Exercise 2: SystemC and Virtual Prototyping

SystemC Modules

Lukas Steiner

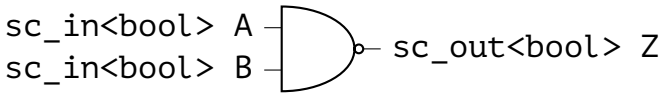
WS 2023/2024

The source code to start this execise is available here:
https://github.com/TUK-SCVP/SCVP.Exercise2

Task 1

NAND Gate

In this task you will write your first SystemC module. The module should have the name `nand` and should implement the functionality of a NAND gate, shown below.



As input and output signals `sc_in` and `sc_out` should be used with the template type `bool`. The input and output signals should be initialized with a proper name in the `SC_CTOR`. The module should have one `SC_METHOD` called `do_nand()`, which is sensitive to the input signals A and B. The module should be implemented in the file `nand.h`.

In order to test your module make sure `nand_main.cpp` is included in the CMake file `CMakeLists.txt`. After successfully testing your NAND gate change the project file to include `exor_main.cpp`. This is necessary to test your next SystemC module in which you will implement an XOR gate using four instances of your NAND.

nand.h

```
#ifndef NAND_H
#define NAND_H

#include <systemc.h>

SC_MODULE(nand)
{
    public:
    sc_in<bool> A;
    sc_in<bool> B;
    sc_out<bool> Z;

    SC_CTOR(nand): A("A"), B("B"), Z("Z")
    {
        SC_METHOD(do_nand);
        sensitive << A << B;
    }

    void do_nand()
    {
        Z.write( !(A.read() && B.read()) );
    }
};

#endif

```

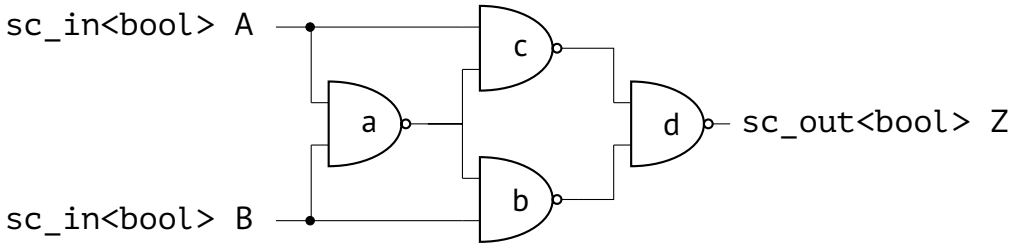
nand_main.cpp

```cpp
#include <systemc.h>
#include "nand.h"

SC_MODULE(toplevel)
{
private:
    nand n1;
    unsigned int cnt;
    sc_signal<bool> A;
    sc_signal<bool> B;
    sc_signal<bool> Z;

public:
    SC_CTOR(toplevel) : n1("n1"), cnt(0)
    {
        n1.A.bind(A);
        n1.B.bind(B);
        n1.Z.bind(Z);

        SC_METHOD(process);
        sensitive << A << B << Z;
    }
private:

    void process()
    {
        cnt++;

        A.write(false);
        B.write(false);

        std::cout << "SC_METHOD process() trigger counter " << cnt << " simulation time " << sc_time_stamp().to_default_time_units() << " ps Δ cycle " << sc_delta_count() << ":\tA " << (A.read() ? "'1'" : "'0'") << " B " << (B.read() ? "'1'" : "'0'") << " Z " << (Z.read() ? "'1'" : "'0'") << std::endl;

        A.write(true);
        B.write(true);

        std::cout << "SC_METHOD process() trigger counter " << cnt << " simulation time " << sc_time_stamp().to_default_time_units() << " ps Δ cycle " << sc_delta_count() << ":\tA " << (A.read() ? "'1'" : "'0'") << " B " << (B.read() ? "'1'" : "'0'") << " Z " << (Z.read() ? "'1'" : "'0'") << std::endl;

        std::cout << std::endl;
    }
};

int sc_main(int, char**)
{

    toplevel top("toplevel");

    sc_start();

    return 0;
}
```

Task 2

SystemC Module Hierarchy – XOR

In this task you will write a SystemC module that is composed of other SystemC modules. The module should have the name `exor` and should implement the functionality of an XOR using only NAND gates, as shown below.



