Politecnico di Torino
III Facoltà di Ingegneria

# Design of a 4-state add-compare-select with SystemC
# Integrated Systems Architecture

Master degree in Electronic Engineering

Group: 02

**Alessi Valeria 198141**
**Renzi Alessandro 197783**
**Tiralongo Antonio 200021**

February 5, 2014

# Contents

**1 4-state ACS** **1**
- 1.1 ACS . . . . . 6
- 1.2 D-type flip flop . . . . . 8

**2 Test-bench** **9**
- 2.1 SystemC simulation . . . . . 9
- 2.2 Modelsim simulation . . . . . 12
- 2.3 Modelsim SystemC-verilog mixed simulation . . . . . 14

**A** **17**

# CHAPTER 1

# 4-state ACS

The 4-state add-compare-select is one of the basic blocks of the **Viterbi Algorithm**. It is used to encode a bit stream by means of a forward error correction, based on a convolutional code. The Viterbi algorithm was developed by Andrew J. Viterbi in 1967, and today is widely used in error-correcting codes in cell phones, dial-up modems, satellite, deep-space communications, 802.11 wireless LANS, speech recognition systems, magnetic disk drives, and DNA research.
Since we are using four states to encode our signal, four ACS units are needed in order to keep track of the state metrics. A state metric can be defined as the "accumulated error metric", or the total branch metric through a path of the trellis diagram.
The block diagram of the whole 4-state ACS is shown in figure 1.1.

It accepts as inputs the state metrics of the current trellis step *C0* and *C1* and generates the new state metrics, combining *C0* and *C1* with the state metrics from the previous trellis step. In fact, as we can notice from the block diagram, there are FFs that introduce a feedback between the outputs and the inputs of the ACS blocks. The input and output ports , declared in SystemC using templates, are the following:

```
-----INPUT PORTS-------
sc_in<bool> clock;
sc_in<bool> enable;
sc_in<bool> reset;
sc_in<sc_int<5> > C1;
sc_in<sc_int<5> > C0;

-----OUTPUT PORTS-------
sc_out<sc_int<8> > A0;
sc_out<sc_int<8> > A1;
sc_out<sc_int<8> > A2;
sc_out<sc_int<8> > A3;
```

