Mazdak & Alborz Design Automation

V2SC Quick Guide

Ali Haj Abutalebi

Leila Mahmoudi Ayough

Table of Contents

General Information

V2SC is a Verilog HDL into SystemC translator. V2SC lets use pre-designed Verilog IPs in SystemC environments which has lower cost and higher speed at the same time. It supports all Verilog constructs except for switch-level components and "specify" block and some other rare-used constructs. Apart from these, all Verilog constructs including dynamic events, delays, wait statements, casex and all the others are fully supported. V2SC guarantees Verilog synthesizable subset conversion. The conversion procedure is fully automatic and no part is converted manually. V2SC converts Verilog test-benches too making functional-test of pre-designed IPs feasible.

V2SC is based on Verilog 2001 IEEE standard and our target is SystemC 2.1.

V2SC is available in Linux and Windows platforms.

V2SC Command and Options

To run V2SC in command line, type:

```
v2sc input_verilog_file.v [-p] [-n] [-d] [-r] [-m  mName]
```

Command line options are for obtaining best output, depending on nature of Verilog input code. Each of above options are interpreted by V2SC as follows:

V2SC Options:

-p: Each module in Verilog is translated into a SC_MODULE SystemC class. If "-p" option is present, if there were any parameters declared in the Verilog module, a templated SC_MODULE class would be generated through which the parameters are passed into the class. At the default case, i.e., while no "-p" option is present, parameters are all interpreted as static constants in the corresponding SC_MODULE class.

-n: While present, each nonblocking assignments is mapped into an individual separate process in the target SystemC file. At the default case, i.e., while no "-n" option is present, all the nonblocking assignments are mapped into a single process in the target SystemC file.

-d: While present, delays and events are ignored. At the default case, i.e., while no "-d" option is present, delays and events are supported.

-r: While present, this product is registered in your computer. It's needed only at the first run, once v2sc is registered in your computer, it will run at the next time with no trouble.

-m: While present, module named "mName" will be assumed as the top level module. A verilog_file_name_main.cpp file will be generated in which module mName will be instantiated and port mapped with the appropriate declared signals. At the default case, i.e., while no "-m" option is present, no verilog_file_name_main.cpp file is generated.

Registering

The first time that v2sc is getting launched in your computer, it needs to be registered. “-r” option in command line does this task. Simply write the following command in command line:

```
v2sc –r
```

You need to have a key in order to register v2sc.

This version does not require registration.

A Simple Example

Here is the representation of an adder along with its test-benches in Verilog.

full_adder.v

```
module full_adder(sum,cout,a,b,cin);
     input  a,b,cin;
     output sum,cout;

     assign {cout, sum} = a + b + cin;
endmodule
```

Fig. 1: Verilog description of a full adder.

adder.v

```
module four_bit_adder(sum, cout, a, b, cin);
     input  [3:0] a, b;
     input  cin;
     output [3:0] sum;
     output cout;

     wire   s1,s2,s3;

     full_adder m0(sum[0],s1, a[0],b[0],cin);
     full_adder m1(sum[1],s2, a[1],b[1],s1);
     full_adder m2(sum[2],s3, a[2],b[2],s2);
     full_adder m3(sum[3],cout, a[3],b[3],s3);
endmodule
```

Fig. 2: Verilog description of a four-bit adder.

test_adder.v

```
module test_adder;
     reg  [3:0] a, b;
     reg cin;
     wire [3:0] sum;
     wire cout;

     four_bit_adder m1(sum, cout, a, b, cin);

     initial
     begin
          a = 0;
          b = 0;
          cin = 0;
          #5;
          a = 4;
          b = 8;
          cin = 0;
          #5;
          a = 6;
          b = 3;
          cin = 1;
          #5;

          a = 5;
          b = 12;
          cin = 0;
          #5;
          a = 8;
          b = 8;
          cin = 0;
          #5;
          a = 9;
          b = 1;
          cin = 1;
          #5;
          $stop;
     end
endmodule
```

Fig. 3: Verilog test-bench of adder.

By the following command we can convert the above Verilog files to corresponding SystemC ones.

```
C:\v2sc\v2sc.exe      full_adder.v  adder.v  test_adder.v -m  test_adder
```

The following SystemC files are generated.

