To incorporate hierarchy in VHDL we must add component declarations and component instantiations to the model. In addition, we need to declare internal signals to interconnect the components. We can also include processes and concurrent statements in the hierarchical model.

Format for Architecture body (for internal signals & hierarchy):

```
architecture architecture_name of entity_name is
signal declarations                    -- for internal signals in model
component decalarations                -- for hierarchical models
begin
    :
    component instantiations
    processes
    concurrent statements
    :
end architecture architecture_name;
```

Format for component declaration:

```
component component_name is
    generic (generic_name(s): type := initial_value;
                :
                generic_name(s): type := initial_value);
    port (signal_name(s): mode signal_type;
                :
                signal_name(s): mode signal_type);
end component component_name;
```

Note that the *component_name* is the same as the *entity_name* from the model being called up. As a result, component declarations look just like the entity statements (including generics and ports) for the component being declared but with "component" substituted for "entity". (In fact, I usually copy the entity from the component being declared and paste it in the component declaration and do a global replacement of "component" for "entity".)

The component instantiation is the actual call to a specific use of the model. A single component declaration can have multiple instantiations. As a result, each component instantiation must include a unique name (*instantiation_label* along with the component (*component_name*) being used. Furthermore, the values assigned to generics in the component instantiation will override any initial values assigned to the generic in the entity statement or component declaration for that model (important when a component is called multiple times with different generic values assigned to each call). There are some other subtle variations (noted in red below) in the format for a component instantiation compared to an entity or component declaration. There are two methods (and formats) for connecting signals to the port of the component: 1) named association (aka, keyword notation) and 2) positional association (aka, positional notation).

Format for component instantiation (*keyword notation*):

```
instantiation_label: component_name
    generic map (generic_name => value,          -- note , at end & not ;
                        :
                 generic_name => value)          -- note no ; at end
    port map (port_name => signal_name,          -- note , at end & not ;
                        :
              port_name => signal_name);
```

Note that *port_name* is the actual *signal_name* from the component declaration (which is also the same as in the original entity statement). The *signal_name* given here is the actual signal in the hierarchical design being connected to that particular *port_name*. The order of the generics values and signal_names can be in any order since each is associated to a unique generic or port by the => operator. The penalty is more junk to type in the spirit of verbosity.

Format for component instantiation (*positional notation*):

```
instantiation_label: component_name
    generic map (generic_value,  -- note , at end and not ;
                        :
                 generic_value)          -- note no ; at end
    port map (signal_name,                -- note , at end and not ;
                        :
              signal_name);
```

Note that the *generic_values* and *signal_names* must appear in the order given in the component declaration in order to connect to the correct *generic_name* or *port_name*, respectively. Finding the correct order is no big deal since it is given above in the component declaration that must also be included in the model (therefore, positional notation is my personal preference).

Components instantiations are treated as concurrent statements where the inputs to component represent the sensitivity list such that the component model is evaluated whenever there is an event on one of its inputs. Therefore, component instantiations cannot be included in a process.

Note that a constant logic value (i.e., '0' or '1') can be assigned to an unused input in the component instantiation which will allow *most* synthesis tools to minimize the logic associated with the unused input (or feature of the model being instantiated). This also applies to bit_vectors as well where a string of constant logic values can be applied. This is a powerful capability when used in conjunction with parameterized modeling by allowing even more use of a parameterized model.

For example, consider a rising edge triggered N-bit counter with active high synchronous reset, active high preset, active high parallel load, and active high count enable (with that order of precedence).

```
entity REG is
   generic (N: int := 3);
   port (CLK,RST,PRE,LOAD,CEN: in bit;
          DATIN: in bit_vector (N-1 downto 0);
          DOUT: buffer bit_vector (N-1 downto 0));
end entity REG;
architecture RTL of REG is
begin
process (CLK) begin
   if (CLK'event and CLK='1') then
          if (RST = '1') then DOUT <= (others => '0');
          elsif (PRE = '1') then DOUT <= (others => '1');
          elsif (LOAD = '1') then DOUT <= DATIN;
          elsif (CEN = '1') then DOUT <= DOUT + 1;
          end if;
   end if;
end process;
end architecture RTL;
```

Next consider the 3 instances of REG in the following hierarchical use of this model:

```
entity TOP is
   port (CLK,X,Y,Z: in bit;
          DIN: in bit_vector(5 downto 0);
          Q1: out bit_vector(5 downto 0);
          Q2: out bit_vector(4 downto 0);
          Q3: out bit_vector(3 downto 0));
end entity TOP;
architecture HIER of TOP is
component REG is
   generic (N: int := 3);
   port (CLK,RST,PRE,LOAD,CEN: in bit;
          DATIN: in bit_vector (N-1 downto 0);
          DOUT: buffer bit_vector (N-1 downto 0));
end component REG;
begin
R1: REG generic map (6)
   port map (CLK,'0','0',X,'0',DIN,Q1);
R2: REG generic map (5)
   port map (CLK,Y,'0','0',Z,"00000",Q2);
R3: REG generic map (4)
   port map (CLK,'0','0','1','0',DIN(3 downto 0),Q3);
end architecture HIER;
```

When we synthesize this circuit, R1 will be a 6-bit register with active high parallel load signal (not reset, preset, or counter logic), R2 will be a 5 bit counter with active high reset and active high count enable (no parallel load or preset logic), and R3 will simply be 4 flip-flops without any features, as can be seen in the following synthesis report from ISE8.2

```
Performing bidirectional port resolution...
Synthesizing Unit <REG_1>.
    Found 6-bit register for signal <DOUT>.
    Summary: inferred   6 D-type flip-flop(s).
Synthesizing Unit <REG_2>.
    Found 5-bit up counter for signal <DOUT>.
    Summary: inferred   1 Counter(s).
Synthesizing Unit <REG_3>.
    Found 4-bit register for signal <DOUT>.
    Summary: inferred   4 D-type flip-flop(s).
Synthesizing Unit <hier>.
HDL Synthesis Report
Macro Statistics
# Counters                                             : 1
 5-bit up counter                                      : 1
# Registers                                            : 2
 4-bit register                                        : 1
 6-bit register                                        : 1
Advanced HDL Synthesis Report
Macro Statistics
# Counters                                             : 1
 5-bit up counter                                      : 1
# Registers                                            : 10
 Flip-Flops                                            : 10
Final Register Report
Macro Statistics
# Registers                                            : 15
 Flip-Flops                                            : 15

Device utilization summary:
Selected Device : 3s200pq208-5
Number of Slices:              3  out of   1920     0%
 Number of Slice Flip Flops: 15  out of   3840     0%
 Number of 4 input LUTs:       6  out of   3840     0%
 Number of IOs:               25
 Number of bonded IOBs:       25  out of    141    17%
    IOB Flip Flops:           10
 Number of GCLKs:              1  out of      8    12%
```

Note that only 3 slices and 6 LUTs were used (actually only 5 LUTs in that final design) for the counter R2 – the other two registers only used flip-flops which already have

clock enable, reset, and preset capabilities (even though none of these features were used in the case of R3). Otherwise, the design would have required 12 slices and 20 LUTs for implement the three full register designs.