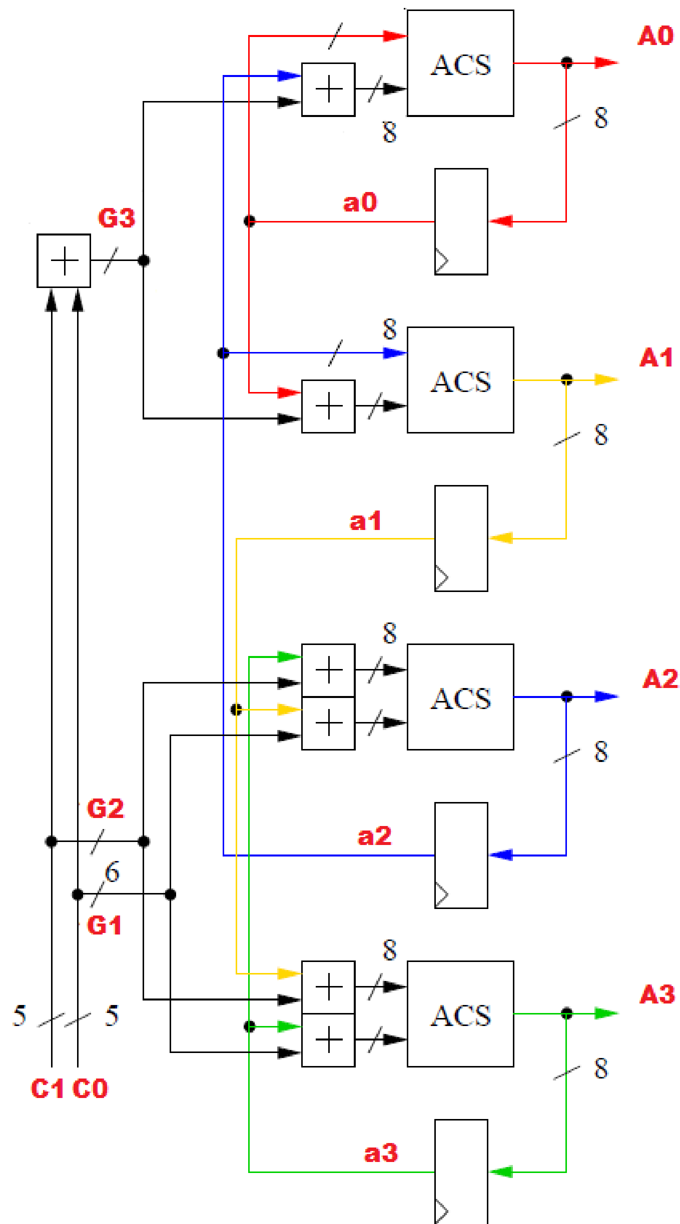


Figure 1.1: 4-state Add-compare-select block

The architecture of the entity is described inside the constructor, which has the same name of the entity, and receives as parameters the blocks *D* and *acs*, that will be instantiated in the *FourStateACS*.

```
SC_CTOR(FourStateACS) : D0("D0"), D1("D1"), D2("D2"), D3("D3"),
    acs0("acs0"), acs1("acs1"), acs2("acs2"), acs3("acs3")
{
   D0.D(A0);
   D0.Q(a0);
```

```
D0.clock(clock);
D0.enable(enable);
D0.reset(reset);

D1.D(A1);
D1.Q(a1);
D1.clock(clock);
D1.enable(enable);
D1.reset(reset);

D2.D(A2);
D2.Q(a2);
D2.clock(clock);
D2.enable(enable);
D2.reset(reset);

D3.D(A3);
D3.Q(a3);
D3.clock(clock);
D3.enable(enable);
D3.reset(reset);

acs0.A(a0);
acs0.B(S0);
acs0.Out(A0);

acs1.A(a2);
acs1.B(S1);
acs1.Out(A1);

acs2.A(S2);
acs2.B(S3);
acs2.Out(A2);

acs3.A(S4);
acs3.B(S5);
acs3.Out(A3);

SC_METHOD(add_0);
  sensitive << a2 << G3;

SC_METHOD(add_1);
  sensitive << a0 << G3;
```

```
  SC_METHOD(add_2);
    sensitive << a3 << G2;

  SC_METHOD(add_3);
    sensitive << a1 << G1;

  SC_METHOD(add_4);
    sensitive << a1 << G2;

  SC_METHOD(add_5);
    sensitive << a3 << G1;

  SC_METHOD(add_6);
    sensitive << C0 << C1;

  SC_METHOD(sign_ext_0);
    sensitive << C0;

  SC_METHOD(sign_ext_1);
    sensitive << C1;
}
```

Inside the constructor there is the port map of all *D* FFs and *acs* blocks, followed by the declaration of the combinational processes. They are all of SC_METHOD type. An SC_METHOD process is executed whenever an event occurs on its sensitivity list. The process then run to completion before returning control back to the SystemC scheduler. At that point the process is suspended until the next event. An SC_METHOD can be considered similar to a VHDL process with a static sensitivity list. For this reason, it is forbidden to insert wait statements inside an SC_METHOD.
There are six processes in charge of computing all the additions that are present in the block diagram, while the remaining processes are in charge of extending the sign of *C0* and *C1* of one bit more. This operation is required because *G3* is the result of the addition between *C1* and *C2* and so its parallelism is 6 bits (one bit more than the operands), while *G1* and *G2* are directly connected to *C1* and *C2*, so they would be represented only on 5 bits.
The implementation of the processes is described in the .ccp file *FourStateACS.cpp*.

```
#include "FourStateACS.h"

void FourStateACS::add_0(void)
{
```

```
   S0.write(a2.read() + G3.read());
}

void FourStateACS::add_1(void)
{
   S1.write(a0.read() + G3.read());
}

void FourStateACS::add_2(void)
{
   S2.write(a3.read() + G2.read());
}

void FourStateACS::add_3(void)
{
   S3.write(a1.read() + G1.read());
}

void FourStateACS::add_4(void)
{
   S4.write(a1.read() + G2.read());
}

void FourStateACS::add_5(void)
{
   S5.write(a3.read() + G1.read());
}

void FourStateACS::add_6(void)
{
   G3.write(C0.read() + C1.read());
}

void FourStateACS::sign_ext_0(void)
{
   G1.write(C0.read());
}

void FourStateACS::sign_ext_1(void)
{
   G2.write(C1.read());
}
```