Table 1: Generated SystemC files.

full_adder_full_adder.h adder_four_bit_adder.h test_adder_test_adder.h	full_adder_full_adder.cpp adder_four_bit_adder.cpp test_adder_test_adder.cpp
	test_adder_main.cpp

In order to generate sc_main function that is required for SystemC simulation, Verilog top module name should be declared in command line after "–m" option. In sc_main function the top-module of design is instantiated and demanded signals for port-mapping this module are declared. SystemC commands for tracing output signals are also located in this function. All these required lines are automatically generated by "-m" option.

full_adder_full_adder.h

```
SC_MODULE(full_adder)
{
     //Interface Ports...
     sc_out<bool>   sum;
     sc_out<bool>   cout;
     sc_in<bool>    a;
     sc_in<bool>    b;
     sc_in<bool>    cin;

     //Processes Declaration...
     void assign_process_cout_sum_5();

     //Constructor Declaration...
     SC_CTOR(full_adder)
     {
          SC_METHOD(assign_process_cout_sum_5);
          sensitive << a << b << cin;
     }
};
```

full_adder_full_adder.cpp

```
#include "full_adder_full_adder.h"

void full_adder::assign_process_cout_sum_5()
{
     sc_uint<2> coutsum_concat;
     coutsum_concat = a.read() + b.read() + cin.read();
     cout = coutsum_concat[1];
     sum = coutsum_concat[0];
}
```

Fig. 4: Generated SystemC files for full_adder.

adder_four_bit_adder.h

```
#include "full_adder_full_adder.h"

SC_MODULE(four_bit_adder)
{
     //Interface Ports...
     sc_out<sc_uint<4> > sum;
     sc_out<bool>   cout;
     sc_in<sc_uint<4> >  a;
     sc_in<sc_uint<4> >  b;
     sc_in<bool>    cin;

     //Internal Signals...
     sc_signal<bool>     s1;
     sc_signal<bool>     s2;
     sc_signal<bool>     s3;

     //Instantiated Modules Object Declaration...
     full_adder     m0;
     full_adder     m1;
     full_adder     m2;
     full_adder     m3;

     //Local Temporary Signals...
     sc_signal<bool> sum_0;
     sc_signal<bool> a_0;
     sc_signal<bool> b_0;
     sc_signal<bool> sum_1;
     sc_signal<bool> a_1;
     sc_signal<bool> b_1;
     sc_signal<bool> sum_2;
     sc_signal<bool> a_2;
     sc_signal<bool> b_2;
     sc_signal<bool> sum_3;
     sc_signal<bool> a_3;
     sc_signal<bool> b_3;

     //Signal Handler...
     void signal_handler()
     {
          sc_uint<4> sum_tmp(sum.read());
          sum_tmp[0] = sum_0.read();
          sum = sum_tmp;
          a_0 = a.read()[0];
          b_0 = b.read()[0];
          sum_tmp[1] = sum_1.read();
          sum = sum_tmp;
          a_1 = a.read()[1];
          b_1 = b.read()[1];
          sum_tmp[2] = sum_2.read();
          sum = sum_tmp;
          a_2 = a.read()[2];
          b_2 = b.read()[2];
          sum_tmp[3] = sum_3.read();
          sum = sum_tmp;
          a_3 = a.read()[3];
          b_3 = b.read()[3];
     }

     //Constructor Declaration...
     SC_CTOR(four_bit_adder) :
          m0("M0") ,
          m1("M1") ,
          m2("M2") ,
          m3("M3")
     {
          m0(sum_0,s1,a_0,b_0,cin);

          m1(sum_1,s2,a_1,b_1,s1);

          m2(sum_2,s3,a_2,b_2,s2);

          m3(sum_3,cout,a_3,b_3,s3);

          SC_METHOD(signal_handler);
          sensitive << sum_0 << a << b << sum_1 << sum_2 << 
sum_3;
     }
};
```

adder_four_bit_adder.cpp

```
#include "adder_four_bit_adder.h"
```

Fig. 5: Generated SystemC files for adder.

test_adder_test_adder.h

```
#include "adder_four_bit_adder.h"

SC_MODULE(test_adder)
{
     //Internal Signals...
     sc_signal<sc_uint<4> >   a;
     sc_signal<sc_uint<4> >   b;
     sc_signal<bool>     cin;
     sc_signal<sc_uint<4> >   sum;
     sc_signal<bool>     cout;

     //Processes Declaration...
     void initial_process_8();

     //Instantiated Modules Object Declaration...
     four_bit_adder m1;

     //Constructor Declaration...
     SC_CTOR(test_adder) :
          m1("M1")
     {
          m1(sum,cout,a,b,cin);

          SC_THREAD(initial_process_8);
     }
};
```

test_adder_test_adder.cpp

```
#include "test_adder_test_adder.h"

void test_adder::initial_process_8()
{
     a = 0;
     b = 0;
     cin = 0;
     wait(5,SC_NS);
     a = 4;
     b = 8;
     cin = 0;
     wait(5,SC_NS);
     a = 6;
     b = 3;
     cin = 1;
     wait(5,SC_NS);
     a = 5;
     b = 12;
     cin = 0;
     wait(5,SC_NS);
     a = 8;
     b = 8;
     cin = 0;
     wait(5,SC_NS);
     a = 9;
     b = 1;
     cin = 1;
     wait(5,SC_NS);
     sc_stop();
}
```

