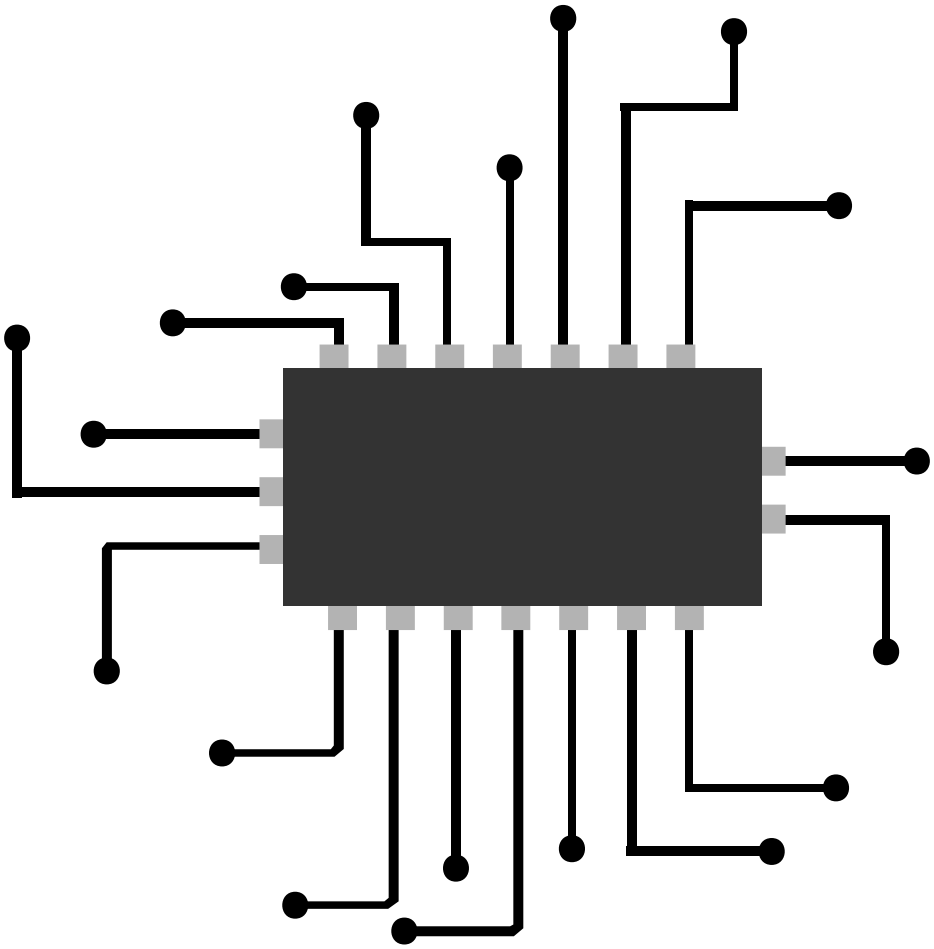


Beginning VHDL

A Hands-on Approach to Digital Design

Tim Shuck

2020-06-28

Table of Contents

Chapter 1. Introduction

Typically used in conjunction with programmable logic, VHDL, or VHSIC Hardware Description Language, is a text-based method of describing electronic hardware.

VHDL began as a project under the directive of the U.S. Department of Defense as a means to document the behavior of electronic devices. It wasn't long after that simulators were developed to allow one to simulate the behavior of the descriptions contained in VHDL files.

VHDL was designed around the same time as the Ada programming language. The Ada programming language was also developed by the U.S. DoD as a replacement for the various languages being used and lent much of its syntax to the VHDL language in order to mitigate drastically differing syntax among users that might have to learn both.

Chapter 2. VHDL Syntax Basics

2.1. Introduction

A digital system can be described by its external connections and its behavior. In other words, by describing the interface of a device and how that device uses inputs to derive outputs (its behavior), we can describe the functionality of a given device. This ability to describe the functionality of a digital device allows for the creation of a simulator that can simulate the interaction of many devices (modeled).

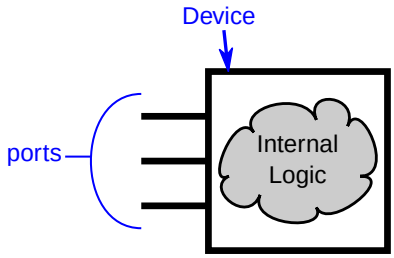


Figure 2.1 A blackbox entity can be used to describe the functionality of a device.

If we define a module as a device with internal behavior and external ports (connections), we can represent a module as a black box with some connections to the world outside of the module.

If we describe this module in VHDL-speak, we would say that a module has an **entity** and an **architecture**.

The **entity** portion of the model describes the connections to the module - how many inputs/outputs, what kind of connection, etc. The **architecture**, on the other hand, describes the internal logic of the device - given a set of input values, what should the output(s) be?

