Introduction to VHDL Modular design

Jorge Ortiz

October 2007

Generic procedure for any design implementation

1. **Design Description:** Figure out what problem you are trying to solve. What is it supposed to accomplish? What different steps are needed? Can we use divide and conquer to simplify particular parts of the design?
2. **Interface Description:** Your design will have input and output sources. Figure out what they are, and how many bits each of them will need. Draw a block diagram showing only the inputs (incoming arrows) and outputs (outgoing arrows) of your design.
3. **Problem Decomposition:** Rarely will a design be atomic in nature. Figure out what level of decomposition you need to be able implement your design's main components. Some examples of component implementation level (in decreasing order of granularity) are listed below:

   Your design components can be:

   a. Other complex designs
   For example CPU's have complex components like ALUs, Register Files, Interface and Control.

   b. Other simple designs
   ALU's can be implemented from simple designs like adders, multipliers, and bitwise Boolean operators.

   c. Register Transfer Level
   State Machines can be expressed as synchronous RTL registers and some combinatorial logic.

   d. Gate Level
   Boolean expressions use gates like OR, AND and XOR to provide their functionality

   e. Transistor Level
   Integrated circuits (IC's) would use a transistor description for their overall design for implementation.

   f. Silicon Level
   Transistors, in turn, can be expressed as the connection of semiconductor components.

Once you have decomposed your design into more manageable parts, you have to implement each individual component. The steps to follow to create each component entity will be exactly the same as what you did for your overall problem. Describe your component design, establish the component interface, and decompose the entity into more components. Repeat Steps 1-2-3 for each component until you have a component that does only one particular operation (and does it well!). It's easier to evaluate the correct behavior of each individual component. Further, each specialized component can be reused for further, or different, designs.

There are two main ways to attack any design implementation:

Top-down
We know what the top level entity will do, how it will do it, and how each separate subsystem will contribute to the solution, though we don't know how each component will work internally. Therefore, we describe the top-level entity, and keep decomposing into components. Once we know who all the components are, we implement their behaviors.

Bottom-up
We have the basic idea of what the top-level entity will do, but not how the components must be organized. However, we have a good notion of what the components will be, so we implement their behavior and use them iteratively in larger components until each subsystem is fully implemented.

VHDL approach and examples

For this particular tutorial, we'll use the bottom-up approach. The bottom-level components can be easily implemented in behavioral VHDL, though we don't know how to connect them together to form larger components (that's why you are reading this!)

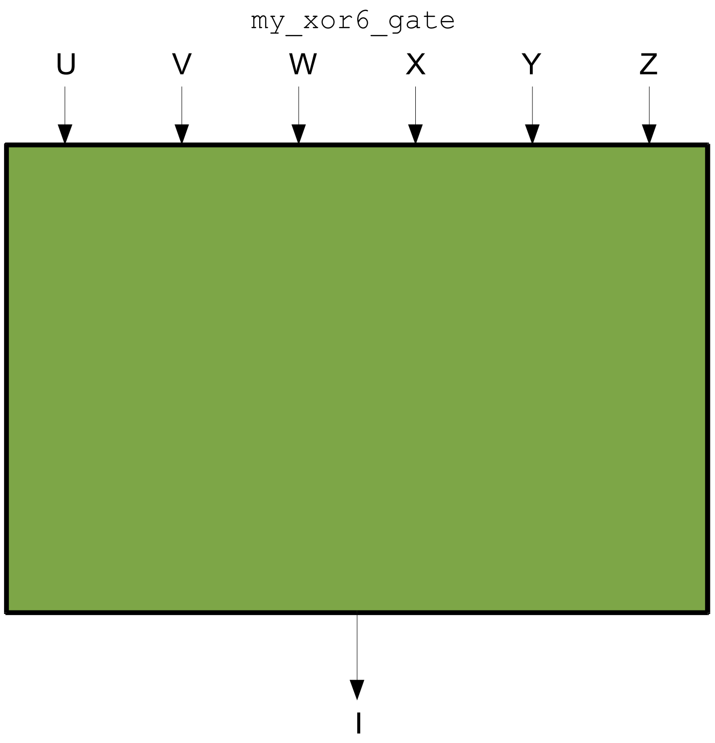
For this example let's take a seemingly difficult circuit, describe it, decompose it, and implement it from the bottom-up.