In order to connect the `nand` modules, you need additional helping signals which you will implement by using the `sc_signal<bool>` datatype. The signals should have the names `h1`, `h2` and `h3`. All input, output and helping signals as well as the `nand` modules should be initialized properly with a name from the `SC_CTOR(exor)`.

If you are done with the implementation, have a look at the `SC_MODULE`s `stim` and `mon` and the `sc_main()` function and try to understand what these components are doing.

Why is the `stim` class using an `SC_THREAD` for its process and not an `SC_METHOD`?

Now lets compile and run your program. If you did everything correctly, you should see the following output:

```
time     A    B    F
0 s      0    0    1
0 s      0    0    0
10 ns    0    1    0
10 ns    0    1    1
25 ns    1    0    1
35 ns    1    1    1
35 ns    1    1    0
45 ns    0    0    0
```

Why are you seeing several outputs for each time, sometimes even with wrong results?

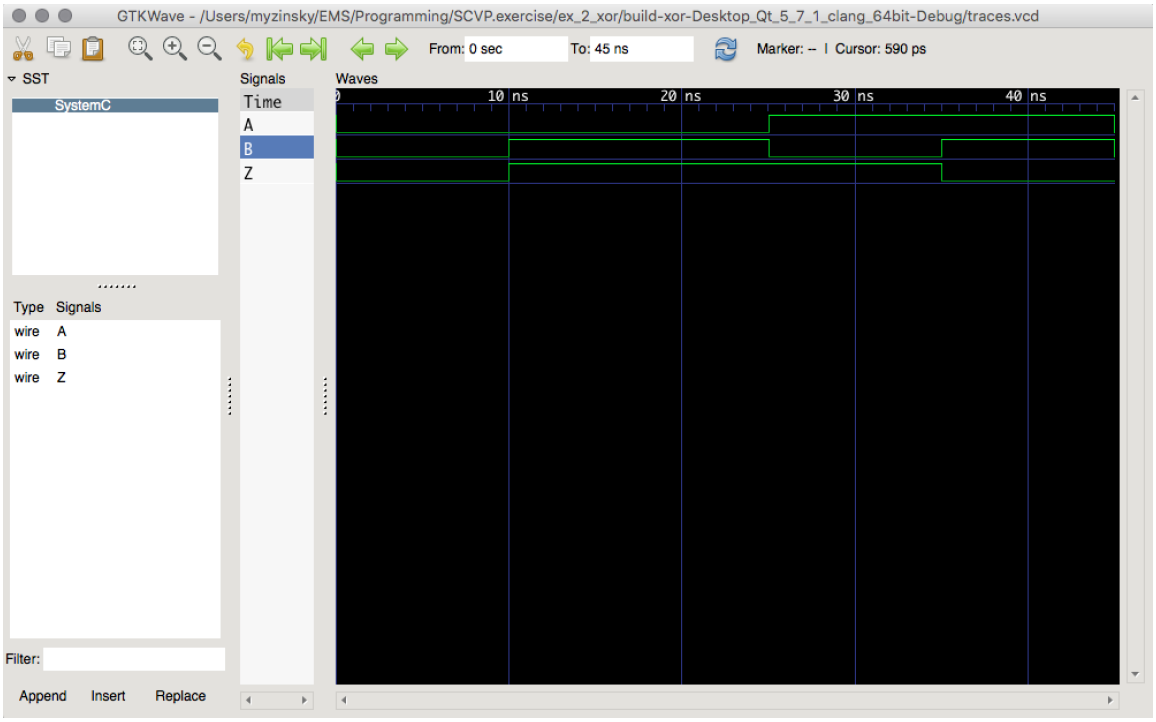
Task 3

Debugging Tracing

Additionally to the mon component, use the waveform feature of SystemC in the `sc_main` method before `sc_start`. Then use the tool GTKWave in order to have a look on the waveform. The file is located inside the build folder.