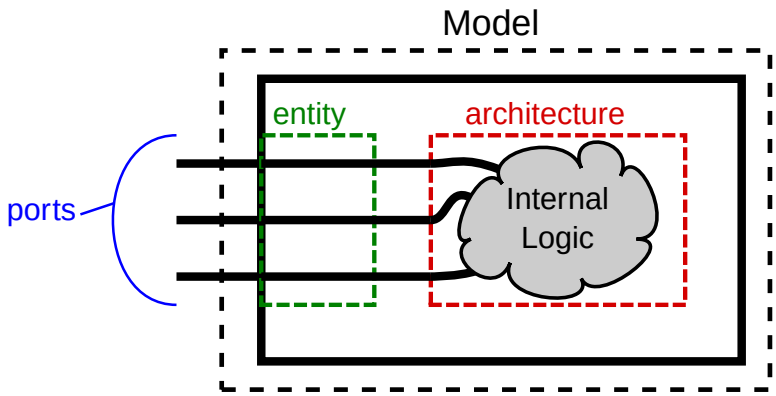


Figure 2.2 A VHDL model is comprised of an **entity** *(how it connects to the outside world) and the* **architecture** *(the internal behavior of the device).*

2.2. A Simple Example to Explain Some Basics

Let's take a look at an example that should help us to get a practical understanding of what VHDL is and does.

Imagine that we take a circuit with a given truth table, Table 2.1:

Table 2.1 The truth table for an AND function.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

We can represent the model of this circuit as a box with some inputs and outputs as before, but we will define the internal logic so as to implement the truth table, where A and B are inputs and Y is the output.

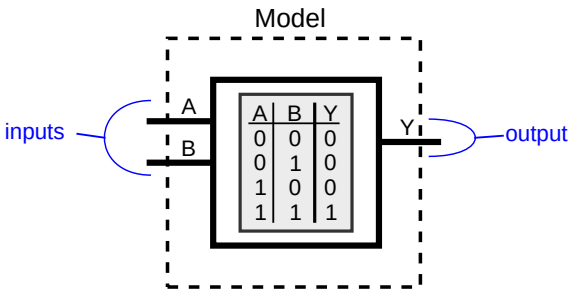


Figure 2.3 The architecture for this device implements the truth table, the entity specifies which signals are inputs (A/B) and which are outputs (Y).

Inputs are most often placed to the left side of a model and outputs are typically placed on the right of the model.

If you have not noticed, this truth table is that of an AND gate. This means that our architecture logic must simply implement and AND operation between A and B, or, expressed mathematically, $Y = A \cdot B$.

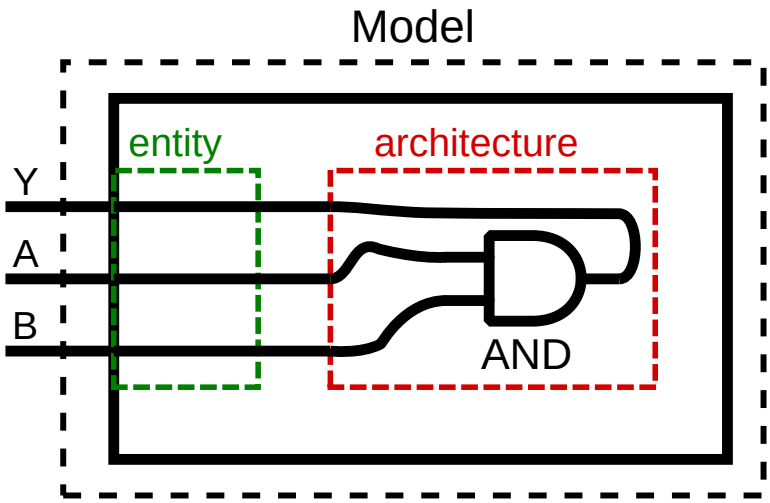


Figure 2.4 This model implements the AND operation.

While this AND gate may be a fairly trivial example, it serves to show some important aspects to understand VHDL. Let us look at the overall VHDL code for this circuit and dissect it accordingly in the following section.

Listing 2.1 ANDGATE.vhd

```
-- (this is a VHDL comment)

-- import std_logic from the IEEE library
library IEEE;
use IEEE.std_logic_1164.all;

entity ANDGATE is
  port (
    Y : out std_logic;
     A : in std_logic;
     B : in std_logic -- <=no semicolon
  );
end entity ANDGATE;

-- this is the architecture
architecture and1 of ANDGATE is
begin
  Y <= A and B;
end architecture and1;
```

You will notice that each statement has a semicolon after it. Note, however, that the last entry in the port section does not have a semicolon after it — it is taken care of by the closing of the port declaration.

2.2.1. Entity, architecture, and libraries, Oh my!

Let's take a closer look at three of the major pieces of a VHDL file:

- entity
- architecture
- libraries and packages

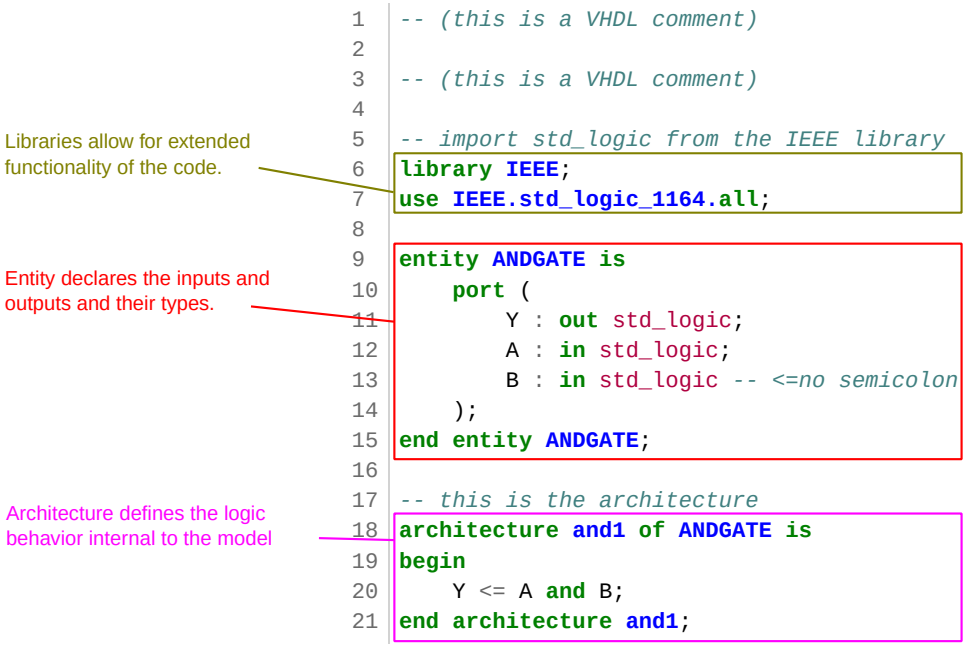


Figure 2.5 The three parts of a VHDL file: entity, architecture, and library inclusions.

Entity

The **entity** describes the way the component interfaces with the outside world. A component can only ever have one entity — it tells other components how many connections and what kind of connection a given device has.

The general form of the entity declaration is as follows:

```
entity <entity_name> is
  generic (
    -- Generics Declarations - we'll talk more about these later
  );
  port (
    <signal_name_1> : <mode> <signal_type>;
    <signal_name_2> : <mode> <signal_type>;
    ...
    <signal_name_n> : <mode> <signal_type>
  );
end entity [<entity_name>];
```

If we compare this formal definition to the AND gate we created earlier, we can see that there are three ports declared in the AND gate's entity declaration: Y, A, and B.

The ports A and B are inputs, Y is an output, and all are of type std_logic:

```
9 Y : out std_logic;
10 A : in  std_logic;
11 B : in  std_logic -- <=no semicolon
```

We can also see in Figure 2.6 that the entity is named "ANDGATE".

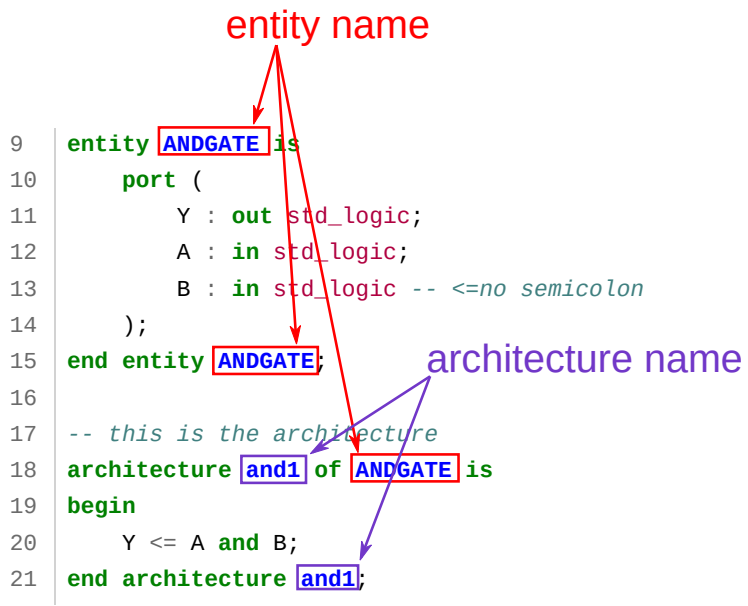


Figure 2.6 A model contains the name of the entity ('ANDGATE') and the name of one, or more, architectures ('and1').

Port Modes

VHDL requires that ports in an entity have a mode, or direction, associated with them. There are 4 modes that a port can have, seen in Table 2.2:

Table 2.2 Entity port modes.

Mode	Description
in	the signal is an input to the module
out	the signal is an output from the module

Mode	Description
buffer	an output that can be read internal to the module*
inout	the signal can be both input and output; bidirectional

* **buffer** mode has limited, often noncompliant, support across available tools. Use mode **inout** instead.

While the logic of an **out** port can be modified within a model, the signal value cannot be read within the model's architecture. We will look at how to work around this drawback shortly.

Architecture

If the entity is where the external connections are defined, then the architecture is where the internal logic is defined.

The architecture defines what happens to the signals specified in the entity as well as any intermediate signals that you may need to create.

The architecture has the general form of:

```
architecture of <name_of_the_entity> is
   -- Declarations - we'll talk more about these later
begin

   -- define the behavior of the entity here


end architecture <architecture_name>;
```

While a model can only have one definition of its external interface (its entity section), a model can have multiple architectures (internal logic definitions), though only one is ever used at a given time during simulation or synthesis (the process of implementing the VHDL in hardware).

Each time a copy of a component is used in a design, that is, the component is **instantiated**, a different architecture can be used.

Because there can be multiple architectures, it is essential to specify the entity an architecture will use (as seen on line 16):

```
16 architecture and1 of ANDGATE is
17 begin
18    Y <= A and B;
19 end architecture and1;
```

The architecture requires:

- a unique name
- the name of the entity describing its external connections
- the begin keyword
- the end of the architecture.

In our example, the architecture is named and1 and the referenced entity is ANDGATE. We closed the architecture with the line “end architecture and1;” — making sure to specify the name of the architecture we are closing.

