Introduction to SystemC

High Level Design of Systems - Andrés Otero and Rodrigo Marino

Course 2018/2019

1 Introduction

In this lab we will put into practice the main aspects of the SystemC language, by doing the following exercises:

- Basic Combinational Models: Multiplexer
- Basic Sequential Models(I): Counter
- Basic Sequential Models (II): Finite State Machine
- Communication Channels: Custom Stack Channel with a producer/consumer scheme.

In each exercise, the corresponding module will be implemented and simulated. The design and the simulation will be carried out with Vivado HLS. For each exercise, we will implement a custom project in Vivado HLS,including the module and the test bench. The result of the simulation will be analyzed with GTKWAVE. You can download it from: `https://sourceforge.net/projects/gtkwave/`. The executable waveform viewer *gtkwave.exe* can be found in the *bin* folder of the downloaded file.

As a reminder, we include the basic structure of a module in SystemC:

```
#ifndef NAME_H
#define NAME_H
#include <systemc.h>

SC_MODULE(NAME)
{
    //Potential internal signals of the module.
        public:
                //List of input/output ports.
                sc_in< port data type > name;
                sc_in< port data typ > name;

            //Constructor of the Module
```

```
        SC_CTOR(NAME){
      //Method describing the behaviour of the module
                SC_METHOD(Name_Behaviour);
                sensitive << signal1 << signal2 << signal3 <<  ...;
        }
    //Definition of the Methods.
    void Name_Behaviour();
};
#endif
```

The definition of the modules will be generated in a header file (.h), while the implementation of the methods will be provided in the corresponding source file (.cpp).

The main file is the starting point of the C/C++ program. In the case of SystemC, the starting point is the **sc_main()**, which encapsulates the **main()** function. The main instruction in this function is the **sc_start()**, which starts and end the simulation. By convention, the file containing the **sc_main()** instruction is named main.cpp. The structure of this function will be as follows:

```
int sc_main(int argc, char* argv[]) {

    //List of signals to connect the Test Bench and the Module
        sc_signal< data type > name_of_the_signal;
                    .....

        sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", SC_DO_NOTHING);

        //Time resolution of the simulation
        sc_set_time_resolution(1, SC_NS);

        //Test Bench Instance and port mapping
        Mux_tb *Mux_tb_inst;
        Mux_tb_inst = new Mux_tb("Mux_tb");
        Mux_tb_inst->sel(sel_tb);
                    .....

        //Component Instance and port mapping
        Mux *Mux_inst;
        Mux_inst = new Mux("Mux");
        Mux_inst->sel(sel_tb);
                    .....

        //Create a trace file.
        sc_trace_file*  tf = sc_create_vcd_trace_file("trace_NAME);
        sc_trace(tf,name of the signal, "name of the signal");
                    .....
```

```
        //Start Simulation
        sc_start(500, SC_NS);

        cout << " Finish Simulation \n";
        sc_close_vcd_trace_file(tf);

        return(0);

}
```

2 Basic Combinational Models: Multiplexer

In this exercise we will design a basic multiplexer. It will have a selection signal (sel), of the type *unsigned int*, for *boolean* data inputs and a single *boolean* data output.

To do so, we will include the following files in the project:

- *Mux.h*: Includes the definition of the Multiplexer Module, including the input and output ports, the constructor and the method which will describe the functionality. It is important to notice the effect of the sensitivity list in the constructor of the module.S
- *Mux.cpp*: Implements the method which corresponds to the specific functionality of the Multiplexer:

  ```
  void Mux::Mux_Behaviour() {

          switch (sel.read()) {
            case 0:
                    out0.write( in0.read() );
              break;

                ....

            default:
            ....
              break;
          }

  };
  ```

  In this case, the functionality is implemented with a *switch ... case* structure.

- *Mux_tb.h*: Contains the description of the Test Bench Module, where the stimuli to simulate the behaviour of the Multiplexer will be evaluated. Please, notice that in Test Bench modules we typically use the

SC_THREAD template, instead of **SC_METHOD**. Can you remember the reason?