Every time there is a variation on one of the signals of the sensitivity list, the operands are read, using the *read()* method, and the result of the addition is written, using the *write()* method, on the corresponding *S* signal.
As for the sign extension, the value of *C0/C1* is directly copied into *G0/G1*, because the value will remain the same with the only difference that it will be represented on a higher number of bits.
Outside the constructor, all signals, that are used to connect the different blocks of *FourStateACS*, are declared in a private section.

```
private:
   sc_signal<sc_int<6> > G1;
   sc_signal<sc_int<6> > G2;
   sc_signal<sc_int<6> > G3;

   sc_signal<sc_int<8> > a0;
   sc_signal<sc_int<8> > a1;
   sc_signal<sc_int<8> > a2;
   sc_signal<sc_int<8> > a3;

   sc_signal<sc_int<8> > S0;
   sc_signal<sc_int<8> > S1;
   sc_signal<sc_int<8> > S2;
   sc_signal<sc_int<8> > S3;
   sc_signal<sc_int<8> > S4;
   sc_signal<sc_int<8> > S5;
```

Now we can focus on the single components of the *FourStateACS* architecture.

## 1.1 ACS

The implementation of the *add-compare-select* unit is shown in figure 1.2.

The multiplexer selects either A or B, depending on the sign of the difference A-B. The subtractor and the multiplexer are implemented using two SC_METHODS, that are declared inside the constructor.

```
SC_CTOR(Acs)
{
   SC_METHOD(subtraction);
      sensitive << A << B;

   SC_METHOD(mux);
```

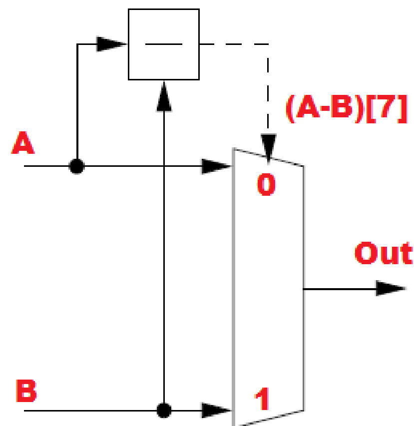


Figure 1.2: Add-compare-select block

```
    sensitive << A << B << sel;
}
```

If the difference A-B is greater or equal to 0, it means that the result is positive and so the control signal *sel* of the mux is set to false. Otherwise, the result is negative and *sel* is set to true.

```
void Acs::subtraction(void)
{
   if((A.read() - B.read()) >= 0)
   {
      sel.write(false);
   }
   else
   {
      sel.write(true);
   }
}
```

Depending on the value of *sel*, the mux connects its output either to the upper or to the lower input accordingly.

```
void Acs::mux(void)
{
   if(sel.read())
   {
      Out = B;
   }
```

```
   else
   {
      Out = A;
   }
}
```

## 1.2 D-type flip flop

There are as many flip flops as the number of ACS units. Flip flops are of D-type and are in charge of connecting the output of an ACS block to the input of another ACS block. The implementation is quite straightforward.

```
void ff_process(void)
{
   if(reset.read() == true)
   {
      Q.write(T(0));
   }
   else
   {
      if(enable.read() == true)
      {
         Q.write(D.read());
      }
   }
   return;
}
```

Every time a rising edge of the clock arrives, the SC_METHOD is activated and the value of the asynchronous reset is checked. If it is high the output $Q$ of the FF is set to zero, otherwise the input $D$ is copied to the output $Q$ if the *enable* signal is high. As we can see, the parallelism and the type of the I/O signals of the FF is defined using a template, that acts like a generic statement in VHDL. For this reason, in the top module FFs are instantiated by specifying *sc_int* as the type and eight as the number of bits of their ports.

```
Dff<sc_int<8> > D0;
Dff<sc_int<8> > D1;
Dff<sc_int<8> > D2;
Dff<sc_int<8> > D3;
```

# CHAPTER 2

# Test-bench

## 2.1 SystemC simulation

The first technique to simulate a SystemC module is to use the built-in SystemC simulator.
The test-bench requires an sc_main function that can be organized in several parts:

- Signals declaration

```
sc_signal<sc_int<5> > C0_i;
sc_signal<sc_int<5> > C1_i;
sc_signal<sc_int<8> > A0_i;
sc_signal<sc_int<8> > A1_i;
sc_signal<sc_int<8> > A2_i;
sc_signal<sc_int<8> > A3_i;

sc_signal<bool> enable;

sc_clock clock("CLOCK",sc_time(5, SC_NS));
sc_clock reset("RESET", sc_time(100, SC_NS), 0.1, sc_time(5, SC_NS),
    true);
```

  The clock has a period of 5 ns, while the reset remains at high level for 5 ns and then goes down for the rest of the simulation.