In this architecture, the internal logic of the model is defined on line 18:

```
18 Y <= A and B;
```

We have assigned Y to the logical AND of A and B using the 'and' operator keyword. The 'and' keyword performs the AND operation on the values to it’s left and right in this case, A and B, meaning when both A and B are '1', the resulting value is '1', and is '0' otherwise. This value is then assigned to Y.

VHDL defines several basic, Boolean operations using the operators name spelled out (e.g. and, or, xor, etc.) See VHDL Operators for information

Libraries and Packages

At the beginning of our AND gate model, we see some stuff that appears to be referencing a library in the following code:

```
3 -- import std_logic from the IEEE library
4 library IEEE;
5 use IEEE.std_logic_1164.all;
```

This is, indeed, referencing a library — the IEEE library. The IEEE library has many parts, each with a more specific purpose called **packages**. The IEEE library's std_logic_1164 package contains a definition for the std_logic type we have used in our entity declaration for port types.

In general, libraries contain many packages. We specify which packages we want and what part of the package is needed for a given model (though we may often include all of the package for brevity).

Chapter 3. Hierarchical Modeling: Putting things together

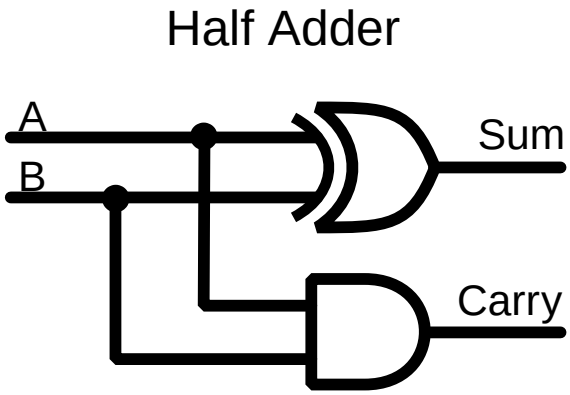
3.1. Introduction

As you may already have found, modeling simple devices like an AND gate is not terribly interesting.

In this section, we will look at how to combine models in hierarchical arrangements, or creating models that contain other models, in order to produce more complicated circuits and how to simulate the models using what is called a testbench.

3.2. Combining Models

Admittedly, the example we came up with in the previous section didn't really **do** anything. An AND gate is rarely useful by itself and must be combined with other devices to do something more interesting.



A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 3.1 A half adder circuit.

Figure 3.1 shows a common, simple application where one might combine an AND gate and another gate (XOR) in order to implement another circuit—a half adder.

A half adder has a sum and carry term. Easily enough, the sum term is 1 if the inputs are not equal—which is the XOR function, and the carry term is only 1 if A and B are 1, which corresponds to the AND function. As such, the half adder is comprised of an AND gate and an XOR gate, both of which share the same inputs.

3.2.1. Creating the XOR Model

Listing 3.1 XORGATE.vhd

```
1 -- import std_logic from the IEEE library
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4
5 entity XORGATE is
6   port (
7     Y : out std_logic;
8     A : in std_logic;
9     B : in std_logic -- <=no semicolon
10   );
11 end entity XORGATE;
12
13 -- this is the architecture
14 architecture xor1 of XORGATE is
15 begin
16   Y <= A xor B;
17 end architecture xor1;
```

Take a moment and compare the XOR gate code to that of the AND gate we defined in the previous section. Notice that the entity names are different, yet they have the same port definitions — this is because they are both two-input, one output gates. We could, however, have named the ports whatever we want, though it is a good idea to keep a common representation for similar devices for your own understanding.

If we look at the architecture of the XOR gate, not much changes — the architecture name, the entity name that the architecture is defining, and the assignment Y ⇐ A xor B;.

It is the XOR operation on line 16 that changes the behavior from the AND gate model to implement the XOR function.

3.2.2. Creating the Half Adder Model

Now that we have both the XOR and AND models created, we need to combine them to form the half adder model — this is where the hierarchical modeling comes in to play.

Listing 3.2 HALF_ADDER.vhd

```
1 -- import std_logic from the IEEE library
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4
5 entity HALF_ADDER is
6   port (
7     Sum : out std_logic;
8     Carry : out std_logic;
9     A : in std_logic;
10     B : in std_logic -- <=no semicolon
11   );
12 end entity HALF_ADDER;
13
14 -- this is the architecture
15 architecture default of HALF_ADDER is
16   -- here we define the ANDGATE component - essentially just the entity
  information
17   component ANDGATE is
18     port (
19       Y : out std_logic;
20       A : in std_logic;
21       B : in std_logic
22     );
23   end component;
24
25   -- here we define the XORGATE component - essentially just the entity
  information
26   component XORGATE is
```

```
27      port (
28          Y : out std_logic;
29          A : in std_logic;
30          B : in std_logic
31      );
32    end component;
33  begin
34
35    -- instantiate a 'XORGATE' model and call this instance 'sumGate'
36    sumGate: XORGATE
37      port map (
38          Y => Sum, -- connect the Y output of this XORGATE to the 'Sum' signal
39          A => A,   -- connect the A input of this XORGATE to the 'A' signal
40          B => B    -- connect the B input of this XORGATE to the 'B' signal
41      );
42
43    -- instantiate an 'ANDGATE' model and call this instance 'sumGate'
44    carryGate: ANDGATE
45      port map (
46          Y => Carry, -- connect the Y output of this ANDGATE to the 'Carry' signal
47          A => A,     -- connect the A input of this ANDGATE to the 'A' signal
48          B => B      -- connect the B input of this ANDGATE to the 'B' signal
49      );
50
51  end architecture default;
```

Parts of the Half Adder Model

Let's break the half adder model we just created down a little more to help explain what each of the parts mean.

