VHDL was designed to allow the same structure as real hardware, mimicking the designs that are created for complicated systems. In particular, individual modules are composed of two sections, as depicted in the following diagram:

The two parts are the communication part and the work part. The communication part is called the “entity”, and the work part is called the “architecture”. Within the entity is found the “port” which is the mechanism used to connect to other modules. The port lists the signals that are visible to the outside world and their directions. These signals can then be used to communicate with other modules that have the same type of signals. That is, buses hook to buses, wires to wires, outputs to inputs, and inputs to outputs. The designer is then free to organize the design in any way that he chooses in order to achieve the objectives of his system.

The modules are hooked together in a hierarchical fashion to minimize the design time and allow designers to use hierarchical concepts in creating the system. As an example, consider the steps involved in designing and testing a four-bit adder. The task is stated in a very straightforward fashion: create a system that will add two four-bit numbers together, along with a carry-in, and create a four-bit result, as well as a carry-out. The designer decides to utilize a ripple-carry concept for simplicity, and the approach is sketched simply in block diagram form as:
The designer of this system is now ready to actually carry out the steps of the design, knowing that the trick is to create a full adder module, then hook the modules together, and finally to test the result.

With this idea in mind, the design will be undertaken from a structural VHDL approach. That is, the purpose of this note is to demonstrate how to create VHDL code that is hierarchical in nature, using groups of less complicated elements together to create a more complicated system. In this case, the designer wants to use gates to create a full-adder, then use four full adders to create a four-bit adder, and then create a testbed to test the four-bit adder. The first step in the design is to understand all the constituent parts, and we assume that the designer has accomplished that part. The full-adder has been designed (truth table, Karnaugh maps, equations…) to give:

\[
\begin{align*}
\text{COUT} &= A \cdot B + A \cdot \text{CIN} + B \cdot \text{CIN} \\
F &= \overline{A} \cdot \overline{B} \cdot \text{CIN} + \overline{A} \cdot B \cdot \text{CIN} + A \cdot \overline{B} \cdot \text{CIN} + A \cdot B \cdot \text{CIN}
\end{align*}
\]

Once the initial part of the design is completed, then the idea is relatively simple: use gates to implement a full adder, use full adders to implement a four-bit adder, use a testbed to test the four-bit adder. The hierarchy of this is represented as:

```
  Test Bed
     |
     v
  Four Bit Adder
     |
     v
  Full Adder
     \\  /  \\
    7410 7404
     /  \\
  7400 7404
```

The full adder can be made with 7400’s (2-input NAND), 7410’s (3-input NAND), 7420’s (4-input NAND), and 7404’s (inverter). Then, four of the full adders can be combined to create the Four Bit Adder, and finally, the testbed is created to test the Four Bit Adder. Each of the lines in the diagram represents the connection of components into another module with the use of port statements and component instantiations. So, let’s create code for the 7400, then code for the Full Adder, then code for the Four Bit Adder, and finally, code for the Test Bed.

We start with the code for the 7400 (and assume that the other gates are so similar that we don’t need to explicitly include code for those modules). The VHDL representation will consist of an entity and an architecture. The entity for this example contains only the port
library IEEE;                   -- make the IEEE library visible
use IEEE.STD_LOGIC_1164.all;    -- access the package for STD_LOGIC

entity NAND2 is                 -- begin entity, name it NAND2
    port (                        -- beginning of port statement
        A_H : in STD_LOGIC;        -- A_H is an input of type STD_LOGIC
        B_H : in STD_LOGIC;        -- B_H is an input of type STD_LOGIC
        F_L : out STD_LOGIC         -- F_L is an output of type STD_LOGIC
    );                            -- parenthesis to close port statement
end NAND2;                      -- end of the entity part

architecture SIMPLE of NAND2 is -- begin architecture, hook to NAND2
    begin                        -- declaration area, empty here
        F_L <= not (A_H and B_H);     -- signal assignment: out <= inputs
            -- must be logically correct
    end SIMPLE;                     -- end of architecture part; name
                                        -- must be name of architecture

The comments included in the code are cryptic, so some explanation is in order. The first
statement (library…) lets the VHDL system know that this module needs access to a
particular library, and makes that library visible to the system. The second statement (use
…) identifies a particular module (STD_LOGIC_1164) in the library, and allows the
VHDL system to access everything in that module (hence, the .all at the end).
STD_LOGIC is the designation given to an IEEE standard that is very useful for allowing
different manufacturers to work together to provide correct and compatible logic. The
STD_LOGIC system allows for tri-state stuff and things that we will explain a lot later.