Fig. 6: Generated SystemC files for test_adder.

test_adder_main.cpp

```
#include "test_adder_test_adder.h"

int sc_main(int ac, char *av[])
{
     //Instantiated Top Module...
     test_adder     dut("dut");

     //VCD File...
     sc_trace_file *tf =
                    sc_create_vcd_trace_file("test_adder");
     sc_trace(tf, dut.a, "a");
     sc_trace(tf, dut.b, "b");
     sc_trace(tf, dut.cin, "cin");
     sc_trace(tf, dut.sum, "sum");
     sc_trace(tf, dut.cout, "cout");

     //Start Simulation...
     sc_start(-1);

     sc_close_vcd_trace_file(tf);

     return 0;
}
```

Fig. 7: Generated sc_main SystemC file.

In order to see the see the result of simulation of SystemC simply compile and link generated systemc files. After running the executable file you can see the generated waveform in a vcd viewer. The following figures. depict two wave-forms of the above adder. The first one is the Verilog-based description simulation result performed in ModelSim simulator, and the second one is SystemC-based description simulation result performed in Evan environment.

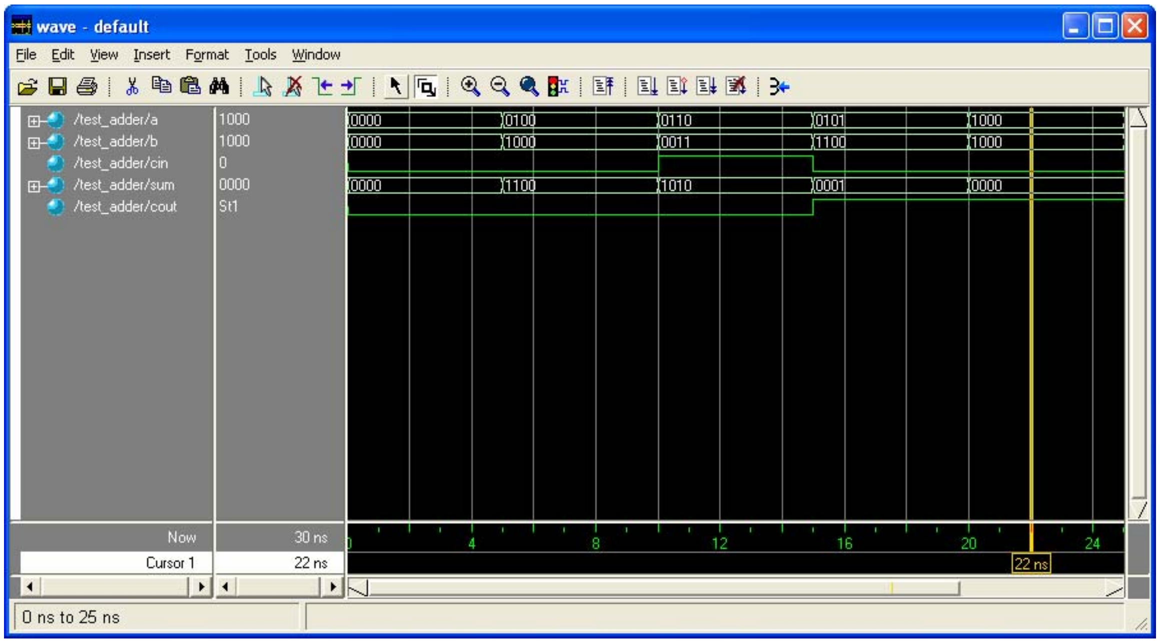
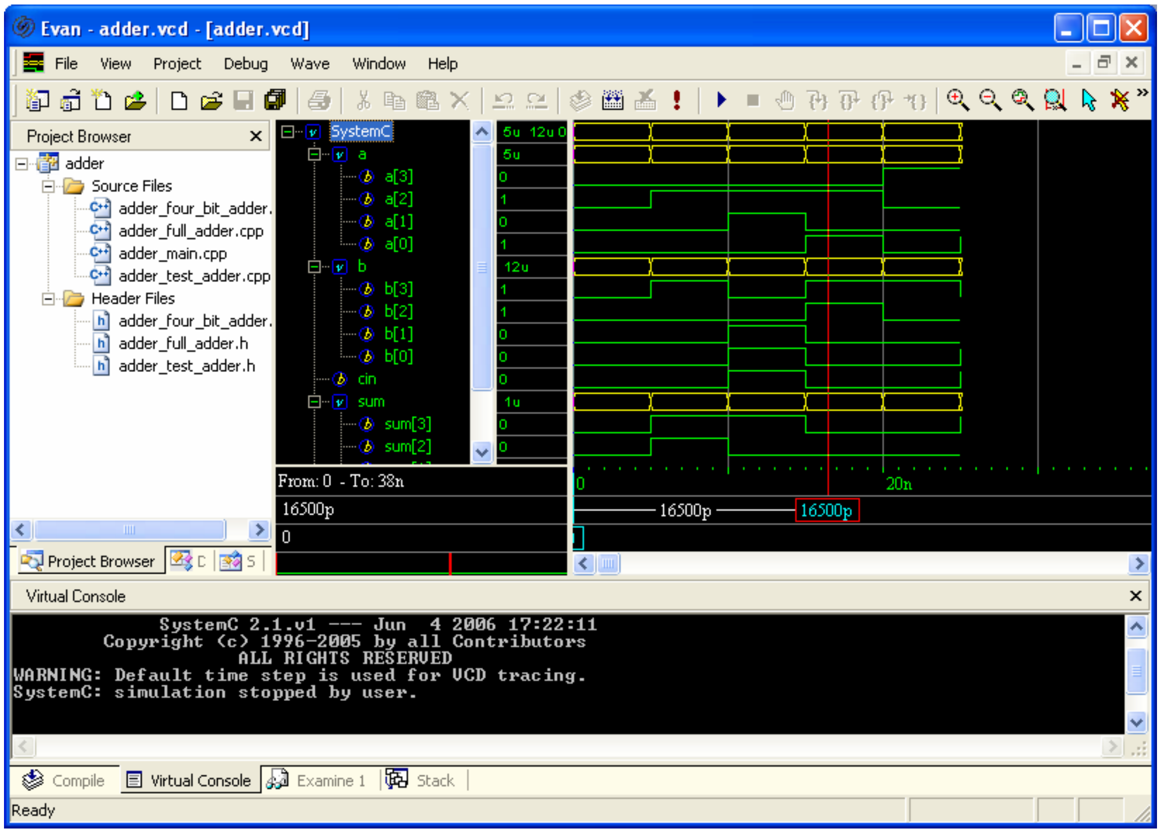