1 - The top-level Entity

Step 1: Design Description: We will make an entity that will accept six bits of input, and output the result of a 6-way XOR-gate applied to those 6 bits

Step 2: Interface Description: There's obviously only 1 bit of output. As for the inputs, you can consider the 6 bits to be either (a) Six 1-bit inputs or (b) One 6-bit long input. Let's have the interface be six 1-bit inputs (we will do the one 6-bit long input case at the end of this tutorial)

Entity Block Diagram 1
Inputs: U,V, W, X, Y, Z
Output: I



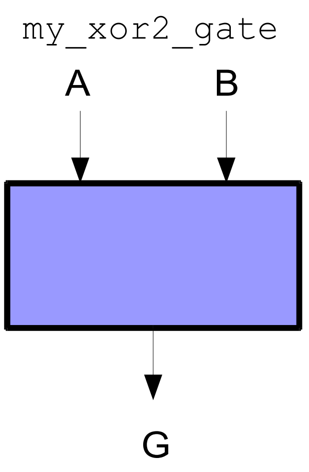
Step 3: Problem Decomposition: Clearly, an XOR gate would be needed somewhere in the design. We're unsure which other components might be needed, so let's start by creating a 2-bit XOR gate.

2- The 2-bit XOR Gate

Step 1: Design Description: An exclusive or gate between inputs A and B yields true whenever only one of the inputs is true. This can be expressed as G = A xor B = A'B or AB'

Step 2: Interface Description: Let's use the names from step one.

Entity Block Diagram 2
Inputs: A and B, Output: G



Step 3: Problem Decomposition: Using the knowledge that VHDL has the keywords AND, OR, and NOT, we can go ahead and implement this component behaviorally.

VHDL Implementation

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

ENTITY my_xor2_gate IS
   PORT (
      A,B : IN STD_LOGIC;
      G   : OUT STD_LOGIC);
END my_xor2_gate;

ARCHITECTURE Behavioral of my_xor2_gate IS
BEGIN
   G <= (A AND (NOT B)) OR ((NOT A) AND B);
END ARCHITECTURE Behavioral;
```

We have our first component! Let’s create another XOR gate with more inputs to reach our goal of a 6-bit input XOR gate.

3- The 3-bit XOR gate

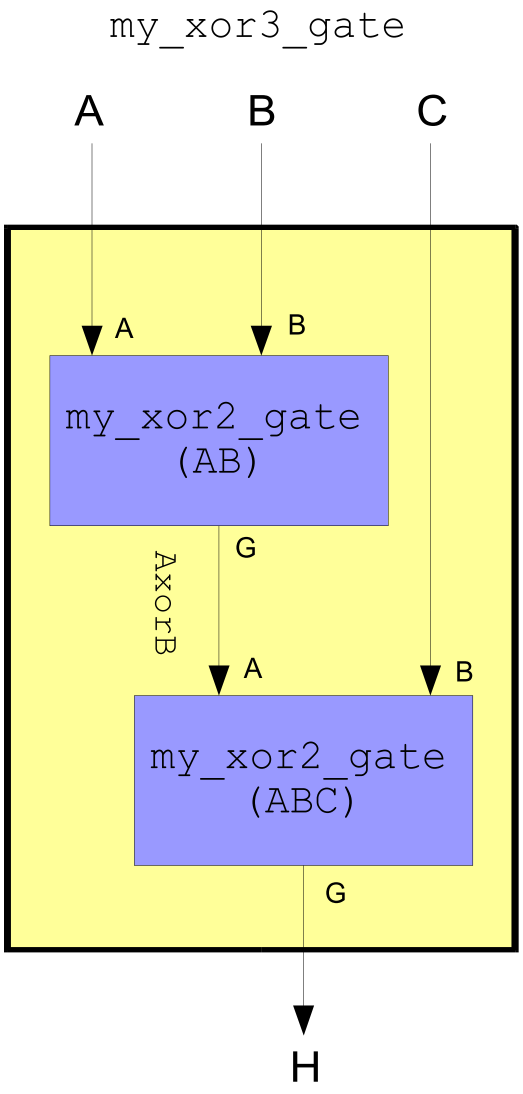
Step 1: Design Description: We make use of the associativity property of XOR gates to figure out that:

H = A xor B xor C = (A xor B) xor C = G xor C

Where 'G' can be the output of our 2-bit xor gate. Hence, we will use `my_xor2_gate` as a `COMPONENT` for our `ENTITY` implementation of `my_xor3_gate`. We will then `INSTANTIATE` it twice to make a 3-bit xor gate.

Step 2: Interface Description: Let's use the names from step one.

Entity Block Diagram 3
Inputs: A, B and C, Output: H



Step 3: Problem Decomposition: We now use our previous `my_xor2_gate` design as a component. This is called `COMPONENT DECLARATION` and it happens immediately after the `ARCHITECTURE` declaration line. The component declaration should match exactly the entity declaration that we made when creating the component.

Also, we need to refer to the 'G' output of the top-left `my_xor2_gate` (AB) by name (because it is neither an input nor an output for the `my_xor3_gate` `ENTITY`), so we declare a signal for it. The signal will be called AxorB. The `SIGNAL DECLARATION` happens also immediately after the `ARCHITECTURE` declaration line.

Partial VHDL Implementation