- Objects declaration

```
Generator gen("Generator");
FourStateACS uut("uut");
```

FourStateACS is the unit under test, while the generator is a module that produces two values for *C0* and *C1* at every clock cycle.

```
SC_MODULE (Generator)
{
public:
   sc_in<bool> clock;
   sc_in<bool> reset;
   sc_out<bool> enable;
   sc_out<sc_int<5> > data0; /// to be connected to C0 input of ACS4
   sc_out<sc_int<5> > data1; /// to be connected to C1 input of ACS4

   SC_CTOR(Generator)
   {
      SC_METHOD(generate);
         sensitive << clock.pos() << reset.pos();
   }

   void generate(void);

};
```

The generator has an SC_METHOD *generate*, that increments *value0* by 1 and *value1* by 2 every time a rising clock edge arrives. Both variables are declared with the static modifier, since they must not loose their value when the function returns the control to the scheduler.

```
void Generator::generate(void)
{
   static sc_int <5> value0;
   static sc_int <5> value1;

   if(reset.read())
      {
         value0 = 0;
         value1 = 0;
      }
   else
      {
         value0++;
         value1 += 2;
      }
```

```
    enable.write(true);
    data0.write(value0);
    data1.write(value1);
}
```

- Port map

```
gen.clock(clock);
gen.reset(reset);
gen.enable(enable);
gen.data0(C0_i);
gen.data1(C1_i);

uut.C0(C0_i);
uut.C1(C1_i);
uut.A0(A0_i);
uut.A1(A1_i);
uut.A2(A2_i);
uut.A3(A3_i);
uut.clock(clock);
uut.enable(enable);
uut.reset(reset);
```

Clock and reset signals are applied to both the modules. Then, the output ports of the generator are connected to the input ports of the uut through the signals *C0_i* and *C1_i*.

- VCD file creation

```
sc_trace_file *tf = sc_create_vcd_trace_file("uut");
((vcd_trace_file*)tf)->set_time_unit(1, sc_core::SC_NS);

sc_trace(tf, clock, "CLOCK");
sc_trace(tf, reset, "RESET");
sc_trace(tf, C0_i, "C0");
sc_trace(tf, C1_i, "C1");
sc_trace(tf, A0_i, "A0");
sc_trace(tf, A1_i, "A1");
sc_trace(tf, A2_i, "A2");
sc_trace(tf, A3_i, "A3");
```

```
sc_start(100, SC_NS);

sc_close_vcd_trace_file(tf);
```

Signals can be dumped in the *vcd* format that is a standard for waveforms and is accepted by *GTKWave* waveform viewer.
First of all, we have to declare a file pointer of type *sc_trace_file* and assign it a filename (uut), where the dump will be stored. Then, we set the resolution in ns and then the signals to be monitored are added to the file pointer using the command *sc_trace*, that receives as input parameters the name of the file pointer, the name of the signal, and the label that will be associated to the signal during simulation.
At this point the simulation can be stated with the command *sc_start* (specifying also the duration and the time unit) and at the end the vcd file will be closed.

The simulation results are presented in figure 2.1.

| Signal | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A0 | 06 | 00 | 03 | 08 | 0F | 19 | 26 | 35 | 46 | | 4D |
| A1 | 06 | 00 | 03 | 09 | 11 | 1B | 28 | 38 | 4A | 42 | 52 |
| A2 | 04 | 00 | 02 | 06 | 0D | 17 | 23 | 31 | 42 | 52 | 4B |
| A3 | 04 | 00 | 02 | 07 | 0F | 19 | 25 | 34 | 46 | 4E | 57 |
| C0 | 02 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 |
| C1 | 04 | 00 | 02 | 04 | 06 | 08 | 0A | 0C | 0E | 10 | 12 |

Figure 2.1: Integrated SystemC simulator results

## 2.2 Modelsim simulation

The previous test-bench cannot be adopted here, because Modelsim does not support the *sc_main* function, so we had to implement the test-bench as a SystemC module.

