HW6 : SystemC AMS Modeling - TDF

Student Name: Amirhosein Yavarikhoo

Student ID: 810199514

***Abstract*— SystemC AMS offers us a solution to connect digital circuits to analog circuits. In this experiment, we use a systemC module to simply create a transmitter and reciever.**

***Keywords*— System C modelling, analog circuits, digital to analog converters, bandpass filter**

I. Introduction

In this assignment, we create a simple way to broadcast and receive a data using a carrier and AM modulator.

II. Message and Carrier

we need 2 sine waves. Amplitude of the message signal is 1 and the carrier is 4.

```
SCA_TDF_MODULE(message_src)
{
    sca_tdf::sca_out<double> out; // output port
    sca_tdf::sca_de::sca_out<double> Mt;
    message_src(sc_core::sc_module_name nm, double ampl_ = 1, double freq_ = 1.0e4,
        sca_core::sca_time Tm_ = sca_core::sca_time(0.1, sc_core::SC_MS))
        : out("out"), ampl(ampl_), freq(freq_), Tm(Tm_) {}

    void set_attributes()
    {
        set_timestep(100, sc_core::SC_NS);
        out.set_rate(1);
        out.set_timestep(100, sc_core::SC_NS);
    }
    void processing()
    {
        double t = get_time().to_seconds(); // actual time
        out.write(ampl * std::sin(2.0 * PI * freq * t));
        Mt.write(ampl * std::sin(2.0 * PI * freq * t));
    }

private:
    double ampl; // amplitude
    double freq; // frequency
    sca_core::sca_time Tm; // module time step
};
```

Fig. 1 message source code

```
SCA_TDF_MODULE(carrier_src)
{
    sca_tdf::sca_out<double> out; // output port
    sca_tdf::sca_de::sca_out<double> Ct;
    carrier_src(sc_core::sc_module_name nm, double ampl_ = 60, double freq_ = 1.0e6,
        sca_core::sca_time Tm_ = sca_core::sca_time(0.1, sc_core::SC_MS))
        : out("out"), ampl(ampl_), freq(freq_), Tm(Tm_), Ct("carrier") {}

    void set_attributes()
    {
        set_timestep(100, sc_core::SC_NS);
        out.set_rate(100);
        out.set_timestep(1, sc_core::SC_NS);
    }
    void processing()
    {
        double t = get_time().to_seconds(); // actual time
        out.write(ampl * std::sin(2.0 * PI * freq * t));
        Ct.write(ampl * std::sin(2.0 * PI * freq * t));
    }

private:
    double ampl; // amplitude
    double freq; // frequency
    sca_core::sca_time Tm; // module time step
};
#endif
```

Fig. 2 carrier source code

Timestep for message source is set to 100 ns. Rate of the port is 1 therefore timestep of the port is 100 ns.

Timestep of the carrier is 100 ns. Rate of the carrier port is 100 therefore carrier port timestep is 1 ns.

III. Mixer

To create a mixer, we need a processing function, a component that has 2 input ports with separate timestep and rate for each input signal and an output port. Code for mixer is shown below:

```
#ifndef MIXER_H
#define MIXER_H
#include "systemc.h"
#include "systemc-ams.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
#define PI 3.14159265358979323846264338327950
using namespace std;
SCA_TDF_MODULE(Mixer) {
    sca_tdf::sca_in<double>msg_in;
    sca_tdf::sca_in<double>carrier_in;
    sca_tdf::sca_de::sca_out<double> out; // output port
    sca_tdf::sca_de::sca_out<double> At;
    SCA_CTOR(Mixer) : out("out"), At("At") {}
    void set_attributes() {
        msg_in.set_rate(1);
        msg_in.set_timestep(100, sc_core::SC_NS);
        carrier_in.set_rate(100);
        carrier_in.set_timestep(1, sc_core::SC_NS);
        out.set_rate(100);
        out.set_timestep(1, sc_core::SC_NS);
        At.set_rate(100);
        At.set_timestep(1, sc_core::SC_NS);
        //set_timestep(sca_core::sca_time(0.1, sc_core::SC_MS));
    }
    void processing() {
        for (unsigned long i = 0; i < 100; i++) {
            double temp = (msg_in.read() * carrier_in.read(i)) + carrier_in.read(i);
            out.write(temp, i);
            At.write(temp, i);
        }
    }
};
#endif
```