```
ENTITY my_xor3_gate IS
   PORT (
      A,B,C : IN STD_LOGIC;
      H     : OUT STD_LOGIC);
END my_xor3_gate;

ARCHITECTURE Structural of my_xor3_gate IS

   -- Component Declaration
   COMPONENT my_xor2_gate IS
      PORT (
         A,B : IN STD_LOGIC;
         G   : OUT STD_LOGIC);
   END COMPONENT my_xor2_gate;
   -- Signal Declaration
   SIGNAL AxorB: STD_LOGIC;

BEGIN

   -- (...To be completed)

END ARCHITECTURE Structural;
```

Notice that now we are using a structural description of the architecture of this entity. This is because we are using components instead of behavioral descriptions to describe how our entity works. To use a structural architecture, we must do three things:

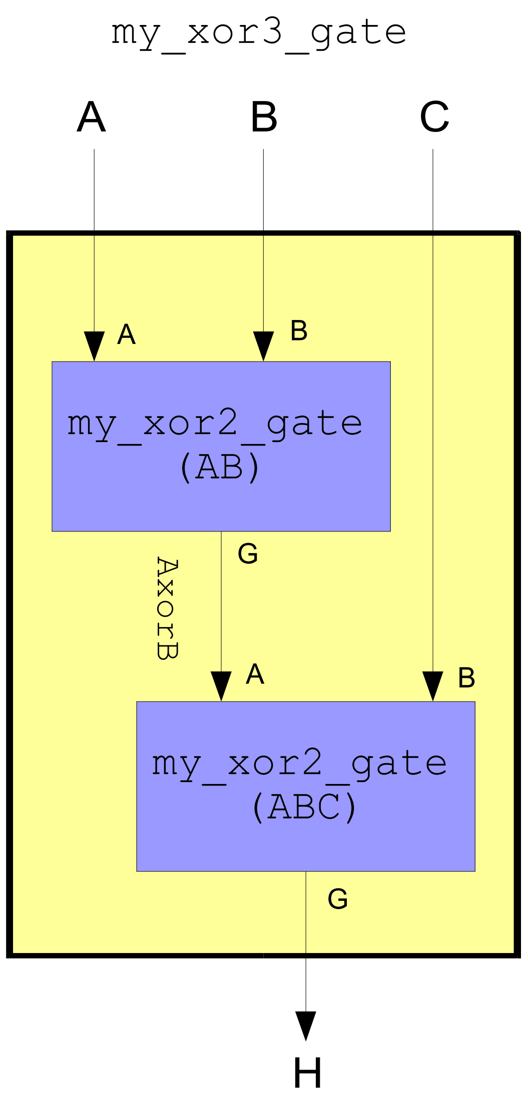
1- Component declaration (already done): This defines the input and output ports for the component(s).

2- Component instantiation: For each component used, we assign a name to them. In the above example, we use AB and ABC for the two `my_xor2_gate`'s.

3- Port mapping: We connect each port of each instantiated component. These connections can be made directly to the encapsulating ENTITY's inputs and outputs, or to internal SIGNALS.

Complete VHDL Implementation

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

ENTITY my_xor3_gate IS
   PORT (
      A,B,C : IN STD_LOGIC;
      H     : OUT STD_LOGIC);
END my_xor3_gate;

ARCHITECTURE Structural of my_xor3_gate IS

   -- Component Declaration
   COMPONENT my_xor2_gate IS
      PORT (
         A,B : IN STD_LOGIC;
         G   : OUT STD_LOGIC);
   END COMPONENT my_xor2_gate;
   -- Signal Declaration
   SIGNAL AxorB: STD_LOGIC;

BEGIN
   -- Component Instantiation
   AB: my_xor2_gate
      PORT MAP (
         A => A,
         B => B,
         G => AxorB);

   ABC: my_xor2_gate
      PORT MAP (
         A => AxorB,
         B => C,
         G => H);

END ARCHITECTURE Structural;
```