- Signals declaration

```
sc_signal< sc_int<5> > C0_i;
sc_signal< sc_int<5> > C1_i;
sc_signal< sc_int<8> > A0_i;
sc_signal< sc_int<8> > A1_i;
sc_signal< sc_int<8> > A2_i;
sc_signal< sc_int<8> > A3_i;

sc_signal<bool> enable;
```

```
sc_clock clk;
sc_clock reset;
```

- Objects declaration

```
Generator gen("Generator");
FourStateACS uut("uut");
```

- Constructor

```
SC_CTOR(tb_modelsim) : gen("Generator"), uut("utt"),
    clock("CLOCK",sc_time(5, SC_NS)),
   reset("RESET", sc_time(100, SC_NS), 0.1, sc_time(5, SC_NS), true)
       /// RESET is true for 5ns and then remain always false (duty
       cycle = 1)
{
   ///Port Map
   gen.clock(clock);
   gen.reset(reset);
   gen.enable(enable);
   gen.data0(C0_i);
   gen.data1(C1_i);

   uut.C0(C0_i);
   uut.C1(C1_i);
   uut.A0(A0_i);
   uut.A1(A1_i);
   uut.A2(A2_i);
   uut.A3(A3_i);
   uut.clock(clock);
   uut.enable(enable);
   uut.reset(reset);
}
```

The difference with respect to the previous testbench is that the waveforms have to be passed as arguments to the constructor and the port map has to be performed inside of it.
Vcd traces are no longer required.

- Compilation script

```
#!/bin/sh

sccom -g ../acs.cpp -I ../
sccom -g ../FourStateACS.cpp -I ../
sccom -g ../generator.cpp -I ../

sccom -g tb_modelsim.cpp -I ../
sccom -link
```

The results of the Modelsim's SystemC simulation are presented in figure 2.2. As we can see the results are obviously the same as the previous simulation.

Figure 2.2: Modelsim simulation results

## 2.3 Modelsim SystemC-verilog mixed simulation

In this case, the SystemC implementation of the 4-state ACS has to be replaced with the verilog description. So, the previous SystemC test-bench has to be slightly modified in order to deal with the verilog module.

- Signals declaration

```
sc_signal< sc_int<5> > C0_i;
sc_signal< sc_int<5> > C1_i;
sc_signal< sc_int<8> > A0_i;
sc_signal< sc_int<8> > A1_i;
sc_signal< sc_int<8> > A2_i;
sc_signal< sc_int<8> > A3_i;

sc_signal<bool> enable;

sc_clock clock;
sc_clock reset;
sc_clock clear;
```

- Objects declaration

```
Generator gen;
FourStateACS_rtl uut;
```

Instead of instantiating the SystemC version of *FourStateACS*, we have to instantiate the rtl version, in order to do this we have to wrap the verilog module within a SystemC module, this is accomplished using a specific tool of Modelsim: *scgenmod*.

```
scgenmod -bool -sc_int acs4 > FourStateACS_rtl.h
```

- Constructor

```
SC_CTOR(tb_mixed) : gen("Generator"), uut("utt", "acs4"),
    clock("CLOCK",sc_time(20, SC_NS)),
reset("RESET", sc_time(1000, SC_NS), 0.9, sc_time(5, SC_NS), false),
clear("CLEAR", sc_time(1000, SC_NS), 0.1, sc_time(5, SC_NS), true)
{
   ///Port Map
   gen.clock(clock);
   gen.reset(clear);
   gen.enable(enable);
   gen.data0(C0_i);
   gen.data1(C1_i);

   uut.C0(C0_i);
   uut.C1(C1_i);
   uut.A0(A0_i);
   uut.A1(A1_i);
   uut.A2(A2_i);
   uut.A3(A3_i);
   uut.CLK(clock);
   uut.CLEAR(clear);
   uut.RST_n(reset);

}
```

The difference with respect to the previous testbench is that the constructor requires two parameters for the uut: a string with the name of the object("uut") and a string with the name of the class ("FourStateACS_rtl").

- Compilation script

```
#!/bin/sh

vlog ../acs.v
vlog ../acs4.v

scgenmod -bool -sc_int acs4 > FourStateACS_rtl.h

sccom -g ../generator.cpp -I ../
sccom -DUSE_VLOG -g tb_mixed.cpp -I ../

sccom -link
```

The results of the mixed simulation are presented in figure 2.3.
As we can see also the third simulation confirms the previous results.

