hierarchical design in VHDL, the designer must understand the concepts of components, packages and libraries.

Component – A component is a VHDL design unit that may be instantiated in other VHDL design units. Before it can be instantiated, it must be declared using the COMPONENT declaration which specifies the name of the component and lists its local signal names.

Package – A package is a collection of VHDL declarations that can be used by other VHDL descriptions. For the purpose of creating hierarchical designs, a package consists of one or more components. However, a package may also include other types of declarations.

Library – A library is a logical storage facility for design units. Before a component can be instantiated in a higher-level design unit, its package must be compiled into a library that is visible to that design unit, usually the current work library.

Simple Example

Consider the following example. A designer discovers that for a specific type of circuit design he commonly needs an unusual type of counter. (“Commonly,” in this reference, means that this counter is likely to be used either multiple times in a particular design or across multiple designs. Both are cases where hierarchical design simplifies things.) This counter is a simple four-bit counter, but it must output a terminal count indication (tc) and roll over to zero when it reaches 1110 rather than 1111.

A design file that would accomplish this is shown in Appendix A. (The reader should understand the contents of the entity-architecture pair—they will not be discussed further.) In order to use this counter in other VHDL design units, it is declared as a component within a package at the top of the file. The component declaration simply names the design unit and lists its signal names. When this file is compiled, the package is placed into the current library and the component it contains may then be instantiated into other designs compiled into that library. If this were a standalone design, the entire package declaration could be omitted.

Now, suppose this design consists of two of these counters with their outputs multiplexed as in *Figure 1.* We can then instantiate our counter twice as in the design in Appendix B. All that is necessary is the statement:

```
use work.cnt_pkg.all;
```

at the top of the file, which makes any components in the cnt_pkg visible within the current design unit, as long as the package was compiled into the work library. The counters are then instantiated by giving them unique labels and listing the signals connected to the port map in the same order as the component declaration.

We could also have created the mux as a separate component and instantiated it, but it is simpler to use the if-then-else structure.

Multiple Components

For further illustration, assume our complete design includes two types of counters, one that rolls over at 1110 and one that rolls over at 1011, as shown in *Figure 2*. We could simply repeat the above procedure and create another design file with another component and package and then use both of these packages in our top-level design file.

However, it may be easier to keep track of things if we keep similar counter designs together in a single package as in Appendix C. This file contains both entity-architecture pairs and two components in a single package. As before, when this file is compiled, the package is added to the current library and its components are made visible with a single-use clause as in Appendix D.

Configurable Components using Generics

When multiple components that have the same basic architecture but differ in one or more parameters are needed (such as the two counters in the previous example) VHDL generics allow a more compact approach. Generics are a means by which parameters may be passed to a component when it is instantiated allowing a configurable component.

In Appendix E a component is created that is the same basic counter, but allows the terminal count to be configured using a generic. Instead of hard-coding this value, a bit_vector is used in the architecture. This bit_vector is then declared in the entity and component declarations. Generics may also be of other types such as integers and a component may contain multiple generics (although our example contains only one).