- *Mux_tb.cpp*: Contains the method that describes the generation of the stimuli:

```
#include "Mux_tb.h"

void Mux_tb::Mux_stimuli() {

          while(true) {

                  sel.write(0);
                  in0.write(false);
                  in1.write(false);
                  in2.write(false);
                  in3.write(false);
                  wait(10,SC_NS);

                  sel.write(0);
                  in0.write(true);
                  in1.write(false);
                  in2.write(false);
                  in3.write(false);
                  wait(10,SC_NS);
                        ...
              }
}
```

- *main.cpp*: Main file of the project, where the different modules are instantiated (in this case the component and the test bench), the simulation is started and the log trace file completed.

3 Basic Sequential Models(I): Counter

In this exercise we will implement an up counter including its test bench. The counter will have a *clk*, *count* and *reset* input signals, and will provide the result of the count in the output *q*, which will be of the type int.

The project will be composed of the following files:

- *counter.h*: Includes the definition of the counter module, according to the expected behaviour. In this case we will explore the alternative constructor for the module:

```
SC_MODULE(counter)
{
```

```
  int value;

public:
  sc_in<bool> clk;
  sc_in<bool> count;
  sc_in<bool> reset;
  sc_out<int> q;

  SC_HAS_PROCESS(counter);

  counter(sc_module_name nm): sc_module(nm), value(0) {
    SC_METHOD(do_count);
    sensitive << clk.pos() << reset;
  }

  void do_count();

};
```

- *counter.cpp*: Includes the definition of the process of the counter, describing the functionality of the module.
- *tb_counter.h*: Definition of the module of the counter test bench. In this case, two different processes are included: one for generating the stimuli signals and another one to generate the clock.
- *tb_counter.cpp*:Definition of the functions used inside the test bench.

```
void testbench::clk_gen() {
  while(true) {
    clk.write(true);
    wait(10,SC_NS);
    clk.write(false);
    wait(10,SC_NS);
  }
}

void testbench::stimuli() {
    while(true) {
      reset.write(true);
      count.write(false);
      wait(10,SC_NS);
          ....
    }
  }
```

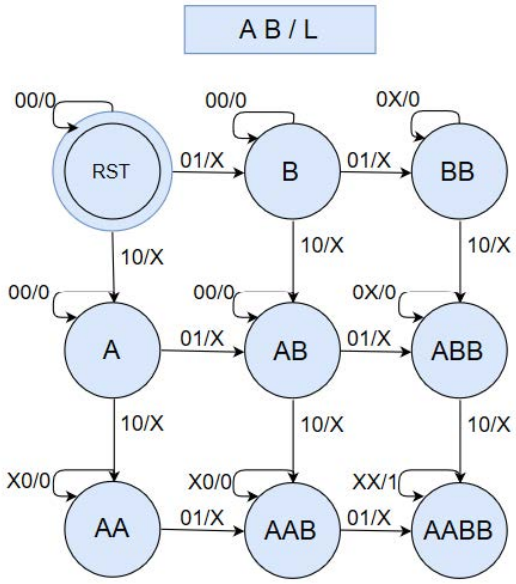
- *top.h*: Definition of the top level class, where the counter and the test bench are mapped together. Differently to the previous exercise, in this case a specific module will be included to map together the testbench and the module under test, so a single module is instantiated in the main file. This way, we practice with structural descriptions:

```
class top: public sc_module {

        public:
                sc_signal<bool> clk_sig, count_sig, reset_sig;
                sc_signal<int> q_sig;

                counter uut; //Counter instance
                testbench tb; //TB instance

                top(sc_module_name nm): sc_module(nm), uut("uut"), tb("tb") {
                        tb.clk(clk_sig);
                        tb.reset(reset_sig);
                        tb.count(count_sig);

                        uut.clk(clk_sig);
                        uut.reset(reset_sig);
                        uut.count(count_sig);
                        uut.q(q_sig);
                }

};
```

- *main.cpp*: Main file where the simulation is created and controlled. Only the class in *top.h* is instantiated.