Figure 2.3: Mixed simulation results

# APPENDIX A

**FourStateACS.h**

```
#ifndef FOURSTATEACS_H
#define FOURSTATEACS_H

#include <systemc>
#include "dff.h"
#include "acs.h"

using namespace sc_core;
using namespace sc_dt;


SC_MODULE(FourStateACS)
{

public:

   sc_in<bool> clock;
   sc_in<bool> enable;
   sc_in<bool> reset;
   sc_in<sc_int<5> > C1;
   sc_in<sc_int<5> > C0;

   sc_out<sc_int<8> > A0;
   sc_out<sc_int<8> > A1;
   sc_out<sc_int<8> > A2;
   sc_out<sc_int<8> > A3;
```

```
Dff<sc_int<8> > D0;
Dff<sc_int<8> > D1;
Dff<sc_int<8> > D2;
Dff<sc_int<8> > D3;

Acs acs0;
Acs acs1;
Acs acs2;
Acs acs3;

SC_CTOR(FourStateACS) : D0("D0"), D1("D1"), D2("D2"), D3("D3"),
    acs0("acs0"), acs1("acs1"), acs2("acs2"), acs3("acs3")
{
   D0.D(A0);
   D0.Q(a0);
   D0.clock(clock);
   D0.enable(enable);
   D0.reset(reset);

   D1.D(A1);
   D1.Q(a1);
   D1.clock(clock);
   D1.enable(enable);
   D1.reset(reset);

   D2.D(A2);
   D2.Q(a2);
   D2.clock(clock);
   D2.enable(enable);
   D2.reset(reset);

   D3.D(A3);
   D3.Q(a3);
   D3.clock(clock);
   D3.enable(enable);
   D3.reset(reset);

   acs0.A(a0);
   acs0.B(S0);
   acs0.Out(A0);

   acs1.A(a2);
```

```
    acs1.B(S1);
    acs1.Out(A1);

    acs2.A(S2);
    acs2.B(S3);
    acs2.Out(A2);

    acs3.A(S4);
    acs3.B(S5);
    acs3.Out(A3);


    SC_METHOD(add_0);
      sensitive << a2 << G3;

    SC_METHOD(add_1);
      sensitive << a0 << G3;

    SC_METHOD(add_2);
      sensitive << a3 << G2;

    SC_METHOD(add_3);
      sensitive << a1 << G1;

    SC_METHOD(add_4);
      sensitive << a1 << G2;

    SC_METHOD(add_5);
      sensitive << a3 << G1;

    SC_METHOD(add_6);
      sensitive << C0 << C1;

    SC_METHOD(sign_ext_0);
      sensitive << C0;

    SC_METHOD(sign_ext_1);
      sensitive << C1;
  }

  void add_0(void);
  void add_1(void);
  void add_2(void);
```

```
   void add_3(void);
   void add_4(void);
   void add_5(void);
   void add_6(void);
   void sign_ext_0(void);
   void sign_ext_1(void);


private:
   sc_signal<sc_int<6> > G1;
   sc_signal<sc_int<6> > G2;
   sc_signal<sc_int<6> > G3;

   sc_signal<sc_int<8> > a0;
   sc_signal<sc_int<8> > a1;
   sc_signal<sc_int<8> > a2;
   sc_signal<sc_int<8> > a3;

   sc_signal<sc_int<8> > S0;
   sc_signal<sc_int<8> > S1;
   sc_signal<sc_int<8> > S2;
   sc_signal<sc_int<8> > S3;
   sc_signal<sc_int<8> > S4;
   sc_signal<sc_int<8> > S5;
};


#endif
```

**FourStateACS_rtl.h**

```
#ifndef _SCGENMOD_acs4_
#define _SCGENMOD_acs4_

#include "systemc.h"

class acs4 : public sc_foreign_module
{
public:
    sc_in<bool> CLK;
    sc_in<bool> RST_n;
    sc_in<bool> CLEAR;
    sc_in<sc_int<5> > C0;
```

```
    sc_in<sc_int<5> > C1;
    sc_out<sc_int<8> > A0;
    sc_out<sc_int<8> > A1;
    sc_out<sc_int<8> > A2;
    sc_out<sc_int<8> > A3;


    acs4(sc_module_name nm, const char* hdl_name)
     : sc_foreign_module(nm),
       CLK("CLK"),
       RST_n("RST_n"),
       CLEAR("CLEAR"),
       C0("C0"),
       C1("C1"),
       A0("A0"),
       A1("A1"),
       A2("A2"),
       A3("A3")
    {
        elaborate_foreign_module(hdl_name);
    }
    ~acs4()
    {}

};

#endif
```