Fig. 8: Verilog simulation result of a simple 4-bit adder in Modelsim.



```
          SystemC 2.1.v1 --- Jun  4 2006 17:22:11
      Copyright (c) 1996-2005 by all Contributors
                 ALL RIGHTS RESERVED
WARNING: Default time step is used for VCD tracing.
SystemC: simulation stopped by user.
```

Fig. 9: SystemC simulation result of a simple 4-bit adder in Evan.

Not-Supported Constructs

V2SC does not support some features of Verilog. There are some technical reasons for not supporting them. In this chapter we'll take a closer look at them.

1. Procedural Continuous Assignment (assign/deassign/force/release)

There is not an equivalent code for procedural continuous assignment in SystemC. Procedural continuous assignment means assign, deassign, force and release in Verilog always and initial body.

The following figure shows an example.

```
module flipflop (d, clk, reset, q);
input d, clk, reset;
output q;
reg q;
     always @(reset) begin
          if (reset == 1)
               assign q = 0;
          else
               deassign q;
     end
     always @(posedge clk) begin
          q = d;
     end
endmodule
```

Fig. 10: assign/deassign example.

2. Verilog Switch Level

There is not an equivalent code for Verilog switch level in SystemC.

The following figure shows an example.

```
module cmos_not (in, out);
input in;
output out;
supply1 vdd;
supply0 gnd;
   nmos g1 (out, gnd, in);
   pmos g2 (out, vdd, in);
endmodule
```

Fig. 11: Verilog Switch-Level example.

3. Specify Block

There is not an equivalent code for Verilog specify construct in SystemC. The following figure shows an example.

```
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

specify
     (a => out) = 9;
     (b => out) = 9;
     (c => out) = 11;
     (d => out) = 11;
endspecify

and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule
```

Fig. 12: Specify block example.

4. System Tasks and Functions

V2SC supports a subset of Verilog System tasks. The following table shows the supported ones.

Table 2: Supported System Tasks and Functions.

$stop
$finish
$display, $displayb, $displayh, $displayo
$write, $writeb, $writeh, $writeo
$strobe, $strobeb, $strobeh, $strobeo
$monitor
$time
$stime
$realtime
$itor, $rtoi

The other Verilog system tasks are not supported.

