Royal Institute of Technology (KTH)
School of Electrical Engineering and
Computer Science (EECS)
Department of Electronics (ELE)

Version 1.7
Last revision 2020-10-07

SystemVerilog - Laboration 2

IL2203 Digital Design and Validation using HDLs

Thorough testing of the Microcontroller CPU

Student Name:…………………………………………………………..

Personal Number:…………………………………………………….

Date of Approval:…………………………………………………….

Assistant:……………………………………………………………….

SystemVerilog - Laboration 2

Purpose

The purpose of this homework is to study how an advanced testbench using OOP and scoreboards can be written in SystemVerilog and test it on a familiar design using a mixed-langauge simulation. The testbench should thoroughly test the CPU developed during the VHDL labs. The specification of the design is contained in VHDL Lab 3. Remember also that the controller lab contained two specification bugs (one for the bypass path, and one for the flags). You should treat it as if the specification was fixed before you start testing, i.e., add your Lab 3 bug fixes to the specification.

A fully automated testbench with a scoreboard is shown in Figure 1. The testbench should check that all implemented instructions of the processor controller are correct, and perform operational tests using different usage scenarios. The testbench should contain five layers, the Test-layer, the Scenario-layer, the Functional layer, the Command layer and the Signal layer. Each box (except the DUT) should be implemented using a class. However, these classes can be rather complicated, and difficult to get to work as a Validation and SystemVerilog novice, so in the lab we will implement these layers a bit differently.

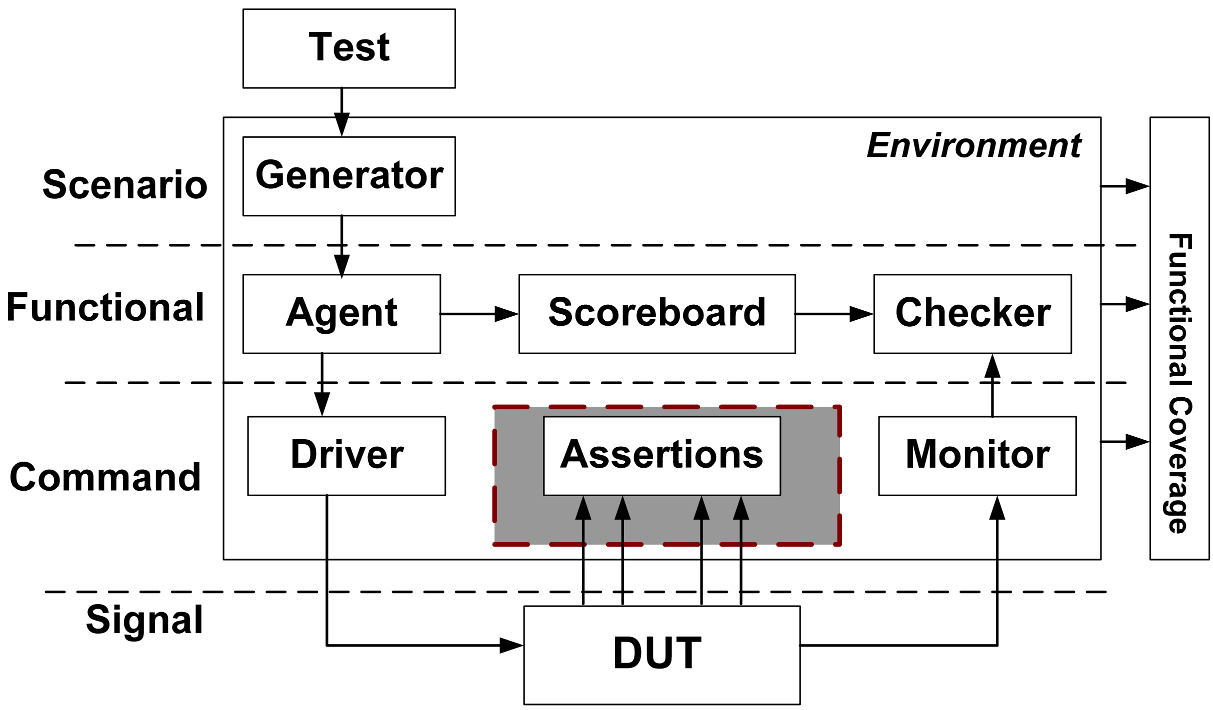

Figure 1. The test environment.

The DUT in our case will be the CPU from the VHDL labs. The functional and the command layer will be the memory component that is connected to the DUT. The generator will be an instruction class, orchestrated from inside the memory component's initial procedure.

Tasks

You should debug the CPU you developed in the VHDL labs, by using a testbench that generates random instructions. The Generator generates random instructions for the addresses 0-255, and ends with a BRA instruction to itself to stall the testing.

Classes & Randomization

Complete the classes outlined in Appendix B. You should enable, disable and/or write more randomization constraints as your test progresses. The goal is to validate that all instructions in your CPU has been implemented correctly. If you want to add directed tests, you can do that in the *post_randomize* –method inside the Instruction class, by overwriting some of the generated ones.