**FourStateACS.cpp**

```
#include "FourStateACS.h"

void FourStateACS::add_0(void)
{
   S0.write(a2.read() + G3.read());
}

void FourStateACS::add_1(void)
{
   S1.write(a0.read() + G3.read());
}

void FourStateACS::add_2(void)
```

```
{
   S2.write(a3.read() + G2.read());
}

void FourStateACS::add_3(void)
{
   S3.write(a1.read() + G1.read());
}

void FourStateACS::add_4(void)
{
   S4.write(a1.read() + G2.read());
}

void FourStateACS::add_5(void)
{
   S5.write(a3.read() + G1.read());
}

void FourStateACS::add_6(void)
{
   G3.write(C0.read() + C1.read());
}

void FourStateACS::sign_ext_0(void)
{
   G1.write(C0.read());
}

void FourStateACS::sign_ext_1(void)
{
   G2.write(C1.read());
}
```

**acs.h**

```
#ifndef ACS_H
#define ACS_H

#include <systemc>

using namespace sc_core;
using namespace sc_dt;
```

```
SC_MODULE(Acs)
{

public:

   sc_in<sc_int<8> > A;
   sc_in<sc_int<8> > B;

   sc_out<sc_int<8> > Out;


   SC_CTOR(Acs)
   {
      SC_METHOD(subtraction);
         sensitive << A << B;

      SC_METHOD(mux);
         sensitive << A << B << sel;
   }



private:

   sc_signal<bool> sel;

   void subtraction(void);
   void mux(void);
};


#endif
```

**acs.cpp**

```
#include "acs.h"


void Acs::subtraction(void)
{
   if((A.read() - B.read()) >= 0)
```

```
    {
        sel.write(false);
    }
    else
    {
        sel.write(true);
    }
}

void Acs::mux(void)
{
    if(sel.read())
    {
        Out = B;
    }
    else
    {
        Out = A;
    }
}
```

**dff.h**

```
#ifndef DFF_H
#define DFF_H

#include <systemc>
// #include "sysc/kernel/sc_module.h"

using namespace sc_core;
using namespace sc_dt;



template<class T>
SC_MODULE(Dff)
{
public:
    sc_in<T> D;
    sc_in<bool> clock;
    sc_in<bool> reset;
    sc_in<bool> enable;
```

```
   sc_out<T> Q;

   SC_CTOR(Dff)
   {
      SC_METHOD(ff_process);
         sensitive << clock.pos();
   }


   void ff_process(void)
   {
      if(reset.read() == true)
      {
         Q.write(T(0));
      }
      else
      {
         if(enable.read() == true)
         {
            Q.write(D.read());
         }
      }
      return;
   }

};



#endif
```

## generator.h

```
#ifndef GENERATOR_H
#define GENERATOR_H

#include <systemc>
// #include "sysc/kernel/sc_module.h"

using namespace sc_core;
using namespace sc_dt;
```

```
SC_MODULE (Generator)
{
public:
   sc_in<bool> clock;
   sc_in<bool> reset;
   sc_out<bool> enable;
   sc_out<sc_int<5> > data0; /// to be connected to C0 input of ACS4
   sc_out<sc_int<5> > data1; /// to be connected to C1 input of ACS4

   SC_CTOR(Generator)
   {
      SC_METHOD(generate);
         sensitive << clock.pos() << reset.pos();
   }

   void generate(void);

};

#endif
```

**generator.cpp**

```
#include "generator.h"

void Generator::generate(void)
{
   static sc_int <5> value0;
   static sc_int <5> value1;

   if(reset.read())
      {
         value0 = 0;
         value1 = 0;
      }
   else
      {
         value0++;
         value1 += 2;
      }

   enable.write(true);
   data0.write(value0);
```

```
  data1.write(value1);
}
```