5. Fork/Join Constructs

V2SC supports fork/join construct mostly for creating the test-benches. But fork/join can also be used in modeling hardware. V2SC dose not supported fork/join constructs for hardware modeling.

The following figures show how fork/join construct are used both in test-benches and hardware modeling.

```
module test_bench;
reg a,b,c;
initial
     fork
          a=0;b=0;
          #5 a=1;
          #10 b =1;
          #20 a=0;
          #30 b=0;
          #50 $stop;
     join
endmodule
```

Fig. 13: Supported fork/join in test-bench.

```
module iaf4;
integer i, j;

initial
begin
     i = 3;
     j = 4;
     #1
     fork
          i = #1 j;
          j = #1 i;
     join
end
endmodule
```

Fig. 14: Not-Supported fork/join.

6. Four-Value Logic Operators

V2SC is equipped by an algorithm for recognizing two-value and four-value logic. SystemC library supports two-value logic extensively but lacks some operators for four-value logic. For example four-value comparison expression is one of them.

The following figure shows an example.

```
module dcd_3_to_8(adr, so);
input [2:0] adr;
output [7:0] so;
assign so =
     adr === 3'b000 ? 8'b00000001 :
     adr === 3'b001 ? 8'b00000010 :
     adr === 3'b00Z ? 8'b00000010 :
     adr === 3'b010 ? 8'b00000100 :
     adr === 3'b0Z0 ? 8'b00000100 :
     adr === 3'b011 ? 8'b00001000 :
     adr === 3'b01Z ? 8'b00001000 :
     adr === 3'b0Z1 ? 8'b00001000 :
     adr === 3'b0ZZ ? 8'b00001000 :
     adr === 3'b100 ? 8'b00010000 :
     adr === 3'b101 ? 8'b00100000 :
     adr === 3'b10Z ? 8'b00100000 :
     adr === 3'bZ01 ? 8'b00100000 :
     adr === 3'bZ0Z ? 8'b00100000 :
     adr === 3'b110 ? 8'b01000000 :
     adr === 3'b1Z0 ? 8'b01000000 :
     adr === 3'bZ10 ? 8'b01000000 :
     adr === 3'bZZ0 ? 8'b01000000 :
     adr === 3'b111 ? 8'b10000000 :
     adr === 3'b11Z ? 8'b10000000 :
     adr === 3'b1Z1 ? 8'b10000000 :
     adr === 3'b1ZZ ? 8'b10000000 :
     adr === 3'bZ11 ? 8'b10000000 :
     adr === 3'bZ1Z ? 8'b10000000 :
     adr === 3'bZZ1 ? 8'b10000000 :
     adr === 3'bZZZ ? 8'b10000000 :
     8'BXXXXXXXX;
endmodule
```

Fig. 15: Four-Value comparison example.

7. Disable Statement

V2SC supports Verilog disable statements in a loop statement. But if a disable statement is used for disabling a task or another process, there is no equivalent code for it.

The following figures show two examples of disable statement.

```
module find_true_bit;
reg [15:0] flag;
integer i;
initial
begin
     flag = 16'b0010_0000_0000_0000;
     i = 0;
     begin: block1
          while(i < 16)
          begin
               if (flag[i])
               begin
                    $display("Encountered a TRUE bit at
                             element number %d", i);
                    disable block1;
               end
               i = i + 1;
          end
     end
end
endmodule
```

Fig. 16: Disable statement for a loop.

```
module disable1;

initial begin
     do_it;
     $display("Finished do it at time %0d",$time);
End

initial begin
#55 disable do_it;
End

task do_it;
     forever
     #10 $display("doing it at time %0d",$time);
endtask

endmodule
```

Fig. 17: Disable statement for a task.

8. Verilog 2001 Generate Statement and Elaboration Time Constructs

Generate statement and some features of Verilog language are performed in elaboration phase. V2SC is a tool for translating Verilog to SystemC and does not have elaboration phase. Therefore V2SC does not support theses constructs.

The following figure shows an example. This figure shows a conditional instantiation. V2SC does not support this example.

```
module multiplier (a, b, product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width;
input [a_width-1:0] a;
input [b_width-1:0] b;
output [product_width-1:0] product;

generate
if((a_width < 8) || (b_width < 8))
     CLA_multiplier #(a_width, b_width)
          u1 (a, b, product);
else
     WALLACE_multiplier #(a_width, b_width)
          u1 (a, b, product);
endgenerate

endmodule
```

Fig. 18: Generate statement.