Component Definition

The component definitions describe all of the component used within a given VHDL model. In this example, the half adder uses both the XORGATE and ANDGATE models, so

the architecture of the half adder must define the components.

The component declaration simply tells the instantiating model what the interface is like between the model and its component models (ANDGATE and XORGATE) — as such, it is the **entity** information of each of the models, but with the entity keyword replaced by the **component** keyword.

Compare lines 17-23 of the half adder with the entity section for the ANDGATE model and see that they have the same information.

Model Instantiation

Instantiation is a complicated-sounding word that simply means 'create an instance of something', so don't get thrown just yet.

We see lines 35-41 serve to create an instance of the XORGATE model we created before:

```
35 -- instantiate a 'XORGATE' model and call this instance 'sumGate'
36 sumGate: XORGATE
37    port map (
38       Y => Sum, -- connect the Y output of this XORGATE to the 'Sum' signal
39       A => A,   -- connect the A input of this XORGATE to the 'A' signal
40       B => B    -- connect the B input of this XORGATE to the 'B' signal
41    );
```

On line 36, we see that we are creating an instance of the XORGATE model within the half adder named 'sumGate', or, to put it graphically:

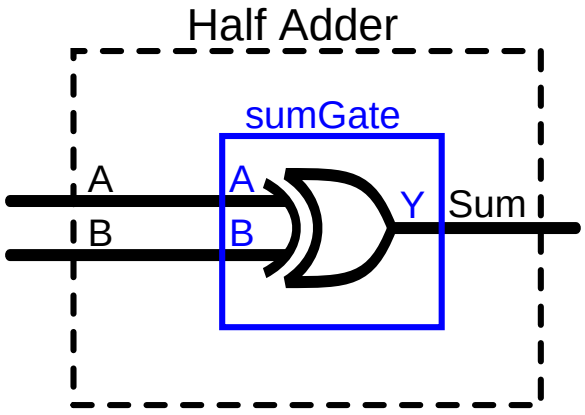


Figure 3.2 The XORGATE model definition is instantiated with the name 'sumGate'.

We can see this XORGATE instance's port map on lines 37-41. The port map simply tells the half adder model how to connect the ports defined in the XORGATE model to the signals in the half adder.

If you look to the left of the '=>', you will see the name of the port as defined in the XORGATE definition. To the right of the '=>', you will see the name of the signal that the XORGATE-defined port connects to in side the half adder.

Table 3.1 Signal mapping for the instantiated model.

XORGATE-defined Port	Half Adder Signal
Y	Sum
A	A
B	B

The process of defining what port connects to what signal is typically referred to as **mapping** the port — hence the port map (...) section in the code. This mapping function allows us to map the port names of a model (A/B/Y) to the signal names within a higher order model.

By instantiating the XORGATE model with a name, we could create many more instances of the XORGATE model with different names, and they would all function independently of one another, implementing the XORGATE logic on whatever connects to each instance's A,B, and Y ports.

This same process is applied to the ANDGATE model used in the half adder code, resulting in a model like that shown in Figure 3.3.

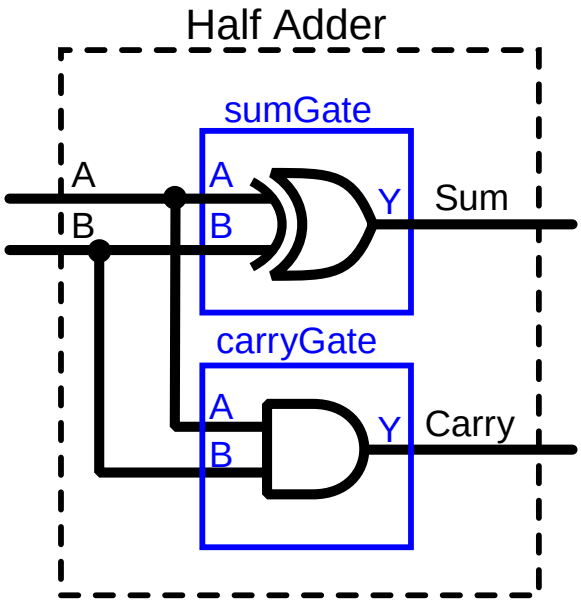


Figure 3.3 The half adder implemented in VHDL.

3.2.3. Creating a Testbench

At this point, we have a half adder model comprised of XOR and AND gate models, but we haven't actually verified it's behavior — something that can be tested with the help of a **testbench**.