**tb_compiler.cpp**

```
//#include<stdio.h>
#include <systemc>
// #include "sysc/tracing/sc_vcd_trace.h"
#include "FourStateACS.h"
#include "generator.h"

using namespace sc_core;

int sc_main (int argc, char **argv)
{
  //sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
     SC_DO_NOTHING);

  sc_signal<sc_int<5> > C0_i;
  sc_signal<sc_int<5> > C1_i;
  sc_signal<sc_int<8> > A0_i;
  sc_signal<sc_int<8> > A1_i;
  sc_signal<sc_int<8> > A2_i;
  sc_signal<sc_int<8> > A3_i;

  sc_signal<bool> enable;

  sc_clock clock("CLOCK",sc_time(5, SC_NS));
  sc_clock reset("RESET", sc_time(100, SC_NS), 0.1, sc_time(5, SC_NS),
     true); /// RESET is true for 5ns and then remain always false (duty
     cycle = 1)


  ///Component instantiation
  Generator gen("Generator");
  FourStateACS uut("uut");

  ///Port Map
  gen.clock(clock);
  gen.reset(reset);
  gen.enable(enable);
  gen.data0(C0_i);
  gen.data1(C1_i);
```

```
    uut.C0(C0_i);
    uut.C1(C1_i);
    uut.A0(A0_i);
    uut.A1(A1_i);
    uut.A2(A2_i);
    uut.A3(A3_i);
    uut.clock(clock);
    uut.enable(enable);
    uut.reset(reset);

    ///VCD file creation
    sc_trace_file *tf = sc_create_vcd_trace_file("uut");
    ((vcd_trace_file*)tf)->set_time_unit(1, sc_core::SC_NS);

    sc_trace(tf, clock, "CLOCK");
    sc_trace(tf, reset, "RESET");
    sc_trace(tf, C0_i, "C0");
    sc_trace(tf, C1_i, "C1");
    sc_trace(tf, A0_i, "A0");
    sc_trace(tf, A1_i, "A1");
    sc_trace(tf, A2_i, "A2");
    sc_trace(tf, A3_i, "A3");

    sc_start(100, SC_NS);

    sc_close_vcd_trace_file(tf);

    return 0;
}
```

**tb_modelsim.cpp**

```
#include<stdio.h>
#include "FourStateACS.h"
#include "generator.h"

SC_MODULE(tb_modelsim)
{
    sc_signal<sc_int<5> > C0_i;
    sc_signal<sc_int<5> > C1_i;
    sc_signal<sc_int<8> > A0_i;
    sc_signal<sc_int<8> > A1_i;
```

```
   sc_signal<sc_int<8> > A2_i;
   sc_signal<sc_int<8> > A3_i;

   sc_signal<bool> enable;

   sc_clock clock;
   sc_clock reset;

   ///Component instantiation
   Generator gen;
   FourStateACS uut;

   SC_CTOR(tb_modelsim) : gen("Generator"), uut("utt"),
       clock("CLOCK",sc_time(5, SC_NS)),
   reset("RESET", sc_time(100, SC_NS), 0.1, sc_time(5, SC_NS), true) ///
       RESET is true for 5ns and then remain always false (duty cycle = 1)
   {
      ///Port Map
      gen.clock(clock);
      gen.reset(reset);
      gen.enable(enable);
      gen.data0(C0_i);
      gen.data1(C1_i);

      uut.C0(C0_i);
      uut.C1(C1_i);
      uut.A0(A0_i);
      uut.A1(A1_i);
      uut.A2(A2_i);
      uut.A3(A3_i);
      uut.clock(clock);
      uut.enable(enable);
      uut.reset(reset);
   }
};

SC_MODULE_EXPORT(tb_modelsim);
```

**tb_mixed.cpp**

```
#include<stdio.h>
#include "FourStateACS_rtl.h"
#include "generator.h"
```

```
SC_MODULE(tb_mixed)
{
   sc_signal< sc_int<5> > C0_i;
   sc_signal< sc_int<5> > C1_i;
   sc_signal< sc_int<8> > A0_i;
   sc_signal< sc_int<8> > A1_i;
   sc_signal< sc_int<8> > A2_i;
   sc_signal< sc_int<8> > A3_i;

   sc_signal<bool> enable;

   sc_clock clock;
   sc_clock reset;
   sc_clock clear;

   ///Component instantiation
   Generator gen;
   acs4 uut;

   SC_CTOR(tb_mixed) : gen("Generator"), uut("utt", "acs4"),
       clock("CLOCK",sc_time(20, SC_NS)),
   reset("RESET", sc_time(1000, SC_NS), 0.9, sc_time(5, SC_NS), false),
   clear("CLEAR", sc_time(1000, SC_NS), 0.1, sc_time(5, SC_NS), true)

   {
      ///Port Map
      gen.clock(clock);
      gen.reset(clear);
      gen.enable(enable);
      gen.data0(C0_i);
      gen.data1(C1_i);

      uut.C0(C0_i);
      uut.C1(C1_i);
      uut.A0(A0_i);
      uut.A1(A1_i);
      uut.A2(A2_i);
      uut.A3(A3_i);
      uut.CLK(clock);
      uut.CLEAR(clear);
      uut.RST_n(reset);
```

```
  }
};

SC_MODULE_EXPORT(tb_mixed);
```