# *SystemC 2 Verilog*

phuertao@opensocdesign.com
jcastillo@opensocdesign.com
www.opensocdesign.com

Rev. 1.2
January, 2005

**Revision History:**

| Rev | Date | Author | Description |
|---|---|---|---|
| 1.0 | 3/10/2004 | Javier Castillo | Initial Release |
| 1.1 | 18/12/2004 | Javier Castillo | Adapted to sc2v 0.2 |
| 1.2 | 17/01/2005 | Javier Castillo | Updates to fit sc2v 0.2.2 |
| | | | |
| | | | |
| | | | |
| | | | |

# 1 OVERVIEW

SystemC is a powerful language that allows the designer to develop a complete system description of his design.
From this system level design is necessary to get down to a RT synthesizable description that allows a physicall implementation of the model. But at this point there is a lack of synthesis tools that get a SystemC RT description as input. That means it is necessary a translation step to a supported HDL, that means Verilog or VHDL. Due to the similitude of Verilog and C it looks reasonable to convert the SystemC RT description to a Verilog equivalent one.

This translator performs this translation. From a SystemC description written following some rules we can obtain a synthesizable Verilog description supported by most of the synthesis tools available in the market.

# 2 Instructions of use

## *1- Compiling the sources:*

The sc2v translator is based on lex and yacc tools. You need lex and yacc installed before trying to compile sc2v.

For compiling the sources just type "make" under the directory you unzipped the fonts. It will generate three executables: sc2v_step1, sc2v_step2 and sc2v_step3.
.

## *2- Translating an SystemC module:*

A script that performs the translation is included in ./bin directory

Just type:

> sc2v.sh module_name

## *3- Format and restrictions of the SystemC files:*

- Supported SystemC types are:
  - bool
  - sc_uint<>
  - sc_int <>
  - sc_biguint <>
  - sc_bigint<>

Do not use C standard types as: char, int, long ……..

- Each module must have a .h file with the declarations of ports, signals, and processes, and there must exist a .cpp file with the code of the processes.

- For writing to a port or a signal you must ALWAYS use the .write() method.

- No functions supported

-Macros with no parameters are supported, but may cause little problems with name of variables. Macros with parameters are not supported.

-Only data types: bool, sc_int, sc_bigint, sc_uint and sc_biguint are supported.

-No global variables supported.

- No inout ports supported

## *4- Known bugs*

-The usage of macros and defines may cause some errors.

## *TODO:*

- Repair all known bugs.
- Support functions.
- Support macros.
- Support inout ports
- Support new constructions

# 3 Commented example: MD5

The code of the MD5 hash algorithm is included in the ./examples directory inside md5.h and md5.cpp files. The complete project including SystemC and Verilog testbenches is available at www.opencores.org webpage under the *systemcmd5* project.

In this chapter we will see the SystemC RT coding style in order to be automatically translated to Verilog.

The first thing you must take in account is that a file only can contain a module and it must be separated into a header file with the module declaration and a implementation file with the methods code.

## 3.1 HEADER FILES

First we will take a look to the md5.h file:

# *md5.h*

```
#include "systemc.h"

SC_MODULE (md5)
{

  //Port declaration
  sc_in < bool > clk;
  sc_in < bool > reset;

  sc_in < bool > load_i;
  sc_out < bool > ready_o;
  sc_in < bool > newtext_i;

  //Input must be padded and in little endian mode
  sc_in < sc_biguint < 128 > > data_i;
  sc_out < sc_biguint < 128 > > data_o;

  // -----------------------------------------
  //Signals
  sc_signal < sc_uint < 32 > >ar, br, cr, dr;
  sc_signal < sc_uint < 32 > >next_ar, next_br, next_cr, next_dr;
  sc_signal < sc_uint < 32 > >A, B, C, D;
  sc_signal < sc_uint < 32 > >next_A, next_B, next_C, next_D;

  sc_signal < bool > next_ready_o;
  sc_signal < sc_biguint < 128 > >next_data_o;
```

```
  sc_signal < sc_biguint < 512 > >message, next_message;
  sc_signal < bool > generate_hash, hash_generated,next_generate_hash;

  sc_signal < sc_uint < 3 > >getdata_state, next_getdata_state;

  sc_signal < sc_uint < 2 > >round, next_round;
  sc_signal < sc_uint < 6 > >round64, next_round64;

  sc_signal < sc_uint < 44 > >t;

  sc_signal < sc_uint < 32 > >func_out;

  //Method functions
  void md5_getdata ();
  void reg_signal ();
  void round64FSM ();
  void md5_rom ();
  void funcs ();

  SC_CTOR (md5)
  {

    SC_METHOD (reg_signal);
    sensitive_pos << clk;
    sensitive_neg << reset;

    SC_METHOD (md5_getdata);
    sensitive << newtext_i << data_i << load_i << getdata_state;
    sensitive << hash_generated << message;
    sensitive << func_out << A << B << C << D << ar << br << cr << dr;
    sensitive << generate_hash;

    SC_METHOD (round64FSM);
    sensitive << newtext_i << round << round64 << ar << br << cr <<dr;
    sensitive << generate_hash << func_out;
    sensitive << getdata_state << A << B << C << D;


    SC_METHOD (md5_rom);
    sensitive << round64;

    SC_METHOD (funcs);
    sensitive << t << ar << br << cr << dr << round << message;
    sensitive << func_out;

  }
};
```