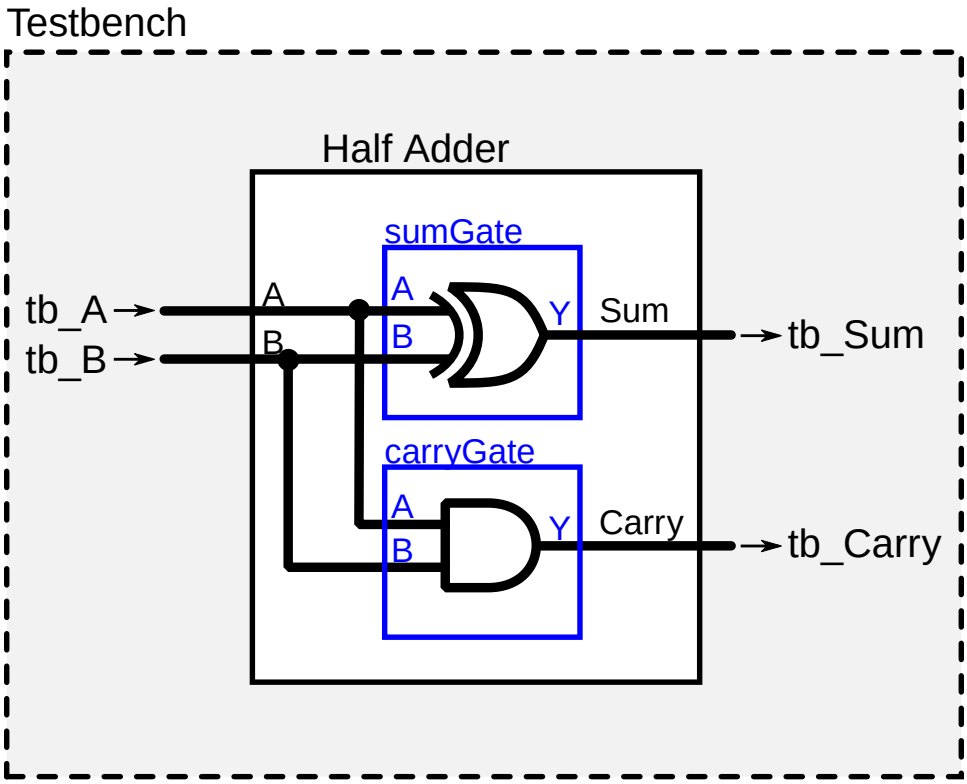


Figure 3.4 A testbench is a model that has no ports and contains other models within it.

A testbench is a complicated-sounding name for a very simple construct — it's just another model that contains the other model(s) to simulate.

Look at Figure 3.4, the testbench has no ports and contains the half adder we created earlier.

Below is the code for the half adder testbench:

Listing 3.3 HALF_ADDER_TB.vhd

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity HALF_ADDER_TB is
5    -- no entity ports for a testbench!
6 end entity HALF_ADDER_TB;
7
8 architecture default of HALF_ADDER_TB is
9  -- here we define the half adder component
```

```
10  -- (only components instantiated in this file need to be defined)
11  component HALF_ADDER is
12     port (
13        Sum : out std_logic;
14        Carry : out std_logic;
15        A : in std_logic;
16        B : in std_logic -- <=no semicolon
17     );
18  end component;
19
20  -- these signals only exist within this testbench
21  signal tb_A : std_logic;
22  signal tb_B : std_logic;
23  signal tb_Sum : std_logic;
24  signal tb_Carry : std_logic;
25
26 begin
27
28  -- instantiate a HALF_ADDER model and call this instance 'ha'
29  ha: HALF_ADDER
30     port map (
31        Sum   => tb_Sum,
32        Carry => tb_Carry,
33        A     => tb_A,
34        B     => tb_B
35     );
36
37     process
38     begin
39        tb_A <= '0';
40        tb_B <= '0';
41        wait for 5 ns;
42        tb_A <= '1';
```