Notice that Ports A and B of `my_xor2_gate` instance 'AB' connect directly to the entity's `my_xor3_gate` input ports A and B. The output port G of 'AB' connects to the signal 'AxorB'. In turn, port A of instance 'ABC' connects to signal 'AxorB' (connecting the two instances), while port B of instance 'ABC' connects to the entity's input Port C. The output G of instance 'ABC' connects to the overall entity output H.

Be sure to keep good track of component port names and entity port names, especially when dealing with multiple instances of the same declared component.

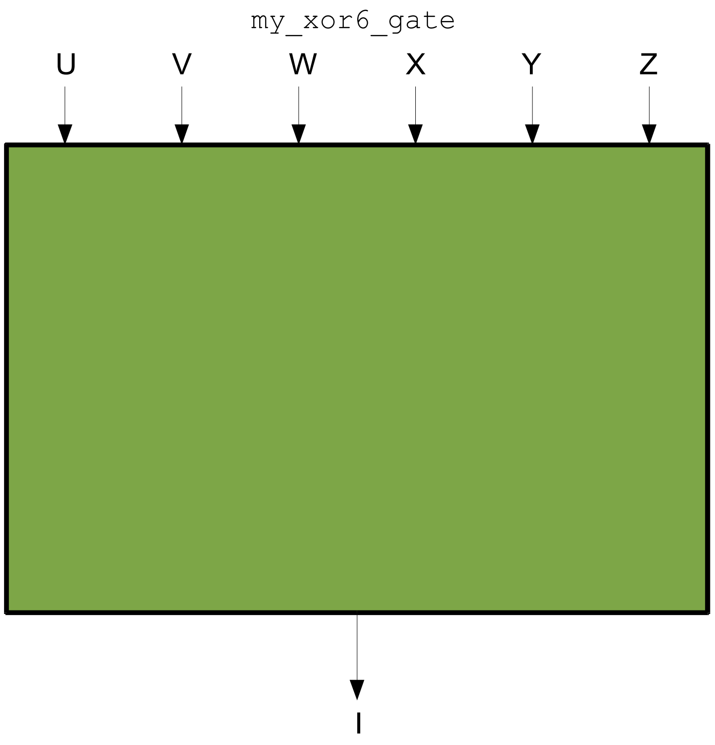
We use the associativity property of XOR again to go directly to our final top-level implementation.

4 - The top-level Entity (revised)

Step 1: Design Description: We will make an entity that will accept six bits of input, and output the result of a 6-way XOR-gate applied to those 6 bits

Step 2: Interface Description: There's obviously only 1 bit of output. As for the inputs, you can consider the 6 bits to be either (a) Six 1-bit inputs or (b) One 6-bit long input. Let's have the interface be six 1-bit inputs (we will do the one 6-bit long input case at the end of this tutorial)

Entity Block Diagram 4
Inputs: U,V, W, X, Y, Z
Output: I



Step 3: Problem Decomposition: Using the associativity of XOR, we can see that:

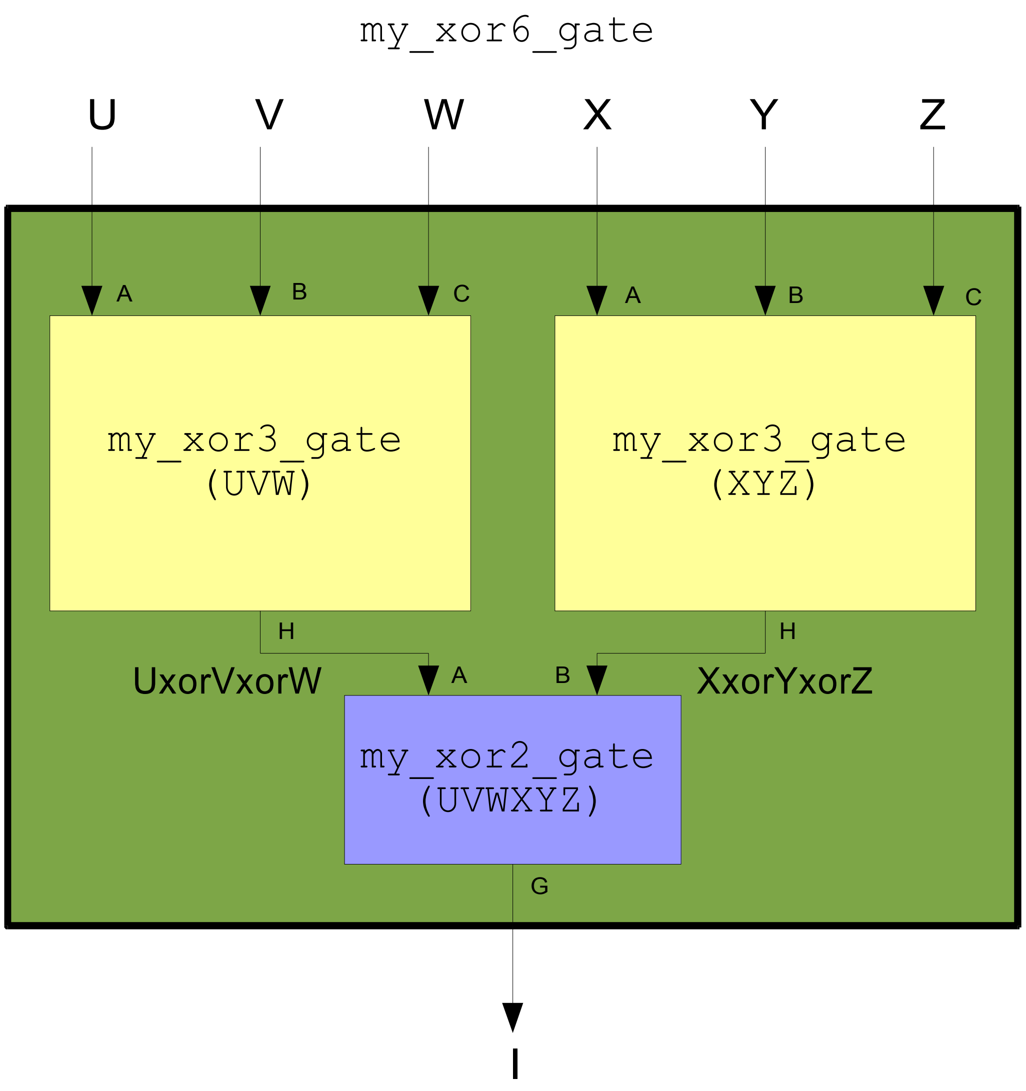
I = U xor V xor W xor X xor Y xor Z = (U xor V xor W) xor (X xor Y xor Z)

So we can use both of our previous components, `my_xor2_gate` and `my_xor3_gate` to accomplish this.