The entity part begins with the keyword “entity” followed by the name of the module. If
the compilation process is successful, a module of this name will be placed in the current
library for use by other modules. After the name of the module comes the keyword “is”;
this is a required part of the entity statement.

The only thing in the entity statement of this example is the port statement, which begins
with the keyword “port”, followed by an opening parenthesis. The purpose of a port
statement is to identify the inputs and outputs (as well as any bi-directional signals, which
are not used here) of the module and give their names and directions. This is
demonstrated in the next 3 lines. It is possible to combine similar inputs/outputs in a
single line (such as A_H and B_H), but I suggest that you use a single line for each input
or output, as shown in the example above. The format of each of the entries is:
NAME_OF_SIGNAL : TYPE_OF_SIGNAL;. That is, the first thing needed is the name
of the signal. Use mnemonic methods – name the signal something that gives an
indication of what it is. In this case, the input is simply A_H. Note that the H ending
indicates that this signal is asserted high. After the name comes a colon, and after the
colon comes a designator of what the direction of this signal is. Direction can be in, out,
or inout, for inputs, outputs, and bi-directional signals. After the direction comes the type
of the signal, which in the example above is STD_LOGIC. Other signal types are
possible, as we shall see below. Finally, the line is terminated with a semi-colon. This is
true for all entries in the port statement, except the last one, which doesn’t have a semicolon. The port statement is concluded with a closing parenthesis and semi-colon.

The entity statement is closed with an “end” followed by the name of the entity. This name is optional, but it is a good habit to get into – including names to identify function of the end. The entity statement contains the information needed to communicate with other elements in the system. In the case of this module, we know that the gate is expecting two inputs, and these inputs are named A_H and B_H, and that one output (F_L) will be produced.

The work of the system is given in the architecture part. In the example above, that part begins with the keyword “architecture” followed by the name of the architecture, which in this example is “SIMPLE”. This name is followed by the keyword “of” and another name, which is the name of the entity to which this architecture belongs. Finally, this part of the statement is concluded with the keyword “is”. Between the “is” and the “begin” which actually begins the work of the system is what is called the declaration area of the architecture. This declaration area is used to declare things that will be used in the architecture. But in this example there is nothing more needed than the signals presented in the entity, so nothing appears in the declaration area.

The “begin” marks the beginning of the work of the architecture, and like every other “begin” in VHDL, it will eventually be matched with an “end”. Between the “begin” and the “end” is the work of the system. In all architecture bodies, the statements that appear between the “begin” and the “end” are parallel in nature. That is, if there are more than one statement, the statements execute in parallel, and there is no timing relationship implied by the order of the statements in the architecture.

In this example, the equation is used to simply implement a NAND gate, so the statement is very simple. The example is a signal assignment statement, and demonstrates the use of the compound delimiter ‘<=’, which is called the signal assignment operator. It simply says, “send the value calculated on the right to the element on the left”. This is an over-simplification of what happens, but is enough for now. On the right is the logical definition of what should be sent to the left. This is a very simple example, but the syntax for signal assignment statements can be very complicated. The thing to remember about signal assignments that are simply gating representations (as shown above) is that the precedence of all logical operators (and, or, nand, nor, xor, etc) is the same, so you need to add parentheses to make the statement represent exactly what you want to represent.

The architecture body terminates with the “end”, again followed by the name of the architecture.

This same model can be followed to create the other basic gates needed for this example. Note, however, that in this simple gating representation the timing information is missing. That can be built into the representation of the gate, but we will save that complication for a later example. The point of this note is to demonstrate how to take
these simple elements and combine them together to make a more complicated system, as shown in the hierarchical diagram above.

