Working with SystemVerilog Structures by MBT

A note on testing

- If you are having problems with SystemVerilog constructs; write a small test that focuses just on the items you want to verify; otherwise other factors in your code may cause the things not to work and you may infer an erroneous conclusion.
- Note this example was all done on Quartus 10.1sp1; if you are different version, you should test it; as the syntax has evolved slightly over time.

Basic Points

- I recommend strongly against using "int" in your structs if you want them to synthesize; I have never used it.
- Same goes with using unpacked structs.
- Project settings should say SystemVerilog 2005
- You should put the struct in a separate file, and then include it with all of the files that use the struct, e.g.

```
`include "my_struct_s.v"
```

Defining the struct

```
typedef struct packed {
logic [17-1:0] instr;
logic [10-1:0] addr;
} instr_packet_s;
```

Example declaration that passes a structure both up and down:

```
module fifo#(parameter I_WIDTH=17, A_WIDTH=10, LG_DEPTH=3)
(
input clk,
input instr_packet_s instr_packet_i, // down
input enque_i,
input deque_i,
input clear_i,
output instr_packet_s instr_packet_o, // up
output empty_o,
output full_o,
output valid_o
);
```

To set fields in a structure-based wire:

```
instr_packet_s ip_in;
assign ip_in.addr = pc_r;
assign ip_in.instr = ram_data;
```

Alternative "setter":

```
assign ip_in = '{addr: pc_r, instr:ram_data};
```

To read fields out of a structure (wire or reg):

```
instr_packet_s ip_out;

assign instruction_addr_o = ip_out.addr;
assign instruction_data_o = ip_out.instr;
```

This alternative getter does not work in Quartus 10.1sp1:

```
// assign '{addr: instruction_addr_o, instr:
instruction_data_o} = ip_out;
```

This is not recommended because it will do the wrong thing if you change the order of fields in instr_packet_s:

```
// assign { instruction_addr_o, instruction_data_o } =
ip_out;
```

Example instantiation of module that passes a structure both up and down:

```
fifo fetch_fifo
(
.clk(clk)
,.instr_packet_i(ip_in)
,.deque_i(dequeue_i)
,.clear_i(fifo_clear)
,.enque_i(fifo_enqueue)
,.instr_packet_o(ip_out)
,.empty_o(fifo_empty)
,.full_o(fifo_full)
,.valid_o(fifo_valid)
);
```

Using Structs to Pass Data Between Synthesized and Unsynthesized Modules

(See this directory for code samples.)

It appears that there is an incompatiblity between quartus and modelsim.
When synthesizing a top-level module that takes in system verilog structus, quartus expands the structure and increases the number of parameters to the module; expanding out the struct. However, modelsim does not do the same expansion, so the testbench which uses a struct coming out of your synthesized module will report an incorrect number of parameters:

```
# ** Warning: (vsim-3017) Z:/Documents/UCSD/CSE
```

```
141L/Processor/
test.tfw(44): [TFMPC] - Too few port connections. Expected
18, found 11.
```

To work around it, you want to convert to flatten structures that go across module boundaries that are synthesized on one side, and non-synthesized on the other (typically, this will only happen for your core.v module)

To flatten a struct (note use of `$bits` macro which returns the size of a packed struct, in bits)

```
// original structure
my_struct_s my_struct = ...
// flattened structure
wire [$bits(my_struct_s)-1:0] my_struct_flat;
assign my_struct_flat = my_struct; // convert struct to
flattened struct
```

To unflatten a structure:

```
wire [$bits(my_struct_s)-1:0] my_struct_flat = ... ;
my_struct_s my_struct;
assign my_struct = my_struct_flat; // converted flattened
struct to real struct
```

So for instance, let's say we had a module that takes two structs, one that goes in and the other that goes out. Here we show how to flatten the structs across the boundary between the testbench and the core (you do not need to flatten structs that are passed between modules that are enclosed inside a synthesized module

BEFORE:

```
// ************************************************
// tb: nonsynthesizable; in tb.v
// tb is our testbench
`include "my_struct_s.v"
module tb ();

my_struct_s in, out;

.. code here ..

core mycore (.my_struct_i(in), .my_struct_i(out));

endmodule

// ************************************************
// core: synthesizable; in core.v
// core is the module we are actually creating
`include "my_struct_s.v"
module core
( input my_struct_s my_struct_i
,output my_struct_s my_struct_o
);
.. code here ..
```

```
endmodule
```

AFTER:

```
// *************************************************
// tb: nonsynthesizable; in tb.v
// tb is our testbench
`include "my_struct_s.v"
module tb ();

my_struct_s in, out;
logic [$bits(my_struct_s)-1:0] in_flat, out_flat;
assign in_flat = in;
assign out = out_flat;
..
core mycore (.my_struct_flat_i(in_flat),
.my_struct_flat_o(out_flat));

endmodule

// *************************************************
// core: synthesizable; in core.v
// core is the module we are actually creating
`include "my_struct_s.v"
module core
( input [$bits(my_struct_s)-1:0] my_struct_flat_i //
flattened struct params of synth module
,output [$bits(my_struct_s)-1:0] my_struct_flat_o //
flattened struct params of synth module
);

my_struct_s my_struct_i, my_struct_o; // the actual structs

assign my_struct_i = my_struct_flat_i; // unflattened in
param
assign my_struct_flat_o = my_struct_o; // unflattened out
param

.. code here ..

endmodule
```