Introduction to Hardware Verification

Final Project Assignment – 2024/02

Connectivity Verification

Introduction

Connectivity is a significant Hardware Design problem because the number of functionalities and components implemented inside a chip continuously increases, but they are still bounded by the available physical IO (input/output) connections to the external world. To get signals out of the chip, complex routing and switching between IP (Intellectual Property) blocks and the outputs are required.

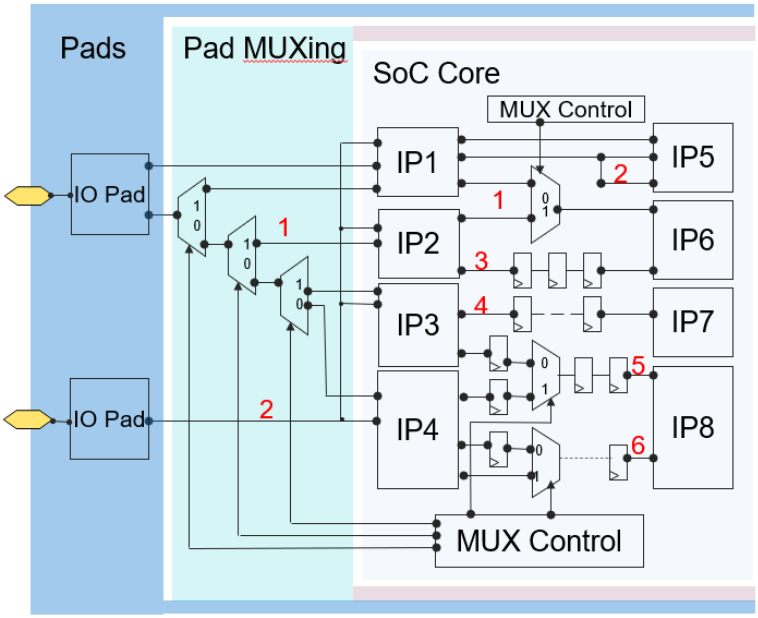

Fig. 1: Example of Pad MUXing network to switch inputs and outputs from a SoC

Connectivity Verification is the problem of checking that the inputs and outputs of an IP block end up at the correct IO pin. There might be thousands of these connections to check against thousands of different pads on the chip interface. Running this verification with Simulation is expensive due to the exponential number of combinations of possible pins and switching patterns. On the other hand, that makes it a suitable problem for Formal Verification methods.

Problem Statement

This project assignment consists of a set of Verilog designs, each with a corresponding Connectivity specification file. The specification files are in the CSV format described below:

1. The first line of each CSV contains the header with 6 column names separated by commas. You can safely ignore the header because all CSVs in the assignment follow exactly the same schema.
2. All the following lines contain 6 columns separated by columns according to the header. Each line represents a connection between two signals in the Verilog design.
3. Each connection is specified by the following columns (in order left to right):

a. **CONNECTION keyword:** You can safely ignore this keyword because this assignment is only concerned with CONNECTION specifications.
b. **Connection name:** This is a string identifying the connection, used for reporting verification results.
c. **Source Instance:** Hierarchical path for an instance in the Verilog design. An empty string in this column refers to the top instance.
d. **Source Signal:** Name of a signal inside the source instance from (c).
e. **Destination Instance:** Hierarchical path for an instance in the Verilog design. An empty string in this column refers to the top instance.
f. **Destination Signal:** Name of a signal inside the destination instance from (e).

```
#,Name,Source Instance,Source Signal,Destination Instance,Destination Signal
CONNECTION,ip8_in_connection,,in1,soc_core.ip8,ip8_in
CONNECTION,ip3_out_connection,soc_core.ip3,ip3_out,pad_muxing,out2
```

Fig. 2: Example of CSV connectivity specification used in this assignment.

The Connectivity Verification goal is to show that, for each connection, there exists a path in the circuit from source signal inside the source instance until the destination signal inside the destination instance. Assume that any physical connection between those signals is a valid connection, ignoring input or output port directions.

You must perform three tasks in this assignment:

- **Formal Connectivity Verification:**
  Use Formal Verification with Jasper to prove that all connections in the CSV specifications are correctly implemented in the Verilog designs.
  For this task, implement the Tcl script **connectivity.tcl** in the project folder. This script receives the CSV specification file in the `$spec` Tcl variable.
  Parse the CSV file and create assertions using Jasper `assert` Tcl command. Each connection in the CSV must be used to create an assertion checking that the signals at both ends of the connection are equal. For instance, the CSV from Fig. 2 would generate the following Tcl assertions:

```
assert "in1 == soc_core.ip8.ip8_in"
assert "soc_core.ip3.ip3_out == pad_muxing.out2"
```

  You only need to create the assertions in Jasper invoking the `assert` Tcl command. The assertions created will be proved automatically by the evaluation script, so there is no need to manually compile the design or run proof commands.
  Hint: There is no need to search for open source Tcl libraries for parsing CSV, you can fully parse the file format in this assignment using Tcl's built-in split command. For instance:

```
set line_contents [split $csv_line ,]
# line_contents is a list containing the CSV row data
```

Hint 2: All the contents in connectivity.tcl only depend on Jasper's `assert` command. If you don't have access to Jasper, you can still implement your code using only built-in Tcl commands and defining a dummy `assert` procedure as exemplified below. Then directly run your script with your OS' `tclsh` interpreter.

```
proc assert {args} {
    puts "DUMMY ASSERT: creating assertion '$args'"
}
```

- **Structural Connectivity Verification:**

  Implement a netlist graph traversal to check that all connections in the CSV specifications are valid in the Verilog designs. For each connection, the source and destination signals must be reachable from each other.