To demonstrate the hierarchical nature of the system, note that we want to instantiate four types of gates (2-input, 3-input, 4-input NAND gates, inverter) to make a full adder, then instantiate 4 full adders to make a 4-bit adder, and finally instantiate a 4-bit adder in a test bed to test the system. Using a gate in another module, or stated more generally, to use one module in another module, the first module must be declared in such a way that it is known to the second module. The most direct way to do this is to include a “component declaration” in the declaration area of the second module. This lets the VHDL system know that the module exists. Then, the user can instantiate the first module in the second by naming it and hooking up the signals in the port statement appropriately. Let us return to the full adder example, and represent the gating equations above as follows:

This represents the equations above, and the functionality needed for the full adder. However, note that in the diagram the remaining wires have been added, wires that connect the ANDing functions to the ORing functions. So, in the declaration area of the full adder the following things need to be declared: INV, NAND2, NAND3, NAND4, and seven wires. Then, in the work area of the architecture body, three inverters, three 2 input NAND gates, five 3 input NAND gates, and one 4 input NAND gate need to be instantiated, and then connected as shown in the diagram above. The code for system is:
library IEEE; -- make IEEE library visible
use IEEE.STD_LOGIC_1164.all; -- open STD_LOGIC_1164 package

entity FULL_ADDER is -- this for FULL_ADDER element
port(
    A_H   : in  STD_LOGIC;    -- input A of full adder
    B_H   : in  STD_LOGIC;    -- input B of full adder
    CIN_H : in  STD_LOGIC;    -- carry input of full adder
    F_H   : out STD_LOGIC;    -- function out of full adder
    COUT_H: out STD_LOGIC     -- carry out of full adder
);
end FULL_ADDER;

architecture STRUCTURAL of FULL_ADDER is -- begin architecture

component INV is -- component declaration for
    port (                      -- inverter - port statement
        A_H : in  STD_LOGIC;       -- just as it appears in the
        F_L : out STD_LOGIC        -- file of the inverter
    );
end component;

component NAND2 is -- component declaration for
    port (                      -- 2 input NAND - port statement
        A_H : in  STD_LOGIC;       -- just as it appears in the
        B_H : in  STD_LOGIC;       -- file of the 2 input NAND
        F_L : out STD_LOGIC
    );
end component;

component NAND3 is -- component declaration for
    port (                      -- 3 input NAND - port statement
        A_H : in  STD_LOGIC;       -- just as it appears in the
        B_H : in  STD_LOGIC;       -- file of the 3 input NAND
        C_H : in  STD_LOGIC;
        F_L : out STD_LOGIC
    );
end component;

component NAND4 is -- component declaration for
    port (                      -- 4 input NAND - port statement
        A_H : in  STD_LOGIC;       -- just as it appears in the
        B_H : in  STD_LOGIC;       -- file of the 4 input NAND
        C_H : in  STD_LOGIC;
        D_H : in  STD_LOGIC;
        F_L : out STD_LOGIC
    );
end component;

-- the following statement declares the 3 wires needed for inverses
signal A_L, B_L, CIN_L : STD_LOGIC;
-- the following statement declares the 7 wires needed for design
signal W1_L, W2_L, W3_L, W4_L, W5_L, W6_L, W7_L : STD_LOGIC;

begin

I1: INV                            -- instantiate I1 as an inverter
    port map (                    -- map the port - hook it up
        A_H => A_H,                -- input to A_H
        F_L => A_L                 -- output to A_L
    );

I2: INV                            -- instantiate I2 as an inverter
I3: INV
port map (                   -- instantiate I3 as an inverter
  A_H => CIN_H,              -- input to CIN_H
  F_L => CIN_L               -- output to CIN_L
);  

N2_1: NAND2                  -- instantiate N2_1 as 2 input NAND
port map (                   -- map the port - hook it up
  A_H => A_H,                -- let this be A_H
  B_H => B_H,                -- let this be B_H
  F_L => W1_L                -- output to wire 1
);  

N2_2: NAND2                  -- instantiate N2_2 as 2 input NAND
port map (                   -- map the port - hook it up
  A_H => A_H,                -- let this be A_H
  B_H => CIN_H,              -- let this be CIN_H
  F_L => W2_L                -- output to wire 2
);  

N2_3: NAND2                  -- instantiate N2_3 as 2 input NAND
port map (                   -- map the port - hook it up
  A_H => B_H,                -- let this be B_H
  B_H => CIN_H,              -- let this be CIN_H
  F_L => W3_L                -- output to wire 3
);  

N3_1: NAND3                  -- instantiate N3_1 as 3 input NAND
port map (                   -- map the port - hook it up
  A_H => W1_L,               -- function is really ORing function
  B_H => W2_L,               -- so hook the three wires to the
  C_H => W3_L,               -- three inputs, and then the
  F_L => COUT_H              -- output becomes COUT_H
);  