```
43          tb_B <= '0';
44          wait for 5 ns;
45          tb_A <= '0';
46          tb_B <= '1';
47          wait for 5 ns;
48          tb_A <= '1';
49          tb_B <= '1';
50          wait for 5 ns;
51          wait; -- wait forever
52      end process;
53
54 end architecture default;
```

This testbench has some internal signals declared (see lines 21-24). It was mentioned before that a testbench does not have any ports, that is, external connections, but they will generally have internal signals.

These internal signals allow us to connect testbench level signals to the half adder ports.

Table 3.2 Testbench signal mapping.

Half adder port	Testbench signal
Sum	tb_Sum
Carry	tb_Carry
A	tb_A
B	tb_B

Looking at lines 37-52, a process is used to control the inputs to the half adder by setting tb_A and tb_B to various values. We'll go more in depth on what process is later, however, for now, simply understand that this process initially sets tb_A and tb_B both both to '0'. These signals are toggled through all of the input combinations, changing state every 5ns.

If your design is correct, you should come up with the following truth table:

Table 3.3 Testbench truth table.

tb_A	tb_B	tb_Sum	tb_Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Appendix A: Using GHDL and GTKWave

A.1. Introduction

While any VHDL simulator can be used to simulate these designs, we will use an open-source process to simulate and view the results.

GHDL is an open source VHDL simulator that we can use to simulate the code that we developed earlier. We can use this simulator to generate a value-change dump (VCD) file. A VCD file captures the value of signals within the design and the times during the simulation that they change. This VCD file can be viewed using another open source software called GTKWave.

The GHDL/GTKWave flow is shown in Figure 4.1.

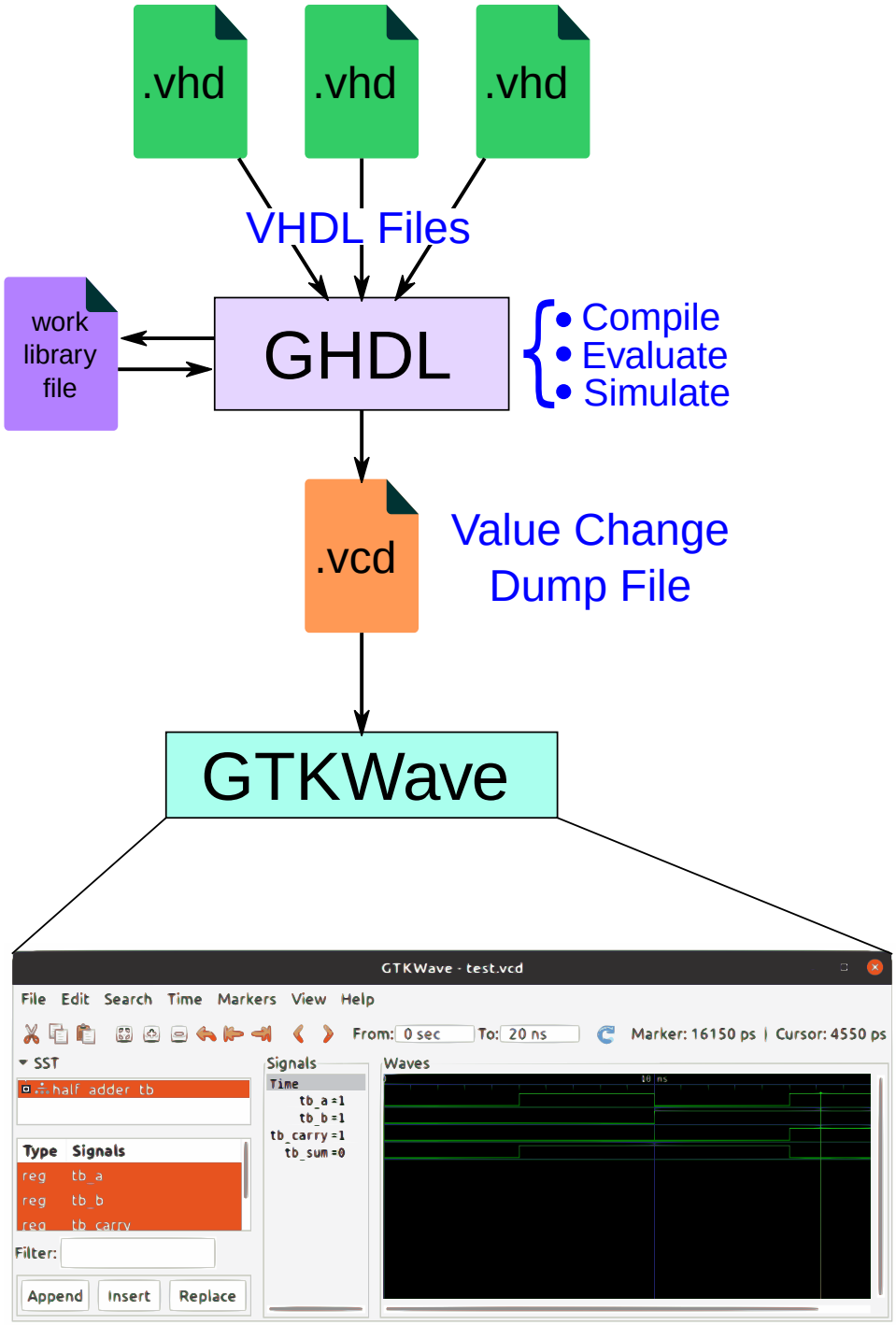


Figure 4.1 GHDL is used to compile, evaluate, and simulate a design, creating a VCD that can be viewed by GTKWave.

A.2. Using GHDL to Simulate the Design

To simulate a design with GHDL, we need to:

1. **Analyze the design:** converts the source code into a form that GHDL can evaluate.
2. **Evaluate the design:** creates a program that can be run by the computer.
3. **Run the simulation:** executes the evaluated design and generates waveforms.

To perform each of these steps, GHDL is called with different flags: -a to analyze, -e to evaluate, and -r to run the simulation.

A.2.1. Analyzing Files with GHDL

The first step to simulate with GHDL is to analyze all of the source files needed for the design.

The GHDL command to analyze source file(s) is:

```
$ ghdl -a <source file(s)>
```

In the case of the half adder testbench, the following files have to be analyzed by GHDL:

- ANDGATE.vhd
- XORGATE.vhd
- HALF_ADDER.vhd
- HALF_ADDER_TB.vhd

To analyze the files, we can either call GHDL for each file or specify each of the files on the same line:

```
$ ghdl -a ANDGATE.vhd XORGATE.vhd HALF_ADDER.vhd HALF_ADDER_TB.vhd
```

or, we can also do:

```
$ ghdl -a ANDGATE.vhd
$ ghdl -a XORGATE.vhd
$ ghdl -a HALF_ADDER.vhd
$ ghdl -a HALF_ADDER_TB.vhd
```

Whichever method we choose, we should end up with a file, work-obj93.cf, which describes the library work. This work library will contain the designs for each of the files analyzed.

A.2.2. Evaluating Designs with GHDL

The second step when using GHDL is to perform an evaluation of the design analyzed in the first step.

These two steps, analysis and evaluation, might seem a little odd until you realize GHDL is actually creating machine code that runs on your machine!

Similar to the analysis step, each of the files must be evaluated using nearly the same command as before, just with -e instead of -a and using the design name to evaluate:

```
$ ghdl -e ANDGATE
$ ghdl -e XORGATE
$ ghdl -e HALF_ADDER
$ ghdl -e HALF_ADDER_TB
```

The evaluation of the designs (-e) should use the same flags as during the analysis phase (-a).

A.2.3. Simulating a Design with GHDL

Once the design and all underlying designs have been analyzed and evaluated, the final step for GHDL is to run the simulation.

To run a simulation, we call GHDL with the -r flag:

```
$ ghdl -r <design unit> --vcd=<VCD file name> [--stop-time=<time to stop>]
```

For our half adder testbench, we can use the following command:

```
$ ghdl -r half_adder_tb --vcd=half_adder_signals.vcd --stop-time=20ns
```

This command tells GHDL to run a simulation using the half_adder_tb design unit, to generate a VCD output named 'half_adder_signals.vcd', and to run the simulation for 20ns.

After running this command (if you are simulating a large design, it may take some time), we should have the 'half_adder_signals.vcd' file in the directory.

This VCD file contains the signals, their values, and when they changed during the simulation. To view these waveforms, we will use GTKWave.

A.3. Using GTKWave to View the Simulation Results

GTKWave is used here to import the VCD file generated by GHDL during the simulation run and display the signals used in the design.

Figure 4.2 shows the main window of GTKWave.

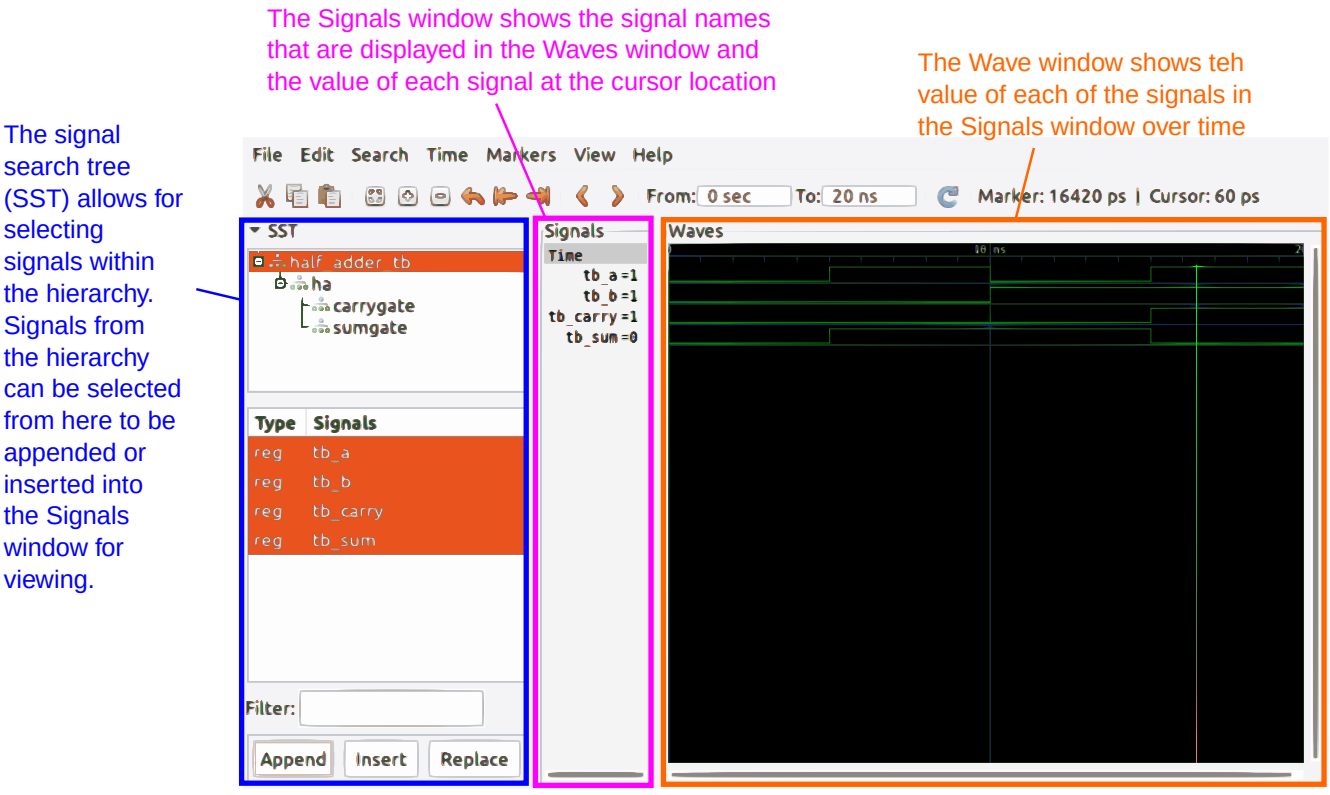


Figure 4.2 The GTKWave main window.

To use GTKWave to view the signals, we need to:

1. Load the VCD file with the signals to view

2. Select the signals to view from the signal search tree (SST).
3. 'Append' or 'Insert' the selected signals from the SST.
4. Verify the signals have been added to the 'Signals' window and are visible in the waveform viewing window.

Once the waveforms have been loaded, the cursor can be used to view the values of each of the signals at that point in time.