Constraints to add

- Make sure that the target address of your branch instruction ends up in the program range.

Assertions to add

- In the testbench, add one assertion for each instruction that tests that it is functioning (for instance that the bypass is set properly). One assertion is given for the ST-command as an example. Adapt the property so that it matches your own ST-instruction.
- Load and Store instructions shall not write in the address range where code is store. Data should be stored at the address that follows the code. Throw an assertion if ST or LD-instructions are operating outside the designated area.

Functional coverage

The execution of the generated programs should be checked. This can be checked by adding statistics to the memory_data, by changing it to a class, and/or by adding coverage checks in the instruction package. You check the functional coverage by the QuestaSim command Tool->Coverage Report. The functional coverage report also reports the number of assertions that were covered.

Coverage constraints:

- Add coverage constraints to check what instructions combinations have generated.

Some information about the test can be difficult to get hold of using Coverage checks. You can complement the coverage check by adding static statistics code in your memory class.

Memory class:

- Convert the given mem_type struct to a memory class
- Create a method in the class that measure how many of the instructions in the generated programs were actually executed (some are stepped over due to branches),

Measure (either by using coverage constraints or through methods in the memory class)

- how many of each type of Assembly instructions was actually executed.
- what type branch instructions was actually executed, and what the value of the relevant Flag was when the instruction was executed.

Try to achieve an Instruction Coverage >95%, and an assertion coverage of 100%.

Optional:

Rewrite the code and make proper Agent and Driver classes that communicate through Mailboxes.

Deliverables:

1. Testbench code - list
    a. Added Assertions
    b. Added Coverage Directives
    c. Added Constraints for Random Generation of instructions
2. Complete bug report for each instruction
3. Functional coverage of the different tests

Appendix A – Setting Up QuestaSim

You can (1) work from a computer in room 209 or (2) work from your own laptop by connecting remotely to a KTH server.

In case you choose option (2), open a Unix terminal and type:

ssh –Y YourKTHUsername@malavita.sys.ict.kth.se

If you have a Windows computer and want to work from home, it is recommended that you install Linux near Windows. The Ubuntu linux distribution can very easily be installed along with Windows without requiring you to partition your filesystem, as long as you have some free Gigabytes. You can then delete Ubuntu at any time through the Windows control panel. In alternative you can use Xming and Putty to connect remotely to the KTH server, but be aware that this solution is sometimes buggy.

From this time on, everything you type in the terminal happens remotely on the server. The remainder of the instructions are identical for both scenarios (1) and (2), i.e. both if you work in the lab and if you work at home. If you work in the lab, open a terminal using the graphical interface and type all commands there.

First we have to make questasim work. Give the command: *vsim*

• if questasim opens, you are fine.
• if nothing happens or modelsim opens then follow the next instructions:

gedit ~/.bashrc

An editor will pop out, search for all lines that contain the string "modelsim" and comment them out with a hash or delete them. Save the .bashrc file, then close the editor, log out, log in, try again giving the command *vsim*. This time you should get the message: "command not found".

• if you get the message "command not found" or similar, then type on the console the two following commands:

export LM_LICENSE_FILE=1727@lic02.ug.kth.se
export PATH=$PATH:/afs/kth.se/pkg/mentor/questa/20170221/questa_core_prime_10.5c_4/questasim/bin/

Now try giving the command *vsim*, questasim 10.5 should open. Note: every time you open a new terminal, you should retype the two *export* commands.

QuestaSim is an advanced version of ModelSim that allows mixed language simulations.

Appendix B – Sketch of testbench classes

--- Top.sv ---

```
module top;
  bit clk;
  always #5 clk = ~clk;

  logic Din[15:0];
  logic Dout[15:0];
  logic [15:0]Address;
  logic RW;
  logic reset;

  bit [7:0] ST_Count=0;

  cpu #(.N(16),.M(8)) dut
                  (.clk(clk),
        .reset(reset),
        .Din(Din),
        .Dout(Dout),
        .Address(Address),
        .RW(RW));

 memory mem (.clk(clk),
                  .reset(reset),
                  .Din(Dout),
                  .Dout(Din),
                  .Address(Address),
                  .RW(RW));

  initial begin
    reset = 1'b0;
    @(posedge clk);
    reset=1'b1;
    @(posedge clk);
    reset=1'b0;
  end;

  // Instruction properties
/*
  assert property (
    @(posedge clk) ((dut.Instr[15:12]==ST) && (dut.uPC==1)) |-> ##2 $fell(RW)
  );
*/
```

```
always @(posedge clk)
begin
  case (dut.Instr[15:12])
    ST: if (dut.uPC==3) assert (!(RW)) begin
                                      $display("%0t: ST works ok",$time);
                                      ST_Count++;
                       end
                                    else
                                      $display("%0t: ST instruction has an error",$time);
    default:$display("%0t: Not a ST instruction",$time);
  endcase
end

endmodule
```

```
--- memory_class.sv ---
// The mem_type should be converted into a class so that you can collect statistics of the instructions
typedef bit [15:0] uint16;
typedef uint16 mem_type;
```