N3_2: NAND3                  -- instantiate N3_2 as 3 input NAND
port map (                   -- map the port - hook it up
  A_H => A_L,                -- this three input gate implements
  B_H => B_L,                -- the not-A and not-B and CIN
  C_H => CIN_H,              -- function of the FOUT
  F_L => W4_L                -- output becomes W4_L
);  

N3_3: NAND3                  -- instantiate N3_3 as 3 input NAND
port map (                   -- map the port - hook it up
  A_H => A_L,                -- this three input gate implements
  B_H => B_H,                -- the not-A and B and not-CIN
  C_H => CIN_L,              -- function of the FOUT
  F_L => W5_L                -- output becomes W5_L
);  

N3_4: NAND3                  -- instantiate N3_4 as 3 input NAND
port map (                   -- map the port - hook it up
  A_H => A_H,                -- this three input gate implements
  B_H => B_H,                -- the A and B and CIN
  C_H => CIN_H,              -- function of the FOUT
  F_L => W6_L                -- output becomes W6_L
);
The above code implements a single full-adder. Seems like a lot of text for something so simple. But the method can be continued to create systems however complex a designer wants to make them. Note that in the work area of the architecture there are only component instantiations. Hence, the design process is one of hooking up modules together in a way that produces the correct result according to system specification.

Continuing the hierarchy concept, the next step is to take four of these full adders and put them together into a four-bit adder. In this case, there is only one type of element to declare in the architecture: the full adder. Then, in the work area of the architecture the four full adders are put together in order to create a four-bit unit. The code to accomplish this is:

```vhdl
library IEEE; -- make IEEE library visible
use IEEE.STD_LOGIC_1164.all; -- open STD_LOGIC_1164 package

entity FOUR_BIT is
  port ( -- this time, use buses for input, output vals
            A_H    : in  STD_LOGIC_VECTOR ( 3 downto 0 ); -- bus for A val
            B_H    : in  STD_LOGIC_VECTOR ( 3 downto 0 ); -- bus for B val
            CIN_H  : in  STD_LOGIC;
            F_H    : out STD_LOGIC_VECTOR ( 3 downto 0 ); -- bus for output
            COUT_H : out STD_LOGIC
          );
end FOUR_BIT;

architecture STRUCTURAL of FOUR_BIT is

component FULL_ADDER is -- component statement for the
  port ( -- full adder is copy of the
            A_H    : in  STD_LOGIC; -- port statment
            B_H    : in  STD_LOGIC;
            CIN_H  : in  STD_LOGIC;
            F_H    : out STD_LOGIC;
            COUT_H : out STD_LOGIC
          );
end component;

-- the following signal declaration creates three wires for the
-- three carries between stages of the four bit adder...
```
signal C0_H, C1_H, C2_H : STD_LOGIC;

begin

FA0: FULL_ADDER
    port map (    -- map the port - hook it up
        A_H    => A_H(0),    -- hook A_H to A input bus
        B_H    => B_H(0),    -- hook B_H to B input bus
        CIN_H  => CIN_H,     -- hook CIN_H of first adder to CIN_H
        F_H    => F_H(0),    -- hook F_H to F output bus
        COUT_H => C0_H       -- hook COUT_H to C0_H to pass to next
    );

FA1: FULL_ADDER
    port map (    -- map the port - hook it up
        A_H    => A_H(1),    -- hook A_H to A input bus
        B_H    => B_H(1),    -- hook B_H to B input bus
        CIN_H  => C0_H,      -- hook CIN_H of first adder to CIN_H
        F_H    => F_H(1),    -- hook F_H to F output bus
        COUT_H => C1_H       -- hook COUT_H to C1_H to pass to next
    );

FA2: FULL_ADDER
    port map (    -- map the port - hook it up
        A_H    => A_H(2),    -- hook A_H to A input bus
        B_H    => B_H(2),    -- hook B_H to B input bus
        CIN_H  => C1_H,      -- hook CIN_H of first adder to CIN_H
        F_H    => F_H(2),    -- hook F_H to F output bus
        COUT_H => C2_H       -- hook COUT_H to C2_H to pass to next
    );

FA3: FULL_ADDER
    port map (    -- map the port - hook it up
        A_H    => A_H(3),    -- hook A_H to A input bus
        B_H    => B_H(3),    -- hook B_H to B input bus
        CIN_H  => C2_H,      -- hook CIN_H of first adder to CIN_H
        F_H    => F_H(3),    -- hook F_H to F output bus
        COUT_H => COUT_H     -- hook COUT_H to C3_H to pass to output
    );

end STRUCTURAL;

This unit has a construct that we haven’t seen yet, which is the STD_LOGIC_VECTOR.
This construct is the method used here to create a bus. Note that the syntax is open-
parenthesis, upper limit, keyword downto, lower limit, and close-parenthesis. In this way
we can name a collection of wires and treat them as a bus. Hence, the inputs and output
of the module defined above are all 4 bit buses.