Entity Block Diagram 5

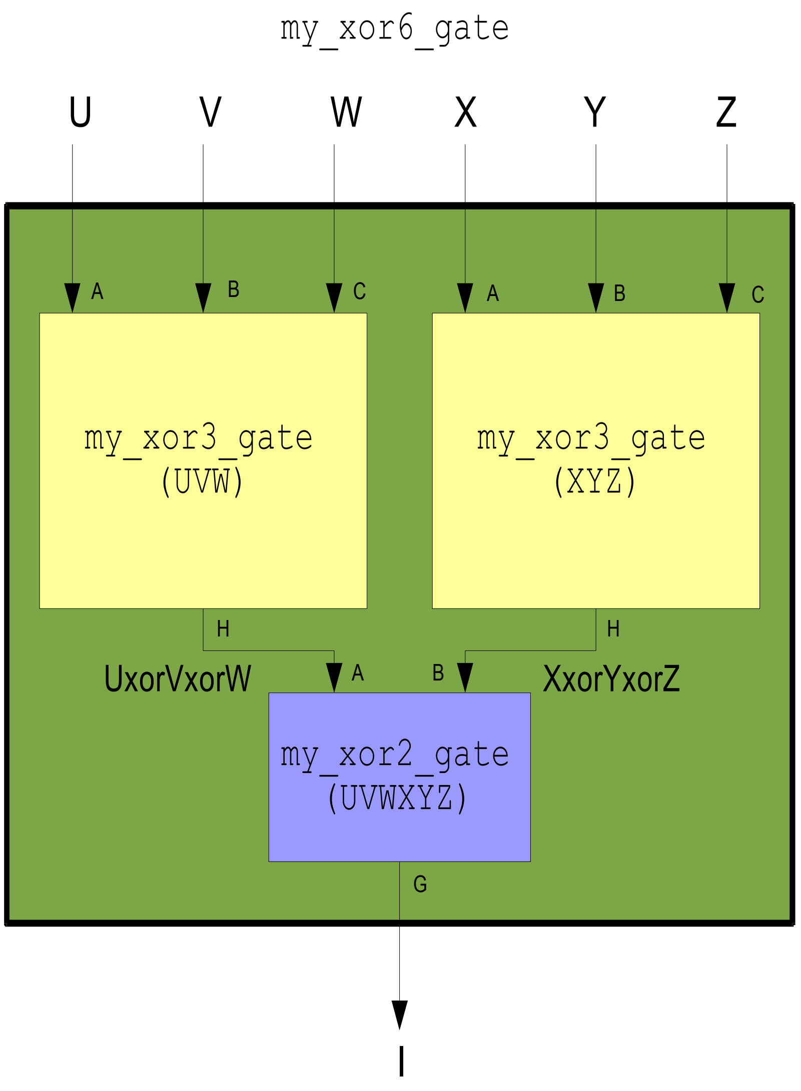
Inputs: U,V, W, X, Y, Z

Output: I



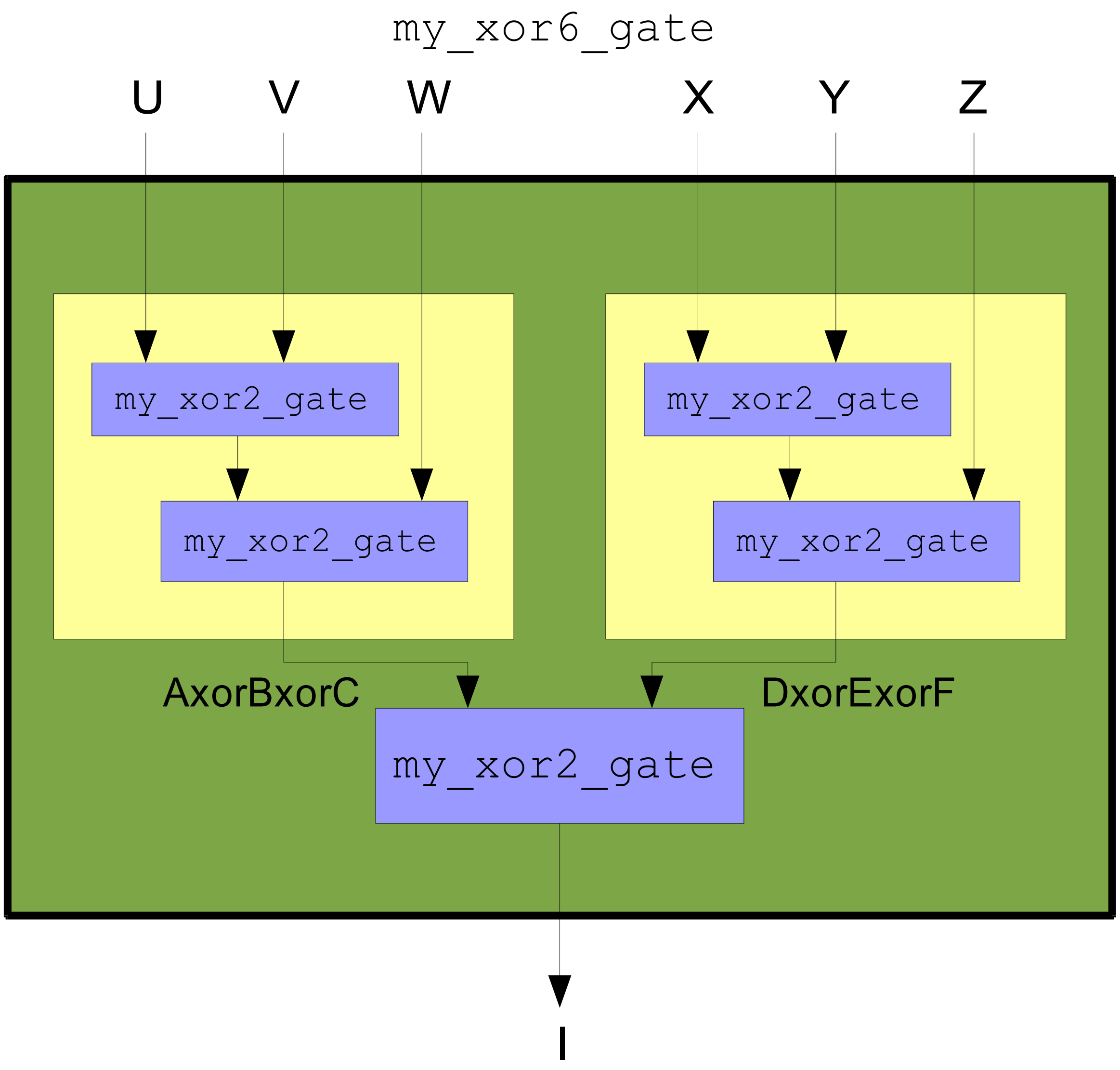
Complete VHDL Implementation

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

ENTITY my_xor6_gate IS
   PORT (
      U,V,W,X,Y,Z: IN STD_LOGIC;
      I          : OUT STD_LOGIC);
END my_xor6_gate;

ARCHITECTURE Structural of my_xor6_gate IS

   -- Component Declaration

   COMPONENT my_xor2_gate IS
      PORT (
         A,B : IN STD_LOGIC;
         G   : OUT STD_LOGIC);
   END COMPONENT my_xor3_gate;

   COMPONENT my_xor3_gate IS
      PORT (
         A,B,C : IN STD_LOGIC;
         H     : OUT STD_LOGIC);
   END COMPONENT my_xor3_gate;

   -- Signal Declaration
   SIGNAL UxorVxorW: STD_LOGIC;
   SIGNAL XxorYxorZ: STD_LOGIC;

BEGIN
   -- Component Instantiation
   UVW: my_xor3_gate
      PORT MAP (
         A => U,
         B => V,
         C => W,
         H => UxorVxorW);
   XYZ: my_xor3_gate
      PORT MAP (
         A => X,
         B => Y,
         C => Z,
         H => XxorYxorZ);

   UVWXYZ: my_xor2_gate
      PORT MAP (
         A => UxorVxorW,
         B => XxorYxorZ,
         G => I);
END ARCHITECTURE Structural;
```



We used multiple components (that we created from the bottom-up) to accomplish our final design. Additionally, our new entity `my_xor6_gate` can be used as a component for higher input XOR gates or different designs. If we hadn't used modular design, our final equation for I = fn(U,V,W,X,Y,Z) would have been long, ugly and prone to error.

For completeness, here is how the final design looks like (considering that the `my_xor3_gate` components are composed of `my_xor2_gate`s)



5 - The top-level Entity (using one 6-bit input)

VHDL has support for the XOR keyword (and now we tell you!). It also has support for STD_LOGIC_VECTORs, which means multiple bits can addressed as one input/output/signal. Further, we can address each individual bit of a STD_LOGIC_VECTOR.

Hence, the top level implementation could have been implemented behaviorally as shown below:

Behavioral VHDL Implementation

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

ENTITY my_xor6_gate IS
   PORT (
      X : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
      Y : OUT STD_LOGIC);
END my_xor6_gate;

ARCHITECTURE Behavioral of my_xor6_gate IS
BEGIN
   Y <= X(5) XOR X(4) XOR X(3) XOR X(2) XOR X(1) XOR X(0);
END ARCHITECTURE Structural;
```