--- Memory.sv ---

```
`include "mem_class.sv"
`include "SV_RAND_CHECK.sv"
import instr_package::*;
program automatic memory(
  input bit clk,
  input logic reset,
  input logic RW,
  input logic [15:0]Address,
  input logic Din[15:0],
  output logic Dout[15:0]
);

  mem_type data[mem_type],idx=1;
  mem_type a;
  Driver_cbs cbs[$];
  Driver_cbs_cover cb_cover;

  initial begin
    while(1) begin
                @(posedge clk)
                a <= Address;
      assert (!$isunknown(Address))
                 a <= Address;
      else begin
                 $warning("Memory Address is set to unknown");
                 $display("%x",Address);
                end
      if (RW) begin
                 // Driver
                 Dout <= {>>{data[a][15:0]}};
      end
      else begin
                 // Monitor
                 data[a] = {<<{Din}};
      end;
    end;
  end;
```

```
// Load Program
  initial begin
     cb_cover = new();
     cbs.push_back(cb_cover);
     @(posedge reset)
     @(negedge clk)
     for(int i=0;i<256;i++) begin
        instr = new();
        `SV_RAND_CHECK(instr.randomize());
        instr.print_instruction;
        data[i]=instr.Compile(); // 16'h7000; //NOP
        foreach (cbs[i]) begin
             cbs[i].post_tx(instr);
        end
     end
  end;

endprogram
```

--- instruction_package.sv ---

```
package instr_package;
  // Enumerate all the opcodes
  typedef enum  bit [3:0] {ADD,iSUB,iAND,iOR,iXOR,iNOT,MOV,NOP,LD,ST,LDI, NU,
BRZ,BRN,BRO,BRA} opcode_t;

  localparam num_tests = 1000;

  typedef bit [7:0] u_byte_t; // Unsigned byte

  function void print_time_string(input string my_string);
    $display("%0t: %0s", $time, my_string);
  endfunction // print_time_string

class Instruction;
  rand opcode_t opcode;
  rand bit [2:0] target,first,second; // target, R1 and R2 of the opcode
  rand var signed [11:0] branch_offset;
  rand var signed [9:0] immediate; // Data returned by the 3rd memory access of a read
  static bit [7:0] count = 0; // Count to keep track of the memory address
  static string    opcode_str; // To be able to view the instruction in the viewer.
  bit [7:0]   address; // Starting memory address for the Instruction

  constraint no_NU {(opcode != NU);}
  constraint no_LDI_to_PC { !((opcode==LDI) && (target==7)); }
  // Add more code generation constraints here

  function void print_instruction;

    // Print out the assembly instructions
    if (opcode<iNOT)
      $display("%0h: %s R%0h,R%0h,R%0h", address, opcode, target, first ,second);
    else if ((opcode == iNOT) || (opcode == MOV))
      $display("%0h: %s R%0h,R%0h", address, opcode, target, first);
    else if (opcode == LD)
      $display("%0h: %s R%0h,<R%0h>", address, opcode, target, first);
    else if (opcode == ST)
      $display("%0h: %s <R%0h>,R%0h", address, opcode, first, second);
    else if (opcode == LDI)
      $display("%0h: %s R%0h, #%0h", address, opcode, target, immediate);
    else
      $display("%0h: %s #%0h", address, opcode, branch_offset);

  endfunction // print_instruction

  function int Copmile();
   bit [15:0] ret_value;
   ret_value=16'h7000; // NOP
    if (opcode<iNOT) begin
       ret_value[15:12]=opcode;
       ret_value[11:9]=target;
       ret_value[8:6]=first;
```

```
      ret_value[5:3]=second;
    end
    else if ((opcode == iNOT) || (opcode == MOV)) begin
      ret_value[15:12]=opcode;
      ret_value[11:9]=target;
      ret_value[8:6]=first;
    end
    else if (opcode == LD) begin
      ret_value[15:12]=opcode;
      ret_value[11:9]=target;
      ret_value[8:6]=first;
    end
    else if (opcode == ST) begin
      ret_value[15:12]=opcode;
      ret_value[8:6]=first;
      ret_value[5:3]=second;
    end
    else if (opcode == LDI) begin
      ret_value[15:12]=opcode;
      ret_value[11:9]=target;
      ret_value[8:0]=immediate;
    end
    else begin
      ret_value[15:12]=opcode;
      ret_value[11:0]=branch_offset;
    end
    return ret_value;
  endfunction

  function new();
    address = count++;
  endfunction // new

  function void post_randomize;
    if (address==255) begin
      opcode=BRA;
      branch_offset=12'b0;
    end
   // Add directed tests here by overwriting selected target addresses
    opcode_str = opcode.name;
  endfunction

endclass

 // Here follows declaration of Mailboxes, that is used to implement Agents and Drivers

endpackage
```

--- SV_RAND_CHECK.sv ---

```
`define SV_RAND_CHECK(r) \
  do begin \
    if (!(r)) begin \
      $display("%s:%0d: Randomization failed \"%s\"", \
               `__FILE__, `__LINE__, r); \
      $finish; \
    end \
  end while (0)
```