Information on this feature can be found here: https://www.doulos.com/knowhow/systemc/tutorial/debugging/

In GTKWave you have to drop the signals to the waveform and you have to zoom out a little in order to see the final result:



Task 4

Clocked Processes

Add an `sc_clock` to the `sc_main` function. Remove the `wait(XX, SC_NS)` statements in the `stim` module and replace them with empty `wait()` statements. Add an `sc_in<bool> Clk` to the `stim` and the mon components and make the processes of both modules only sensitive to the positive edge of the clock. Then connect the `sc_clock` in the `sc_main` to the modules. What you will observe at the terminal output? Now add the clock signal to the waveform and analyze it with GTKWave.

exor.h

```
#ifndef EXOR_H
#define EXOR_H

#include <systemc.h>
#include "nand.h"

SC_MODULE(exor)
{
    sc_in<bool> A;
    sc_in<bool> B;
    sc_out<bool> Z;

    nand aa;
    nand bb;
    nand cc;
    nand dd;

    //^Auxiliar signals to connect between the ANDs
    sc_signal<bool> h1, h2, h3;

    SC_CTOR(exor): A("A"), B("B"), Z("Z"), h1("h1"), h2("h2"), h3("h3"), aa("aa"), bb("bb"), cc("cc"), dd("dd")
    {
        //^Connect ports
        aa.A.bind(A);
        aa.B.bind(B);
        aa.Z.bind(h1);

        //^Is the same as using .bind
        bb.A(h1);
        bb.B(B);
        bb.Z(h2);

        cc.A(A);
        cc.B(h1);
        cc.Z(h3);

        dd.A(h3);
        dd.B(h2);
        dd.Z(Z);
    }
};

#endif // EXOR_H

```

exor_main.cpp

```cpp
#include <systemc.h>

#include "stim.h"
#include "exor.h"
#include "mon.h"
#include "nand.h"

int sc_main(int, char**)
{
    sc_signal<bool> sigA, sigB, sigZ;  //^Auxiliary signals

    sc_clock clock("Clk", 10, SC_NS, 0.5);  //---Task4

    stim Stim1("Stimulus");   //^Changes signals A and B each clock cycle
    Stim1.A(sigA);
    Stim1.B(sigB);
    Stim1.Clk(clock);  //--Task4

    exor DUT("exor");
    DUT.A(sigA);
    DUT.B(sigB);
    DUT.Z(sigZ);

    Monitor mon("Monitor");
    mon.A(sigA);
    mon.B(sigB);
    mon.Z(sigZ);
    mon.Clk(clock);

    sc_trace_file *wf = sc_create_vcd_trace_file("traceClk");
    sc_trace(wf, sigA, "A");
    sc_trace(wf, sigB, "B");
    sc_trace(wf, sigZ, "Z");
    sc_trace(wf, clock, "Clk");

    sc_start();  // run forever

    sc_close_vcd_trace_file(wf);
    return 0;
}

```

stim.h

```
#ifndef STIMULUS_H
#define STIMULUS_H

#include <systemc.h>

SC_MODULE(stim)
{
public:
    sc_in<bool> Clk; //--------Task4
    sc_out<bool> A, B;

    SC_CTOR(stim)
    {
        SC_THREAD(StimGen);
        sensitive << Clk;  //---Task 4
    }

private:
    void StimGen()
    {
        //wait(SC_ZERO_TIME);
        wait();
        A.write(false);
        B.write(false);
        //wait(10, SC_NS);
        wait();
        A.write(false);
        B.write(true);
        //wait(15, SC_NS);
        wait();
        A.write(true);
        B.write(false);
        //wait(10, SC_NS);
        wait();
        A.write(true);
        B.write(true);
        //wait(10, SC_NS);
        wait();
        A.write(false);
        B.write(false);
        //wait(10, SC_NS);
        wait();
        sc_stop();
    }
};

#endif

```

mon.h

```
#ifndef MONITOR_H
#define MONITOR_H

#include <iostream>
#include <systemc.h>

SC_MODULE(Monitor)
{
public:
    sc_in<bool> A, B, Z;
    sc_in<bool> Clk;   //------Task 4

    SC_CTOR(Monitor)
    {
        std::cout << std::endl <<  "time\tA\tB\tF" << std::endl;
        SC_METHOD(monitor);
        //sensitive << A << B << Z;
        sensitive << Clk;
        dont_initialize();
    }

private:
    void monitor()
    {
        std::cout << sc_time_stamp()  << "\t" << A << "\t" << B << "\t" << Z << std::endl;
    }
};

#endif

```