Wow, it’s been a while since we last had a good old techie talk about Specman so why not now? Today I’d like to focus on applying reuse to Specman external ports. Very much like little caterpillars, DUTs often have tens or even hundreds of pins which can usually be divided into groups and sub-groups based on their functionality. Certain sub-groups may form a bus - data bus, address bus, and so on - in which case it wouldn’t be as interesting to look at individual wires or pins as at the entire bus as a group.

When you add Specman to the picture, you will probably be tempted to use an external simple port defined as a uint with the size of the bus, be it a byte, word, etc. Actually, this is totally OK. But what happens if you have a group of signals that share the same functionality but have absolutely nothing to do with each other?

For example - your task is to verify the behavior of a group of 7 pins, each of which behaves as an output clock with a preset frequency (set by register). What is the most effective way to monitor and check all that in terms of code reuse?

You might be tempted to group these wires together in the top env. Like this:

{code}extend my_env_u {
 clk_out_bus: out simple_port of uint(bits: 7) is instance;
 };{/code}

But I’m not sure if that would be so elegant. Remember - those pins have nothing to do with each other, what good will a 7-bit uint do?

You could try this:

{code}extend my_env_u {
 clk_out0: out simple_port of bit is instance;
 clk_out1: out simple_port of bit is instance;

clk_out6: out simple_port of bit is instance;
};{/code}

But that smells a bit like a waste of code, doesn't it?

What if we encapsulated the entire code required to verify a single pin (a lightweight eVC for each pin, if you will) and used a *list* of simple_ports in the top env instead? for example:

{code}unit clk_out_pin_evc {
// pointer to verilog port
clk_out_p: out simple_port of bit; // only pointer, not instance

// monitor

// ptr to register

// prot. checker
};{/code}

And then in the env we'll simply instantiate a list of ports, a list of evcs, connect them to each other and voila!

{code}extend my_env_u {
clk_out_port_l[7]: list of out simple_port of bit is instance;
clk_out_env_l[7]: list of clk_out_pin_evc;
keep for each in clk_out_env_l {
it.clk_out_p == me.clk_out_port_l[index];
};
};{/code}

Now that’s more like it!

Oh, one more thing worth mentioning about using lists of ports - when you want to access (drive/monitor) an individual port within a list, remember to place the **$** operator *after* the list index and not
before
. For example:

{code}check that (env.clk_out_port_l[i]$ == 1);{/code}

To sum things up, the point I was trying to make in the simple example above is that smart encapsulation leads to efficient code reuse and eventually to a higher quality of your code (less code, clearer code, more organized structure and easier maintenance). So next time you meet a group of signals, ask yourself if what you’re dealing with is a buss or rather - a centipede