4 Basic Sequential Models (II): Finite State Machine

In this exercise we will model a Finite Start Machine (FSM), which is widely used when modeling both software and hardware components. The FSM to be modeled correspond to a locker, in which is necessary to activate twice each of the two buttons (A and B), in any order, to open the door. Opening the door is represented by setting the signal L to one. This behaviour is represented by the following diagram:



In this case, the structure of project is the following:

- *fsm.h*: Description of the FSM module, including two processes: one to compute the next state and the outputs, and another one to update effectively the state. The method to compute the next states will be purely combinational, so it will be only sensitive to the inputs. The method to update the state will be sequential, so it will be triggered by clock edges.

```
#define STATE_Reset 0
#define STATE_A 1
#define STATE_AA 2
        ...

SC_MODULE(fsm){

        int CURRENT_STATE;
        int NEXT_STATE;

public:
        sc_in< bool > a;
        sc_in< bool > b;
        sc_in< bool > clk;
        sc_in< bool > reset;
        sc_out< bool > open;

        SC_CTOR(fsm){
```

```
                SC_METHOD(Next_State_Proc);
                sensitive << clk.pos();
                reset_signal_is(reset, true);

                SC_METHOD(Update_State);
                sensitive << a << b << clk.pos();

        }

        void Update_State();
        void Next_State_Proc();

};
```

- *fsm.cpp*: Provide the behaviour of the methods inside the module

```
void fsm::Update_State() {
        if (reset.read() == true)
                CURRENT_STATE = STATE_Reset;
        else
                CURRENT_STATE = NEXT_STATE;
};


void fsm::Next_State_Proc() {

        open.write(0);

        switch (CURRENT_STATE) {

                case STATE_Reset:
                        //RESET State
                        if(a.read() == 1)
                                NEXT_STATE = STATE_A;
                        else if(b.read() ==1 )
                                NEXT_STATE = STATE_B;
                        else
                                NEXT_STATE = STATE_Reset;
                        break;

                case STATE_A:
                        //State A
                        if(a.read() == 1)
                                NEXT_STATE = STATE_AA;
                        else if(b.read()==1)
                                NEXT_STATE = STATE_AB;
```

```
                else
                        NEXT_STATE = STATE_A;
                break;

        case STATE_B:
                //State B
                if(a.read() == 1)
                        NEXT_STATE = STATE_AB;
                else if(b.read()==1)
                        NEXT_STATE = STATE_BB;
                else
                        NEXT_STATE = STATE_B;

                break;

        ...

        default:
                NEXT_STATE = CURRENT_STATE;
        }
};
```

- *fsm_tb.cpp*: Provide the list of stimuli (the sequence of a and b) to test the behaviour of the FSM.
- *fsm_tb.h*: Simple test-bench module.
- *main.cpp*: Main simulation file. In this case, the Test Bench and the System under test modules are instantiated here in the main file.

5 Communication Channels (I): Custom Stack Channel with a producer/consumer scheme.

In this exercise we model two modules, once a data producer and another one, a data consumer, which act respectively as data source and data sinks, to be used to test a communication infrastructure. Both modules will be communicated using a custom communication infrastructure, which must behave as a stack. Let's start first by modeling the channel. It comprises two main files, the interface and the channel itself. The interface contains the list of methods to be used from the modules connected to the channel. Both the read and write interfaces will be described:

```
#include "systemc.h"

class stack_read_if: virtual public sc_interface
{
```

```
        public:
                //read a character
                virtual bool nb_read(char &c) = 0;
};

class stack_write_if: virtual public sc_interface
{

        public:
                //write a character
                virtual bool nb_write(char c) = 0;

                //empty the stack
                virtual void reset() = 0;

};
```