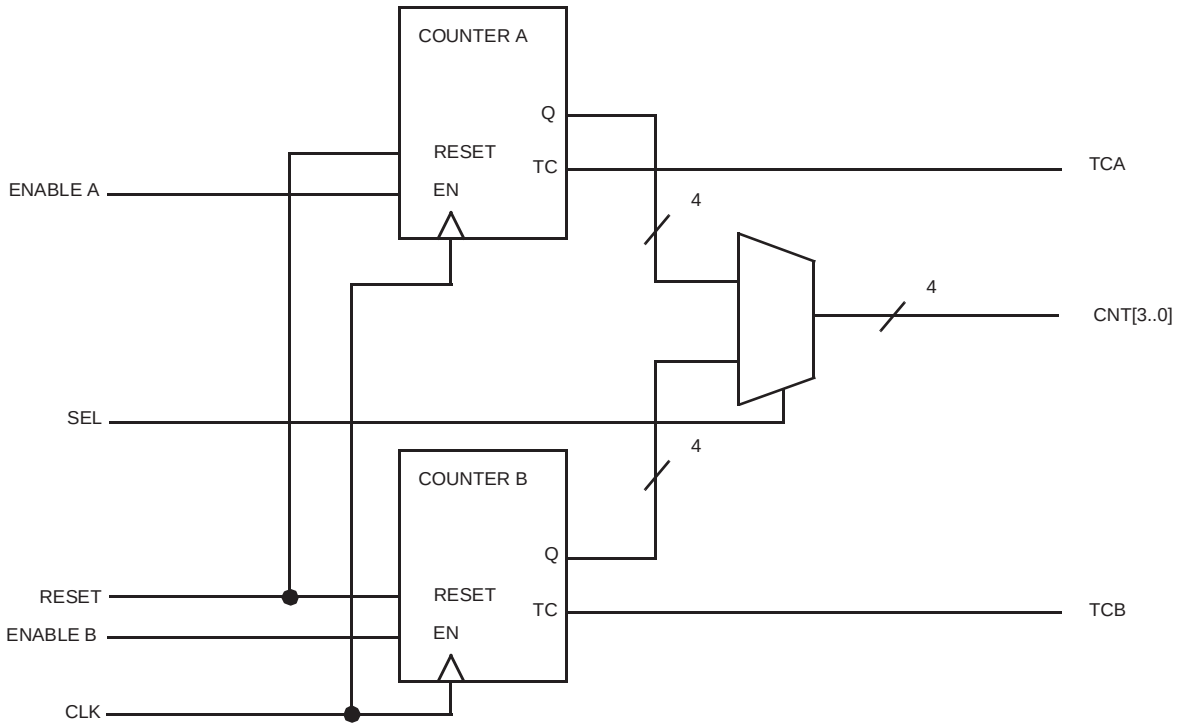

Figure 1. Multiplexed Dual Counter Design

Appendix F is the top-level design unit of the same design from *Figure 2*, but this time it is using the component with the generic rather than two different components. When the component is instantiated, it is configured by passing it the specific bit_vector in the generic map.

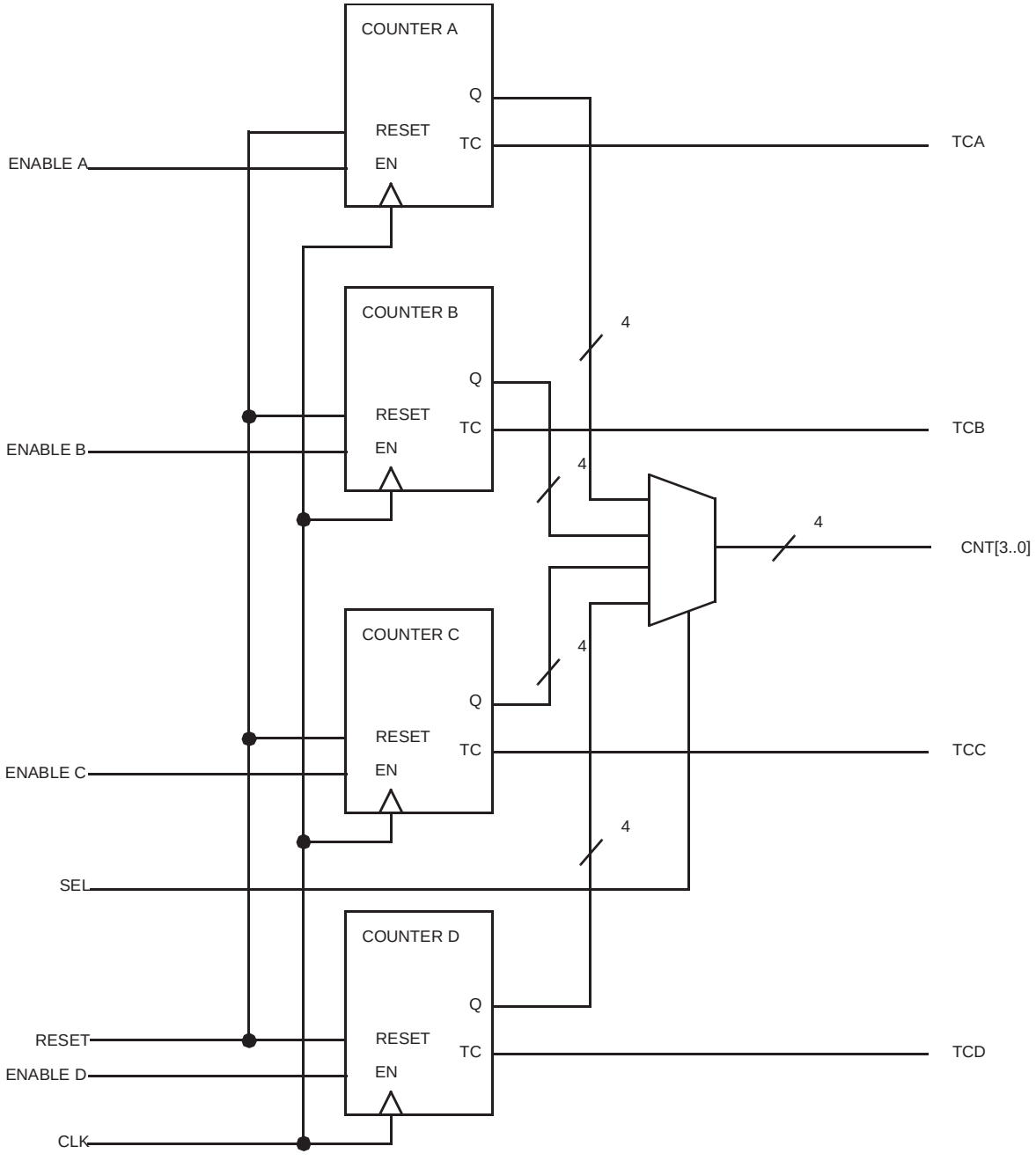

Figure 2. Multiplexed Quad Counter Design

Appendix A. Counter with Terminal Count and Rollover Selection

```
package cnt_pkg is
  component count15 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
end cnt_pkg;

use work.bit_arith.all;

entity count15 is port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end count15;

architecture one of count15 is
begin

process (cnt) begin
    if cnt="1110" then
        tc<='1';
    else
        tc<='0';
    end if;

end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt="1110" and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=cnt+1;
        else
            cnt<=cnt;
        end if;
    end if;

end process;

end one;
```