And once again, the work is done by A) declaring the components to use (in this case,
just one), B) declaring signals to connect components together, C) instantiating the
components, and D) use port maps to identify connections between units, and between
inputs/outputs and units.

Hopefully, this is the correct VHDL for the system. To test it, we need to create a test
bed, which is just another level of code. In the test bed, the top level of the design is
instantiated just like any other component, and signals are declared in order to hook to the
component. Then, the signals are activated in the desired order to test the various
functions of the unit under test. Here is a short test bed for the above code:
library IEEE;                   -- make IEEE library visible
use IEEE.STD_LOGIC_1164.all;   -- open STD_LOGIC_1164 package

entity TT is                   -- entity for test bed is
end TT;                        -- empty entity

architecture TT of TT is      -- simple architecture statement....

-- component statement for the unit under test...
component FOUR_BIT is
  port (
    A_H    : in  STD_LOGIC_VECTOR ( 3 downto 0 );
    B_H    : in  STD_LOGIC_VECTOR ( 3 downto 0 );
    CIN_H  : in  STD_LOGIC;
    F_H    : out STD_LOGIC_VECTOR ( 3 downto 0 );
    COUT_H : out STD_LOGIC
  );
end component;

-- include signals to hook up to unit under test. These signals
-- form the things that you can see at the top level of the
-- simulator...
signal ABUS : STD_LOGIC_VECTOR ( 3 downto 0 );
signal BBUS : STD_LOGIC_VECTOR ( 3 downto 0 );
signal FBUS : STD_LOGIC_VECTOR ( 3 downto 0 );
signal T_CIN_H : STD_LOGIC;
signal T_COUT_H : STD_LOGIC;
begin                         -- the beginning of the architecture
C: FOUR_BIT                   -- instantiate the unit under test
  port map (                  -- map the port - hook it up to signals
    A_H    => ABUS,           -- local signal: ABUS
    B_H    => BBUS,           -- local signal: BBUS
    CIN_H  => T_CIN_H,        -- local signal: T_CIN_H
    F_H    => FBUS,           -- local signal: FBUS
    COUT_H => T_COUT_H        -- local signal: T_COUT_H
  );

-- the above component instantiation hooks up the local signals
-- to the unit under test. Now the system needs to generate the
-- signals to test the unit under test. The easiest way to do this
-- for small systems is to create a "process" to repeat patterns
-- that form the basis for test. A process statement followed
-- immediately by a begin is a construct that will start work
-- immediately after the begin, and continue until the end of the
-- process, then repeat. In this construct, the process must have
-- a wait statement that identifies how long to wait. (In this
-- example, there are many wait statements...) So, the idea is,
-- set up a test pattern, wait for some time, change the test
-- pattern, wait for some time, etc...
-- In the process below, the values to change are all put on a
-- single line; they could be three separate lines. Also,
-- vector values are in double quotes, bit values in single
-- quotes. Also, only a few values are given here... The
-- testbed should test the unit-under-test as exhaustively
-- as feasible....
process
begin
  ABUS <= "0000" ; BBUS <= "0000" ; T_CIN_H <= '0';
  wait for 50 ns;
  ABUS <= "0000" ; BBUS <= "0000" ; T_CIN_H <= '1';
This unit has at another new construct, which is the process. This construct is one of the very few constructs in VHDL in which the statements are sequential, rather than concurrent. Hence, in the architecture above, the “process…begin” starts the work, and the work continues to the “end process” line, which is the terminal statement of the process. So, the designer determines the desired test pattern, puts signal assignments in the process to set up that condition, and then waits for some time. The waits above are shown to be 50 ns, but they could be whatever the designer wants. After some time has been left to observe what is happening, then the inputs are changed, and another wait called for. And the process continues until all the patterns are tested. Then, the process will begin again and go thru the activities again.