  For this task, implement the Python script **connectivity.py** in the project folder. This script receives the CSV specification file and the JSON netlist file in `csv_spec_file` and `json_netlist_file` variables respectively, both inside the script's `main` function.

  The netlist for each design is provided after compiling the Verilog source with Yosys Open SYnthesis Suite. Yosys is an open-source RTL synthesis compiler and can be installed using your OS's package manager. Yosys compiles each Verilog to a JSON netlist representation that is passed as input for your **connectivity.py** script.

  The project folder also provides an auxiliary **netlist.py** Python module that can be used for implementing graph traversals over the JSON netlist. Refer to the `README.md` file in the project folder for details on how to compile the Verilog files with Yosys and the netlist Python module.

  The expected output format for your script is specified on the next task.

- **Combinational Loop Detection:**

  Your **connectivity.py** implementation must also perform a netlist graph traversal to detect combinational loops starting from the source or destination signals in the CSV.

  When evaluating the results of Formal Connectivity Verification, you may notice that some assertions are not proven, but fail with one of the following messages in Jasper:

```
ERROR (ENL024): Combinational loop found within the
cone of influence for "soc_core.ip8.ip8_in".
WARNING (WCK008): Flop "dff" has a loop through its
clock pin.
```

  This task consists of running a new netlist graph traversal to report the combinational loops found starting from the connection signals.

  Your **connectivity.py** script must print a textual output to stdout with the results from both Structural Netlist Verification and Combinational Loop Detection. Print one connection result per line, reporting the connection name from the CSV and the status "connected", "unconnected" or "combinational_loop", separated by a whitespace. For instance, the example from Fig. 2 would have an output in the format:

```
ip8_in_connection combinational_loop
ip3_out_connection connected
```

Note: Combinational loops are loops involving combinational logic gates (such as AND, OR, NOT, etc.), clock pins of flip-flops and enable pins of latches. Those commonly lead to unstable circuits and are avoided on real RTL designs. Fig. 3 shows an example of two combinational loops. However, *sequential* loops are a common and valid practice on RTL design, used for instance for implementing FSMs. Fig. 4 shows a sequential loop. **The loops reported by your script must be combinational only**.

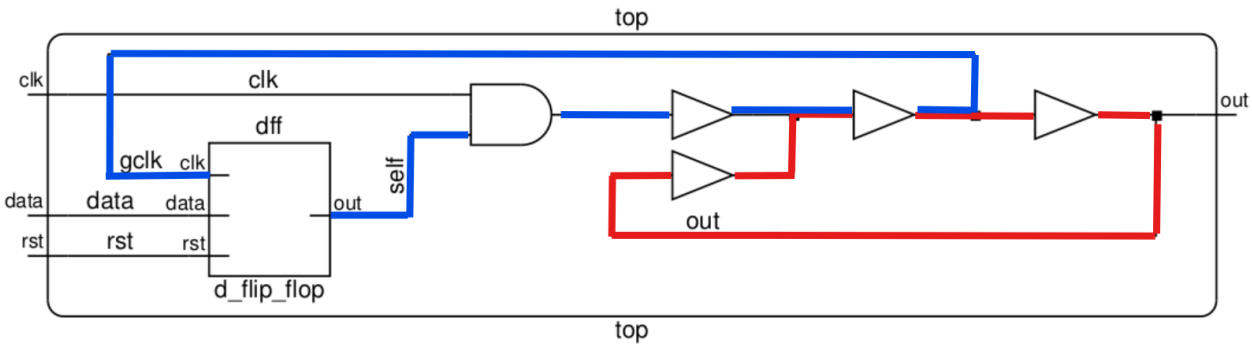

Fig. 3: Example design with two combinational loops: one with the `out` signal and three buffers (in red) and one involving the flip-flop's `gclk` signal, the `self` signal, the AND gate and two buffers (in blue).

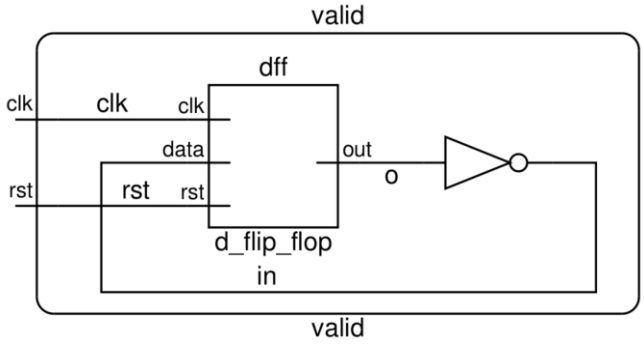

Fig. 4: Example design with a sequential loop. Sequential loops are a common practice on RTL design and must not be reported as problematic loops by your script.

What to hand out

Submit the two files below to the assignment in moodle. The assignment can be done in groups of up to two students. Specify the author(s) name on the comment at the beginning of each script. Only one student per group needs to submit the files.

1. The **connectivity.tcl** file containing the solution of the Formal Connectivity Verification task.
2. The **connectivity.py** file containing the solution of the Structural Connectivity Verification and Combinational Loop Detection tasks.

To test your work, you can run the test cases available in the "tests" folder:

```
cd tests; ./run.sh
```

You can also run your python script on a single test case by passing the JSON and CSV file as command line arguments:

```
python3 connectivity.py example.json example.csv
```