Appendix B. Instantiation of Counter from Appendix A

```
use work.cnt_pkg.all;

entity muxcntr is port(
    clk, enablea, enableb, reset, sel:in bit;
    cnt:out bit_vector (3 downto 0);
    tca, tcb:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb:bit_vector(3 downto 0);

begin

cntra:count15 port map(clk, enablea, reset, muxina, tca);
cntrb:count15 port map(clk, enableb, reset, muxinb, tcb);

process (sel) begin
    if sel='1' then
        cnt<=muxina;
    else
        cnt<=muxinb;
    end if;

end process;
end one;
```

Appendix C. Multiple Counters in a Single Package

```
package cnt_pkg is
  component count15 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
  component count12 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
end cnt_pkg;

use work.bit_arith.all;

entity count15 is port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end count15;

architecture one of count15 is
begin

process (cnt) begin
    if cnt="1110" then
        tc<='1';
    else
        tc<='0';
    end if;
end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt="1110" and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=cnt+1;
        else
            cnt<=cnt;
        end if;
    end if;
```

Appendix C. Multiple Counters in a Single Package (continued)

```
end process;
end one;

use work.bit_arith.all;

entity count12 is port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end count12;

architecture one of count12 is
begin

process (cnt) begin
    if cnt="1011" then
        tc<='1';
    else
        tc<='0';
    end if;
end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt="1011" and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=cnt+1;
        else
            cnt<=cnt;
        end if;
end if;

end process;
end one;
```

Appendix D. Instantiation of Counters in Appendix C

```
use work.cnt_pkg.all;

entity muxcntr is port(
    clk, enablea, enableb, enablec, enabled, reset:in bit;
    sel:in bit_vector (1 downto 0);
    cnt:out bit_vector (3 downto 0);
    tca, tcb, tcc, tcd:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb, muxinc, muxind:bit_vector(3 downto 0);

begin

cntra:count15 port map(clk, enablea, reset, muxina, tca);
cntrb:count15 port map(clk, enableb, reset, muxinb, tcb);
cntrc:count12 port map(clk, enablec, reset, muxinc, tcc);
cntrd:count12 port map(clk, enabled, reset, muxind, tcd);

process (sel)begin
    if sel="11" then
        cnt<=muxina;
    elsif sel="10" then
        cnt<=muxinb;
    elsif sel="01" then
        cnt<=muxinc;
    else
        cnt<=muxind;
    end if;

end process;
end one;
```

Appendix E. Parametrizable Counters Using Generics

```
package cnt_pkg is
    component countg
    generic (stop:bit_vector(3 downto 0):="1111");
      port(
      clk, enable, reset:in bit;
      cnt:inout bit_vector (3 downto 0);
      tc:out bit);
    end component;
end cnt_pkg;

use work.bit_arith.all;

entity countg is
  generic (stop:bit_vector(3 downto 0):="1111");
    port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end countg;

architecture one of countg is
begin

process (cnt) begin
    if cnt=stop then
        tc<='1';
    else
        tc<='0';
    end if;
end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt=stop and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=cnt+1;
        else
            cnt<=cnt;
        end if;
    end if;

end process;
end one;
```

Appendix F. Multiplexed Quad Counter Design

```
use work.cnt_pkg.all;

entity muxcntr is port(
    clk, enablea, enableb, enablec, enabled, reset:in bit;
    sel:in bit_vector (1 downto 0);
    cnt:out bit_vector (3 downto 0);
    tca, tcb, tcc, tcd:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb, muxinc, muxind:bit_vector(3 downto 0);

begin
cntra:countg generic map("1110") port map(clk, enablea, reset, muxina, tca);
cntrb:countg generic map("1110") port map(clk, enableb, reset, muxinb, tcb);
cntrc:countg generic map("1011") port map(clk, enablec, reset, muxinc, tcc);
cntrd:countg generic map("1011") port map(clk, enabled, reset, muxind, tcd);

process (sel) begin
    if sel="11" then
        cnt<=muxina;
    elsif sel="10" then
        cnt<=muxinb;
    elsif sel="01" then
        cnt<=muxinc;
    else
        cnt<=muxind;
    end if;

end process;
end one;
```