Fig. 3 mixer code

According to Fig.3 , we set each input port according to the output port of the corresponding component. Timestep of the mixer is 100 ns and by having rate of each port we can figure out timestep of both.

IV. AM modulator

Now we get instance from each component to create AM modulator.

```cpp
#ifndef AM_MODULATOR_CPP
#define AM_MODULATOR_CPP
#include "Mixer.cpp"
#include "waves.cpp"
SC_MODULE(AM_Modulator) {
    sca_tdf::sca_signal<double> msg, carrier;
    sc_out<double>out;
    sc_out<double> message_trace, carrier_trace, mixer_trace;
    Mixer *mixer;
    message_src msg_src;
    carrier_src carrier_src;
    SC_CTOR(AM_Modulator): msg_src("msg_src_instance"),carrier_src("carrier_instance"){
        msg_src.out(msg);
        msg_src.Mt(message_trace);
        carrier_src.out(carrier);
        carrier_src.Ct(carrier_trace);
        mixer = new Mixer("mixer_instance");
        mixer->msg_in(msg);
        mixer->carrier_in(carrier);
        mixer->out(out);
        mixer->At(mixer_trace);
    }
};
#endif
```

Fig. 4 AM modulator

Output of the modulator is the output of the mixer.

V. RECTIFIER

To create a half wave rectifier, we use a converter to change sca_tdf signal (sample based signal) to discrete event signal. Code for this component is shown below:

```cpp
#ifndef HALFR_H
#define HALFR_H
#include "systemc.h"
#include "systemc-ams.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
SCA_TDF_MODULE(Half_wave_rectifier) {
    sca_tdf::sca_de::sca_in<double> input;
    sca_tdf::sca_de::sca_out<double> output;
    SCA_CTOR(Half_wave_rectifier):input("input"), output("output"){}
    void set_attributes() {
        //set_timestep(sca_core::sca_time(100, SC_US));
        //input.set_timestep(10, sc_core::SC_NS);
        //input.set_rate(100);
        output.set_timestep(1, sc_core::SC_NS);
        //output.set_rate(100);
    }
    void processing() {
        double temp = input.read();
        if (temp > 0) output.write(temp);
        else output.write(0.0);
    }
};
#endif
```

Fig. 5 Rectifier code

VI. FILTER

we create a simple bandpass filter with parallel resistor and capacitor. This circuit gets a discrete signal as current input source (converter is used) and emits a discrete voltage signal as output (another converter is used).

```cpp
#include "systemc-ams.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
SC_MODULE(Filter) {
    sc_in<double> input;
    sc_out<double> output;
    sca_eln::sca_c *cap;
    sca_eln::sca_r *res;
    sca_eln::sca_de_isource i_in;
    sca_eln::sca_de_vsink v_out;
    SC_HAS_PROCESS(sc_module_name);
    Filter(sc_module_name) : i_in("i_input", 1.0), v_out("v_out", 1.0) {
        i_in.p(a);
        i_in.n(gnd);
        i_in.inp(input);
        i_in.set_timestep(1, SC_NS);
        cap = new sca_eln::sca_c("capacitor", 5e-8);
        cap->n(gnd);
        cap->p(a);
        cap->set_timestep(1, SC_NS);
        res = new sca_eln::sca_r("resistor", 3.183e-1);
        res->p(a);
        res->n(gnd);
        v_out.p(gnd);
        v_out.n(a);
        v_out.outp(output);
        v_out.set_timestep(1, SC_NS);
    }
private:
    sca_eln::sca_node_ref gnd;
    sca_eln::sca_node a;
};
```

