**Verilog Lab 2 (ECS1005)**

**Objectives:**

For this lab, please download the Verilog HDL book on the local machines desktop for easy accessibility when needed. The lab builds on the tasks on lab 1, kindly make sure you have them available before you start this lab. The objectives of this lab are as follows.

* Understand the following 2 concepts.
  + Understanding how the instantiation of Gate level primitives and Lower-Level Module differs
  + Difference between Positional Port Mapping and Explicit Port Mapping
* To practice data flow modelling operators using *assign* statement and operators
* To familiarize you with some debugging skills in Replit

**Task 1a: Understanding Module Instantiations**

The term *instantiation* refers to the use or inclusion of a lower-level module within another module. In Verilog, the syntax for instantiating a lower-level module is as follows.

**module\_name <instance\_identifier> (port mapping…);**

The first portion of the instantiation is the module name that is being called. This must match the lower-level module name exactly, including case. The second portion of the instantiation is an instance identifier. An instance identifier is useful when instantiating multiple instances of the same lower-level module. The final portion of the instantiation is the port mapping. You may read more about it in section 5.6.1 Lower-Level Module Instantiation and section 5.6.2 Gate-Level Primitives.

In case of the use of basic gate-level primitives in Verilog, i.e., not(), and(), nand(), or(), nor(), xor(), and xnor(), the **<instance\_identifier>** is not mandatory to use. For modules that we define on our own, we must use the **<instance\_identifier>**

Open your competed project repl for Verilog lab1: Task 2 for a 4-bit adder and fork it locally with label Verilog lab2: Task1. You could instead fork mine given below.

<https://replit.com/@AyeshaKhalid5/ECS1005-Verilog-Lab1-Task2-complete>

In the file *full\_adder.v,* we have used no instance identifiers for the basic gate-level primitives (we used and, or and not gates).

A close up of text

Description automatically generated

However, when instantiating the modules that we defined ourselves (*full\_adder.v*) in the *four\_bit\_adder.v* file, we must give unique instance names, i.e., *full\_adder\_0, full\_adder\_1, full\_adder\_2 and full\_adder\_3,* as this is mandatory.

A close-up of a code

Description automatically generated

Let’s do the same with the full adder as well and give unique instance names to all the gate-level primitives used in *full\_adder.v* (though it is not mandatory but for understanding we are doing that)*.* Let’s call the three instances of not gates as *N0, N1, N2*, the two OR gate instances *O0 and O1* and the 7 AND gates as *A0, A1, …A6*.

A close up of text

Description automatically generated

Recompile the code after these changes and re-run. These harmless changes should not change the behaviour of the repl. Paste the waveform below.

**Task 1b: Understanding Port Mapping**

In the context of Lower-Level Module Instantiation, there are two techniques to connect signals to the ports of the lower-level module, **explicit** and **positional.** You may read more about it in section 5.6.1.1 Explicit Port Mapping and section 5.6.1.2 Positional Port Mapping.

In **explicit port mapping**, the names of the ports of the lower-level sub-module are provided along with the signals they are being connected to. The lower-level port name is preceded with a period (.), while the signal it is being connected is enclosed within parentheses. The port connections can be listed in any order since the details of the connection (i.e., port name to signal name) are explicit. Each connection is separated by a comma.

A diagram of a wiring diagram

Description automatically generated

In **positional port mapping**, the names of the ports of the lower-level modules are not explicitly listed. Instead, the signals to be connected to the lower-level system are listed in the same order in which the ports were defined in the sub-system. Each signal name is separated by a comma. This approach requires less text to describe the connection but can also lead to misconnections due to mistakes in the signal order.

A diagram of a diagram

Description automatically generated

Coming back to our example, we used an explicit port mapping for instantiating the *four\_bit\_adder* in the module *tb\_full\_adder*

A screenshot of a computer code

Description automatically generated

However, we used the positional port mapping in the module *four\_bit\_adder* when we instantiated the 4 instances of *full\_adder.*

A close-up of a number

Description automatically generated

While both techniques are correct, the explicit port mapping is considered superior and a better programming practice as the positional port mapping may lead to misconnections due to mistakes in the signal order. Let’s convert the positional port mapping for the 4 instances of *full\_adder* in the module *four\_bit\_adder* to explicit port mapping. Since the internal port names of the *full\_adder.v* are *A,B,Cin,S,Cout*, we will change the instance *four\_bit\_adder.v* as follows.

A screenshot of a computer code

Description automatically generated

Here, the use of line spaces is completely cosmetic and harmless and done for better readability. Nevertheless, following is also the same.



Convert the port listing for the other three instances of *full\_adder* from position to explicit. Recompile the code after these changes and re-run. These harmless changes should not change the behaviour of the repl. Paste the waveform below.

**Task 2a: Continuous Assignment and Bitwise Logical Operators**

Verilog uses the keyword assign to denote a continuous signal assignment, all these continuous assignments in a design are executed concurrently to model a true depiction of concurrency in hardware. Continuous signal assignment is used to model combinational logic. In the continuous assignment statement, we first use the keyword assign, then we name the destination (LHS) that must be of type wire (not reg). You may read more about it in section 5.5.1 Continuous Assignment. The syntax for the assignment statement is

**assign destination\_wire = source\_expression;**

A close-up of a text

Description automatically generated

The **source\_expression** of the assign statement could be a constant, another variable or an expression comprising of an operator on multiple variables. There are a variety of pre-defined operators in the Verilog. You may read more about it in section 5.4.3 Verilog Operators.