Once the interfaces can be described, we can go on with the definition of the channel, which implements the virtual functions in the interface:

```
class stack: public sc_module, public stack_write_if, public stack_read_if
{

private:
        char data[20];
        int top; //Pointer to the top of stack

public:
        //constructor
        stack(sc_module_name nm) : sc_module(nm)
        {
        }

        //methods in the interface
        bool nb_write(char c){
            ....

        }

        void reset(){
        }

        bool nb_read(char &c){
                ...
        }

        //Report a message when a port is bind.
```

```
        void  register_port(sc_port_base& port_,
                        const char* if_typename_)
        {

                        cout << "binding  " << port_.name() << "to "
                                << "interface: " << if_typename_ << endl;
        }

};
```

Then, we create a simple logic for the consumer and the producer:

```
SC_MODULE(producer) {

public:
        sc_port<stack_write_if> out;
        sc_in<bool> clock;

        void produce_data()
        {
                int i = 0;
                char *String2send = "This is a course on ESL design  ";
                while (true) {
                        wait(); //Wait for a new clock edge

                        if( out->nb_write(String2send[i])){
                                cout << "W " <<String2send[i] << " at " << sc_time_stamp() <
                                i = (i+1) %32;
                        }
                }
        }

        SC_CTOR(producer){
                SC_THREAD(produce_data);
                sensitive << clock.pos();
        }

};

SC_MODULE(consumer) {

public:

        sc_port<stack_read_if> in;
        sc_in<bool> clock;

        void consume_data()
```

```
        {
                char ReadCharFromStack;
                while (true) {
                        wait(); //Wait for a new clock edge

                        if( in->nb_read(ReadCharFromStack) ){
                                cout << "R " << ReadCharFromStack << " at " << sc_time_stamp
                        }
                }
        }

        SC_CTOR(consumer){
                SC_THREAD(consume_data);
                sensitive << clock.pos();
        }

};
```

Finally, you must create a top file including the producer, the consumer and the channel:

```
class top: public sc_module {

public:
        sc_in< bool > clk_prod;
        sc_in< bool > clk_cons;

        //Channel to be used
        stack stack_inst;
        producer prod_inst;
        consumer cons_inst;

        top(sc_module_name nm): sc_module(nm), prod_inst("producer_inst"), cons_inst("consum
        {
                prod_inst.out(stack_inst);
                prod_inst.clock(clk_prod);

                cons_inst.in(stack_inst);
                cons_inst.clock(clk_cons);
        }


};
```

Finally, a test bench file has to be generated including two different clock signals, off different frequencies (50 and 100 NS, one for each block).

As a final exercise, you can figure out how to use of the SC_FIFO channel included in SystemC, and use it in this scheme.

6 Communication Channels (II): Producer/consumer scheme using fifo adapters.

In this exercise we model four modules: producer, consumer, write adapter and read adapter. Both adapters are connected to the FIFO as the producer/consumer interfaces. The objective is to model the handshake protocol and adapt it to the FIFO communication. Let's start first by modeling modifying the producer. It comprises one header file. We reuse part of the methods developed in the previous exercise:

```
// producer.h
#ifndef PRODUCER_H
#define PRODUCER_H

#include "systemc.h"

SC_MODULE(producer){
public:

        sc_out<char> data_out;
        sc_in<bool> clock;
        sc_out<bool> valid_w;
        sc_in<bool> ready_w;


        void produce_data(){
                int i = 0,k=0;
                char String2send[500] ="This is a course of HLS-TLM";

                while(true){
                        wait();

                        if(ready_w==true){
                                cout << "Data sent from the producer " << data_out;
                                cout << " at " << sc_time_stamp() << endl;
                                ...
                        }else{
                                ...

                        }
                }
        }
```

```
        SC_CTOR(producer){
                SC_THREAD(produce_data);
                sensitive << clock.pos();
        }

};

#endif
```