The first part of the module is the port description.

As you read before, only sc_in and sc_out ports are supported. The types of these ports are bool for one bit ports as clk or reset. Other ports can be sc_int or sc_uint for ports with less than 64 bits and sc_biguint ot sc_bigint for ports with more than 64 bits.

Next section is the signal declaration. The signals have the same type restrictions ports have.

All the processes in a SystemC RT description must be declared as SC_METHOD. In this case we can see the declaration of combinational and sequential processes and then inside the constructor we declare the sensitivity list of this processed.

For sequential processed we use the following syntax:

```
    SC_METHOD (reg_signal);
    sensitive_pos << clk;
    sensitive_neg << reset;
```

Or this one:

```
SC_METHOD (reg_signal);
sensitive_pos (clk);
sensitive_neg (reset);
```

At the moment no clk.pos() construction and similar ones are supported, please use only the referred above.

For combinational processes use the following syntax, where you declare all the signals you read inside the process.

```
SC_METHOD (funcs);
sensitive (t);
sensitive (ar);
sensitive << br << cr << dr << round << message;
sensitive << func_out;
```

Both () and << are supported for sensitivity list declaration.

### 3.1.1 Instantiating modules

SystemC has many ways of instantiate modules inside another. At the moment only one constructions is allowed, using pointers to instantiate the module.

This code is taken from the *systemcaes* project, freely downloadable at www.opencores.org.

```
#include "systemc.h"
#include "word_mixcolum.h"

SC_MODULE(mixcolum)

{
      sc_in<bool> clk;
      sc_in<bool> reset;

      sc_in<bool> decrypt_i;
      sc_in<bool> start_i;
      sc_in<sc_biguint<128> > data_i;

      sc_out<bool> ready_o;
      sc_out<sc_biguint<128> > data_o;


      //Signals
      sc_signal<sc_biguint<128> > data_reg, next_data_reg, data_o_reg;
      sc_signal<sc_biguint<128> > next_data_o;

      sc_signal<bool> next_ready_o;

      //Methods
      void mixcol();
      void registers();
      void mux();
      void assign_data_o();

      //Signals
      sc_signal<sc_uint<2> > state, next_state;
      sc_signal<sc_uint<32> > outx, outy, mix_word, outmux;

      //Module declaration
      word_mixcolum *w1;
```

```
      SC_CTOR(mixcolum)
      {
            //Module instantiation
            w1 = new word_mixcolum("w1");
            w1->in(mix_word);
            w1->outx(outx);
            w1->outy(outy);

            SC_METHOD(assign_data_o);
            sensitive << data_o_reg;

            SC_METHOD(mux);
            sensitive << outx << outy;

            SC_METHOD(registers);
            sensitive_pos << clk;
            sensitive_neg << reset;


            SC_METHOD(mixcol);
            sensitive << decrypt_i << start_i << state << data_reg;
            sensitive << outmux << data_o_reg;
      }
};
```

In this example first we include the module to be instantiated:

```
#include "word_mixcolum.h"
```

Then we can create a pointer to the module, instantiate it inside the constructor and connect the signals to the ports.

```
      //Module declaration
      word_mixcolum *w1;


      SC_CTOR(mixcolum)

      {
            //Module instantiation
            w1 = new word_mixcolum("w1");
            w1->in(mix_word);
            w1->outx(outx);
            w1->outy(outy);

            .......
```

## *3.2 IMPLEMENTATION FILES*

Now we will see a example of implementation of a sequential process and another example of a combinational one.