Let’s look first at how the bitwise logical operators work (refer to section 5.4.3.2 Bitwise Logical Operators). We now rewrite the full adder circuit we made in Verilog lab1 task 1 using operator with continuous assignments instead of gate instantiations. Open your competed project repl for Verilog lab1: Task 1 for a full adder and fork it locally with label Verilog lab2: Task2a. You could instead fork mine given below.

<https://replit.com/@AyeshaKhalid5/ECS1005-Verilog-Lab1-Task1-complete>

Replace all not/and/or gates as follows.

A math equations on a white background

Description automatically generated

You may get rid of all the intermediate wires by simply using more than one gate in each assign statement. The assignment for **Cout** will look like this



Carefully write the assignment for Sum **S** as well and remove all the other intermediate wires. Recompile the code after these changes and re-run. These harmless changes should not change the behaviour of the repl. Paste the waveform below.

**Task 2b: Continuous Assignment and Numerical Operators**

Verilog also provides a set of numerical operators (refer to section 5.4.3.9 Numerical Operators). We now rewrite the full adder circuit we made in Verilog lab1 task 1 using Numerical operator with continuous assignments instead of gate instantiations. We must also use the concatenation operator, that is used to concatenate multiple signals together in Verilog (refer to section 5.4.3.7 Concatenation Operator).

. Open your competed project repl for Verilog lab1: Task 1 for a full adder and fork it locally with label Verilog lab2: Task2b. You could instead fork mine given below.

<https://replit.com/@AyeshaKhalid5/ECS1005-Verilog-Lab1-Task1-complete>

Remove the entire body of the code in *full\_adder.v* (except port list, i.e., remove line number 5 to 27) by the following statement



Here we use the numerical operator for addition (+) to add three bits. When we add three bits, the answer could be 2 bits wide. We use concatenation operation ({}) to make two bits signal at the LHS. The outputs of any Verilog module are wires by default (hence Cout and S are wires). This satisfies the condition that the LHS of the assign statement must be a wire.

Recompile the code after these changes and re-run. These changes should not change the behaviour of the repl. Paste the waveform below.

**Task 3a: Write Verilog code for any given SOP**

Let’s see how we can write the Verilog for any SOP/POS expression using the bitwise operator (section 5.4.3.2 Bitwise Logical Operators).

The circuit to be designed consists of four inputs (A,B,C,D) and one output (F). A truth table is given for the output F. Logic minimization is done using K-Map to get a minimal SOP expression. Notice that the input is expected to be BCD, hence input combinations of 1010 to 1111 are not possible and hence they will have output is don’t care (X).

Diagram

Description automatically generated

Perform the following steps.

1. Fork up the following project as lab2 task 3a.

<https://replit.com/@AyeshaKhalid5/ECS1005-Verilog-Lab2-Task3>

1. Write down the SOP expression for the output F of the module *circuit* in the *circuit.v* file.



1. Run the simulation. Match the output waveform with the truth table. Does it match?
2. Paste the waveform below.

**Task 3b: Write Verilog code for any given POS**

Given a POS expression below, lets redo the task 3a.

Text, letter

Description automatically generated

Perform the following steps.

1. Fork up the following project as lab2 task 3b.

<https://replit.com/@AyeshaKhalid5/ECS1005-Verilog-Lab2-Task3>

1. Write down the POS expression for output F in the module *circuit* in the *circuit.v* file. Hint: you will need only the three bitwise logical operators, i.e., &/~/|
2. Run the code and observe the waveform. Using the windows snipping tool paste the waveform below.
3. Fill up the following table by observing its behaviour from the waveform. Match your answers with your neighbours.

A picture containing text, electronics

Description automatically generated

**Task 4 (DIY): Debugging Errors in Replit**

Replit throws some error messages that are not always easy to track down since the error messages are not very informative at times! Let’s put some errors in a working repl and try to analyse the errors. Fork up the compete project (Verilog lab1 task 2) for a 4 bit adder.

<https://replit.com/@AyeshaKhalid5/ECS1005-Verilog-Lab1-Task2-complete>

Perform the following actions and then save the project. Press the Run button and record (and analyse) the error messages you get. After recording the error message that Replit throws, revert the action back to make it error free and proceed with the next error.

|  |  |
| --- | --- |
| **Action** | **Error message/ Effect on simulation by EDA playground** |
| Remove semicolon (;) at the end of line 1 in four\_bit\_adder.v |  |
| Change *S* -> *SUM* (change case) in line 1 in four\_bit\_adder.v |  |
| Give a duplicate name to *full\_adder\_3*, like *full\_adder\_2* in line 13 in four\_bit\_adder.v |  |
| Comment out line 3 in tb\_full\_adder.v (Use // for Single line comment)  // reg [3:0] tb\_A; |  |
| Comment out instantiation of the module *four\_bit\_adder* (line 9-15 in tb\_full\_adder.v) Multi line comment can be done using /\* commented text \*/ |  |