In a different file, we will model the write adapter, attending the handshake behaviour and FIFO constraints.

```
#include "systemc.h"


SC_MODULE(consumer){
public:

        sc_in<char> data_in;
        sc_in<bool> clock;
        sc_in<bool> valid_r;
        sc_out<bool> ready_r;

        char ReadCharFromStack;
        void consume_data(){

                ready_r->write(false);
                while(true){
                        wait();
                        if(valid_r.read() == true){
                                if(ready_r==true){
                                    ...
                                        cout << "Data received by the consumer ";
                                        cout << ReadCharFromStack;
                                        cout << " at " << sc_time_stamp() << endl;
                                }else{

                                        ...
                                }

                        }

                }

        }
```

```
        SC_CTOR(consumer){
                SC_THREAD(consume_data);
                sensitive << clock.pos();
        }
};
```

This procedure is also repited in the consumer case:

```
#ifndef CONSUMER_H
#define CONSUMER_H


#include "systemc.h"


SC_MODULE(consumer){
public:

        sc_in<char> data_in;
        sc_in<bool> clock;
        sc_in<bool> valid_r;
        sc_out<bool> ready_r;

        char ReadCharFromStack;
        void consume_data(){

                ready_r->write(false);
                while(true){
                        wait();

                        if(valid_r.read() == true){

                                if(ready_r==true){

                                        ready_r->write(false);
                                        ReadCharFromStack=data_in;
                                        cout << "Data received by the consumer ";
                                        cout<< ReadCharFromStack ;
                                        cout<< " at " << sc_time_stamp() << endl;
                                }else{

                                        ready_r->write(true);
                                }

                        }
```

```
                }

        }

        SC_CTOR(consumer){
                SC_THREAD(consume_data);
                sensitive << clock.pos();
        }
};



#endif
```

And, in the read adapter:

```
SC_MODULE(adapter_read){
public:

        sc_out<char> data_out;
        sc_port<sc_fifo_in_if<char> > data_fifo_in;
        sc_in<bool> clock;
        sc_out<bool> valid_r;
        sc_in<bool> ready_r;
        char ReadCharFIFO;


        void read_action(){

                bool data_valid=false,data_sent=true;


                while(true){
                        wait();
                        if((data_fifo_in->nb_read(ReadCharFIFO))==true && data_sent==true){

                                //ok data_valid==true
                                data_valid=true;
                                data_sent=false;
                        }else{
                                data_valid=false;
                        }


                        if(data_valid==true){
```

```
                        valid_r->write(true);
                        data_out->write(ReadCharFIFO);
                }


                if(ready_r==true){
                        cout << "Data sent from the reader " << data_out;
                        cout << " at " << sc_time_stamp() << endl;
                        valid_r.write(false);
                        data_sent=true;
                }
        }
}

SC_CTOR(adapter_read){
        SC_THREAD(read_action);
        sensitive << clock.pos();
}

};
```

To join all the modules, we have to create a top module:

```
#ifndef TOP_H
#define TOP_H

#include "systemc.h"
#include "producer.h"
#include "consumer.h"
#include "hierarchical.h"

class top: public sc_module{
public:
    //port declaration
        ...

    //signal declaration
        ...

    //module instatiation
        ...


        top(sc_module_name nm, int size):sc_module(nm), fifo_inst("fifo_inst",size), prod_in

                //interconnections prod
```

```
                ...

                //interconnections adapter_write
                ...

                //interconnections adapter_read
                ...

                //interconnections cons
                ...

                //interconnections clock
                ...
        }

};

#endif
```

For testing the architecture, we must create a testbench. Thus, we have to create the following code in the a header file:

```
#include "systemc.h"
#include "top.h"

int sc_main(int argc,char* argv[]){

        int i=0;

        sc_set_time_resolution(1,SC_NS);

    //signal declaration
    ...

    //top module instantiation
    ...


        //Start Simulation
        sc_start(100000, SC_NS);



        return 0;

}
```