Fig. 6 RC filter

VII. DEMODULATOR

Now we get an instance from filter and rectifier and put them together.

```cpp
#ifndef DEMODULATOR_CPP
#define DEMODULATOR_CPP
#include "ELNmodule.cpp"
#include "HalfR.cpp"
SC_MODULE(Demodulator) {
    sc_in<double> input;
    sc_out<double> output;
    sc_signal<double> connection;
    Filter myfilter;
    Half_wave_rectifier rectifier;
    SC_CTOR(Demodulator):rectifier("rectifier"), myfilter("RC_filter"){
        rectifier.input(input);
        rectifier.output(connection);
        myfilter.input(connection);
        myfilter.output(output);
    }
};
#endif
```

Fig. 7 RC filter

This component is called a demodulator.

VIII. SYSTEM

At last, we create a class called full system and get instance of each signal and component that we use and monitor them.

```cpp
#ifndef FULLSYSTEM_CPP
#define FULLSYSTEM_CPP
#include "AM_Modulator.cpp"
#include "demodulator.cpp"
SC_MODULE(FullSystem) {
    AM_Modulator transmitter;
    Demodulator reciever;
    sc_signal<double> connection;
    sc_out<double> out,message_trace,carrier_trace;
    sc_out<double> modulated;
    SC_CTOR(FullSystem):transmitter("transmitter"),reciever("reciever") {
        transmitter.out(connection);
        reciever.input(connection);
        reciever.output(out);
        transmitter.message_trace(message_trace);
        transmitter.carrier_trace(carrier_trace);
        transmitter.mixer_trace(modulated);
    }
};
#endif
```

Fig. 8 Fullsystem design

IX. TESTBENCH

Now we create a testbench according to the code below and verify the functionality of the circuit.

```cpp
#ifndef TESTBENCH_CPP
#define TESTBENCH_CPP
#include "FullSystem.cpp"
SC_MODULE(Testbench) {
    sc_signal <double> modulated,output,msg_trace,carrier_trace;
    FullSystem* system;
    SC_CTOR(Testbench) {
        system = new FullSystem("HW7");
        system->out(output);
        system->modulated(modulated);
        system->message_trace(msg_trace);
        system->carrier_trace(carrier_trace);
    }
};
int sc_main(int argc, char* argv[]) {
    sc_set_time_resolution(1.0, SC_NS);
    sc_trace_file* tracefile = sc_create_vcd_trace_file("output");
    Testbench* tb = new Testbench("TB");
    sc_trace(tracefile, tb->output, "output");
    sc_trace(tracefile, tb->modulated, "mixer_out");
    sc_trace(tracefile, tb->msg_trace, "msg");
    sc_trace(tracefile, tb->carrier_trace, "carrier");
    sc_trace(tracefile, tb->system->reciever.connection, "HRout");
    sc_trace(tracefile, tb->system->reciever.myfilter.input, "filter_input");
    sc_start(1,SC_MS);
    return 0;
}
#endif
```

Fig. 9 Testbench

Testbench result is shown below:

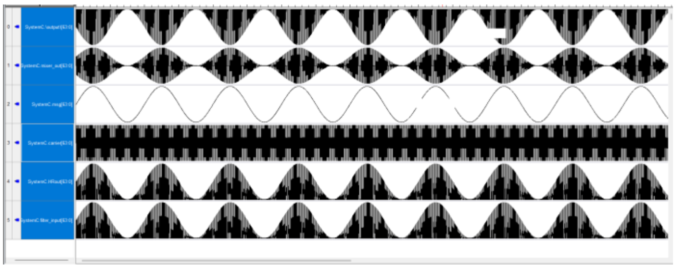

Fig. 10 Testbench result

As we can see testbench confirms the functionality of the circuit.