First we will see the sequential process:

```
void md5::reg_signal ()
{
  if (!reset)
    {
      ready_o.write (0);
      data_o.write (0);
      message.write (0);

      ar.write (0x67452301);
```

```
      br.write (0xEFCDAB89);
      cr.write (0x98BADCFE);
      dr.write (0x10325476);

      getdata_state.write (0);
      generate_hash.write (0);

      round.write (0);
      round64.write (0);

      A.write (0x67452301);
      B.write (0xEFCDAB89);
      C.write (0x98BADCFE);
      D.write (0x10325476);

    }
  else
    {
      ready_o.write (next_ready_o.read ());
      data_o.write (next_data_o.read ());
      message.write (next_message.read ());

      ar.write (next_ar.read ());
      br.write (next_br.read ());
      cr.write (next_cr.read ());
      dr.write (next_dr.read ());

      A.write (next_A.read ());
      B.write (next_B.read ());
      C.write (next_C.read ());
      D.write (next_D.read ());

      generate_hash.write (next_generate_hash.read ());
      getdata_state.write (next_getdata_state.read ());

      round.write (next_round.read ());
      round64.write (next_round64.read ());

    }

}
```

You must notice one important thing, always use the .write() method when writing to a signal or port, is the only way the translator have to distinguish between a variable and a signal. The .read() method is not necessary but is recommended.

The recommended style for a sequential process is:

```
     if(reset){

     }else{

     }
```

But any other might be used.

A combinational process has the following aspect:

```
md5::md5_getdata ()
{

  sc_biguint < 128 > data_o_var;
  sc_biguint < 512 > aux;

  sc_uint < 32 > A_t, B_t, C_t, D_t;
```

```
  next_A.write (A.read ());
  next_B.write (B.read ());
  next_C.write (C.read ());
  next_D.write (D.read ());

  next_generate_hash.write (0);
  next_ready_o.write (0);
  next_data_o.write (0);

  aux = message.read ();
  next_message.write (message.read ());
  next_getdata_state.write (getdata_state.read ());

  if (newtext_i.read ())
    {
      next_A.write (0x67452301);
      next_B.write (0xEFCDAB89);
      next_C.write (0x98BADCFE);
      next_D.write (0x10325476);
      next_getdata_state.write (0);
    }


  ...........
}
```

Local variables must be declared at the beginning of the process, declarations as:

```
sc_uint<32> temp=A;
```

inside a process are not allowed.

The types allowed for variables are the same permitted for ports and signals.

After the translation all the local variables inside a process are converted to Verilog reg type and a prefix with the name of the process is added to each one. For example in the process `funcs` the variable aux is converted to `auxfuncs`.

The beginning of the equivalent Verilog code for this process is:

```
//md5_getdata:
reg[127:0] data_o_varmd5_getdata;
reg[511:0] auxmd5_getdata;
reg[31:0] A_tmd5_getdata,B_tmd5_getdata,C_tmd5_getdata,D_tmd5_getdata;
always @(newtext_i or data_i or load_i or getdata_state or
hash_generated or message or func_out or A or B or C or D or ar or
br or cr or dr or generate_hash)

begin

   next_A  = (A );
   next_B  = (B );
   next_C  = (C );
   next_D  = (D );

   next_generate_hash  = (0);
   next_ready_o  = (0);
   next_data_o  = (0);

   auxmd5_getdata =message ;
   next_message  = (message );
   next_getdata_state  = (getdata_state );

   if (newtext_i )
```

```
   begin

      next_A  = ('h67452301);
      next_B  = ('hEFCDAB89 );
      next_C  = ('h98BADCFE );
      next_D  = ('h10325476);
      next_getdata_state  = (0);

   end

 ......................

end
```

## 3.3 ENUMERATED DATA TYPES

Enumerated data types are supported in order to describe FSM.

You can declare a signal as enumerated in this way:

```
enum state_t {S0,S1,S2};
sc_signal<state_t> state;
```

or in this other:

```
enum {S0,S1,S2} state;
```

or using this second one the translator assumes the variable declared is a sc_signal.

Automatic calculation of the number of bits required to store the enumerated signals is performed by the translator.

The enumerated types described above are translated into the following lines:

```
parameter  S0=0,
           S1=1,
           S2=2;
```

## 3.4 TRANSLATOR DIRECTIVES

Currently sc2v support two directives

```
//translate off
…
//translate on
```

With this directives the lines between them are ignored.

```
/*Verilog begin
……
Verilog end*/
```

In this case the lines between the directives are copy to the verilog file without translate it

# 4 Conclusions

Since this is a beta version of the tool we recommend you to manually review the code after the translation.

For more information about SystemC Synthesizable Subset read "**Describing Synthesizable** RTL in **SystemC**".

Please send comments, feedback, contributions to:

phuerta@opensocdesign.com
jcastillo@opensocdesign.com