A brief guide to understand the Specman_calc1

Before we can use Specman to simulate VHDL DUT, we need to create an interface file (in this case tb.vhdl) to contain all DUT top level signals (for calculator1 all demo-top signals, for calculator2 all calc2_top signals). Specman drive input stimuli through this file to the DUT. Let's look at the Makefile. statement:

specman -c "load ../e/calc_top.e; write stub -qvh" creates a specman stub file called specman_qvh.vhd, this file works with tb.vhdl together to establish communication when simulation starts. Then all vcom compile vhdl into work directory. Statement:

specview vsim -keepstdout tb -do sn.do invokes both Specman and ModelSim GUI simultaneously an then execute the commands contained in sn.do file.

```
DUT=../vhdl
TB=../e
        rm -rf work specman_qvh.vhd *.elog *.err *.wlf transcript *.ecov
        vlib work
        vcom ${DUT}/alu_input_stage.entity.vhdl
        vcom ${DUT}/alu_input_stage.dataflow.vhdl
        vcom ${DUT}/alu_output_stage.entity.vhdl
        vcom ${DUT}/alu_output_stage.dataflow.vhdl
        vcom ${DUT}/exdbin_mac.entity.vhdl
        vcom ${DUT}/exdbin_mac.custom.vhdl
        vcom ${DUT}/holdreg.entity.vhdl
        vcom ${DUT}/holdreg.dataflow.vhdl
        vcom ${DUT}/mux_out.entity.vhdl
        vcom ${DUT}/mux_out.dataflow.vhdl
        vcom ${DUT}/priority.entity.vhdl
        vcom ${DUT}/priority.dataflow.vhdl
        vcom ${DUT}/shifter.entity.vhdl
        vcom ${DUT}/shifter.dataflow.vhdl
```

```
    # create specman stub file "specman_qvh.vhd"
    specman -c "load ../e/calc_top.e; write stub -qvh"
    vcom  specman_qvh.vhd
    vcom ${DUT}/demo_top.entity.vhdl
    vcom ${DUT}/demo_top.schematic.vhdl
    vcom ${TB}/tb.vhdl
    #specsim vsim -keepstdout demo_top    # mti gui only, sn cmd @ shell
    specview vsim -keepstdout tb -do sn.do  # mti and sn gui
```

E file: cmd_s.e defines the inputs body and the motheds to drive inputs to the DUT. All e program exacutabile statements are between the <' and '>. Anything outside are only comment. Internal comments must start with //. This module defines the basic instruction structure, it includes only the bare essentials for the test bench to function. More under cmd_ext.e

```
<'
   // use code delimiter
   // this is code
   // Define the operations
   type cmd_t: [NOP, ADD, SUB, ILG1, ILG2, SHL, SHR] (bits:4);
   struct cmd_s {  // use "_s suffix to inidcate struct
            // this will be useful later when you
            // need to find it again.
   cmd:    cmd_t;             // The command type
   keep cmd != NOP;           // generated commands must not be NOPs
   // Since NOPs are not acknowledged, inserting NOP in a command
   // stream will cause an error in the driver logic. This is a way
   // to prevent that.
   op1:    uint (bits:32);     // Operand1
   op2:    uint (bits:32);     // Operand2
   // The following is to make the shift operand make sense (this
   // is just a guess regarding the meaning of shift in this design)
   // This is a "soft" constraint because in a specific case we may
   // choose to set the value to a higher number (e.g. check for
   // overflow
   keep cmd in [SHL, SHR] => soft (op2 < 32);
   // Below are features needed for driving and checking
   !output:     uint (bits:32);  // will be set by the DUT
   delay:      uint;           // sets delay between commands
   keep soft delay in [0..100];
   // by default delay is in the range above. it is "soft" so that
   // one may overlay a "hard" constraint to a contradicting value
   // (e.g. 12)

   drive1(number :index) is {
      // Drive the first cycle into DUT
      // "number" is the Calculator port number
      // The following two actions use the "number" field
      // to access the proper signal for this instance
      // of the bfm. This is known as a "computed name".
      // See the specman on-line documentation for more details.
      'req(number)_cmd_in'  = cmd;     // Drive first cycle
      'req(number)_data_in' = op1;
   };

   drive2(number:index) is {
      // Drive the second cycle into the DUT
      'req(number)_cmd_in'  = 0;     // Drive second cycle
      'req(number)_data_in' = op2;
   };
```

```
verify(number :index) is {
    // this method checks to see that the data returned is
    // consistent with the command. A DUT error is issued
    // otherwise
    // First grab the data and response from the DUT
    output = 'out_data(number)';
    var resp :uint (bits:2) = 'out_resp(number)';

    // Next report any response errors
    check that resp == 1 else
    dut_error("Channel ",number," reported error ",resp);

    // Finally, if data is valid - check the result
    // Note that the reference computation below is a speculation
    // that clearly doesn't match the implementation ...
    if resp == 1 then {
        //outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
        case cmd {
            ADD : {
               if ( (op1 + op2) > 8'hffffffff )
               {
                     outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
                     dut_error("Channel ",number," error: can not tell overflow");
               }
               else
               {
                     if( output != op1 + op2 )
                     {
                                 outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
                                 dut_error("Channel ",number," error during ADD");
                     }
```

```
   }
};
SUB : {
     if (op2 > op1)
    {
         outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
         dut_error("Channel ",number," error: can not tell underflow");
    }
    else
    {
          if( output != op1 - op2 )
          {
                    outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
                    dut_error("Channel ",number," error during SUB");
         };
     };
};
ILG1 : {
    outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
    dut_error("Channel ",number," error: can not tell invalid command");
};
ILG2 : {
    outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
    dut_error("Channel ",number," error: can not tell invalid command");
};
SHL : {
    if (output != (op1 << op2 ))
    {
          outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
          dut_error("Channel ",number," error during SHL");
    }
```

```
                };
                SHR : {
                    if(output != (op1 >> op2))
                    {
                         outf("%10d %s %10d = %10d\n", op1, cmd, op2, output);
                         dut_error("Channel ",number," error during SHR");
                    };
                };
                default : { // Nothing
                };
            };
        };
    };
};
'>
```

The marker above is end of code - this line is a comment

For the calc2 we need to add one more signal: tag in the command body and use drive1() and drive2() two methods to drive this signal to DUT with the command and two operands. Check the calculator2 specification for the input formats. The results verification is performed by the method verify(). For the calculator2 the logic and timing will become much more complicated. So students need to take care of drive1(), drive2() and verify() three methods, you don't need to worry about other parts, we will make them work properly.