Lab 3: Creating an Interface

In this lab, you will create an interface that allows the counter from Lab 1 to be tested with a testing object, rather than a testing module.

The counter module has been modified to accept an interface on its input port, and now looks like this:

```
 1 module counter (counter_if.cntr_mp i);
 2    always @(posedge i.clk)
 3      if (!i.rst)
 4        i.q <= 0;
 5      else
 6        if (i.ld)
 7          i.q <= i.data_in;
 8        else if (i.inc)
 9          i.q++;
10 endmodule // counter
```

As you can see, the counter now uses an interface to define its data width and its control signals. There are five signals in the interface:

- `q[7:0]` – This 8-bit register is an output from the counter.
- `data_in[7:0]` – This 8-bit bus is the input on a load.
- `clk` – The clock signal that drives the counter.
- `rst` – The reset signal for the counter.
- `ld` – When this signal is high the counter loads.
- `inc` – When this signal is high the counter increments, if the `ld` is not high.

The cntr_mp mod port

The counter_if has a single modport called `cntr_mp`. This mod port contains the signal directions for the counter:

- `q` – Output
- `data_in` – Input
- `rst` – Input
- `clk` – Input
- `ld` – Input

- `inc` – Input

`counter_if` behaviors

The `counter_if` interface needs to implement the following behaviors:

- It must drive the `clk` signal with a clock.
- It must reset the DUT by driving the `rst` signal.

The counter_if.sv file

The `run.do` script in the lab compiles a file called `counter_if.sv`. The file currently looks like this:

```
1 interface counter_if ;
2
3
4 endinterface : counter_if
```

Your job is to fill in the middle of this file.

Running the test

There is a script called `run.do` that will run the simulation. You use the script with the –do option on ModelSim:

```
% vsim –c –do run.do
```

Or in the GUI

```
vsim> do run.do
```

This script assumes that there is a file called `counter_if.sv` that contains the interface.

The Counter Tester

The interface you create must fit into the counter tester. There are two components that must fit with the interface. You do not need to write any of these files, they are for information purposes only.

The Tester Top Level

The top level of the tester looks like this:

```
1  `include "tester.svh"
2
3  module top;
4
5     counter_if ctr_if();
6     counter DUT(ctr_if.cntr_mp);
7     tester my_tester;
8
9     initial begin
10       my_tester=new(ctr_if);
11       fork
12          my_tester.run();
13       join_none
14    end
15
16 endmodule // top
```

The top level instantiates the interface `counter_if` (which you will define in this lab.) Then it puts the interface on the counter's module port, and passes the interface to the tester object.

The Tester Object

The tester object drives the test using the `run()` task. The object assumes that you have defined the signals discussed on the first page. It wants to drive the `ld` and `inc` signals along with `data_in`:

```
1  class tester;
2     virtual interface counter_if i;
3     integer num_loops;
4
5
6     function new (virtual interface counter_if vif, integer n=50);
7       i = vif;
8       num_loops = n;
9     endfunction // new
10
11    task run;
12       repeat (num_loops) begin
13          @(negedge i.clk);
14          {i.ld, i.inc} = $random;
15          i.data_in = $random;
16       end
17       $stop;
18    endtask // run
19 endclass // tester
```

The tester object takes an interface of type `counter_if` as an argument to its constructor, along with the number of loops.

Extra Credit

If you'd like to really make this interface sing, add an always block or two that predict the value of the counter at any point, and check to make sure that the counter is working properly.