# USING THE SYSTEM-C LIBRARY FOR BIT–TRUE SIMULATIONS IN MATLAB

*Jan Schier*

Institute of Information Theory and Automation
Academy of Sciences of the Czech Republic

**Abstract**

In the paper, the possibilities of bit-true simulations in Matlab/Simulink are discussed, with focus on integrating the SystemC simulation language into the Matlab environment.

## 1 Introduction

Matlab/Simulink is well known for its flexibility, rich set of features and the rapid prototyping capabilities. With a number of high-quality toolboxes, it has gained popularity in both scientific and engineering communities. In this paper, we will consider its use for design of the DSP applications, namely for bit-true simulations. We will focus on integrating the Matlab environment with the SystemC design language.

### 1.1 DSP application design flow

For a typical DSP application, the design steps are shown in Fig. 1. Let us give some comments on each of them:

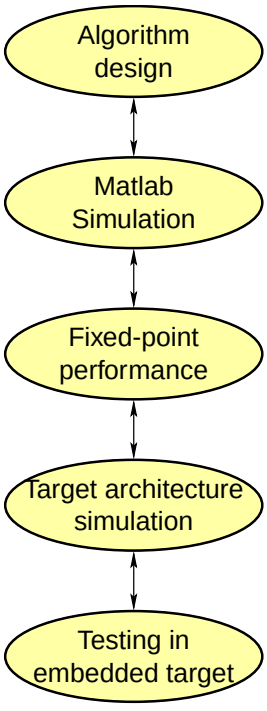



Figure 1: Design flow of a DSP application

1. *Algorithm design* – theoretical derivation of the algorithm. In this phase, decisions are taken on the "mathematical architecture" of the application. This step includes choosing an algorithm from existing ones or deriving a new one, theoretical considerations on the complexity, convergence, modularity and other properties of the algorithm
2. *Matlab/Simulink implementation* – very often, the theoretical analysis cannot answer all questions we ask about the algorithm properties, or an exact analysis would be too costly. The reasons may be complex character of the data to be processed, nonlinearities included in the signal path, etc.

For this reason, and also as a certain “engineers proof-of-concept”, an implementation in a rapid prototyping environment (in our case, in a form of a Matlab program or Simulink model) is used. The Matlab environment is “ideal” in a sense – the computations are implemented in a double-precision floating-point arithmetics, but it already gives good insight on the function of algorithm and, thanks to the number of available toolboxes, it saves lot of programming work.

3. *Fixed-point simulation* – the typical target environment for the DSP applications is an embedded system, using either a DSP processor, or an FPGA/ASIC chip. In the case of FPGA or ASIC, it is (except the high-end FPGA chips) costly to implement the floating-point arithmetic and integer/fixed-point arithmetic must be used instead. This introduces new source of errors (quantization errors, overflows or saturations). To investigate the influence of these effects on the algorithm performance, a bit-true fixed-point simulation is used. In Simulink, this can be performed using the Fixed-point Blockset.

4. *Target architecture simulation* – in this phase, the implementation of the application in the target environment is simulated. This may include also optimization of the target architecture. The simulation environment will usually give an information on the real-time behaviour of our application.

5. *Testing in embedded target* – the final step, when the application is run in the real system, instead of the simulation environment.

### 1.2 Gap between algorithm design and implementation

Traditionally, the system engineer has been using the Matlab/Simulink environment and/or pieces of C/C++ code to verify the concepts and algorithm at the system level, while the hardware implementation was done using VHDL or Verilog languages. Hence, the algorithm had to be manually converted in order to be ported from the prototyping environment to the target system. This approach has several drawbacks:

- manual conversion from one language to another is tedious and error prone,
- once the conversion has been performed, the necessary changes are made in the version coded in the hardware-description language, while the original model quickly becomes obsolete,
- the tests that have been created to validate the original model have to be converted as well.

In the recent time, there have been multiple efforts to avoid this problem and to develop new design methodology using either automated transition from one testing environment to another or an integrated design system.

## 2 Linking the design environments

For the DSP-targeted development in Simulink, there are two basic tools in Simulink, the *Fixed-point blockset* and the *Real-Time Workshop*. The purpose of the first one is to add bit-true simulation capabilities to Simulink, while the second one is used to generate C-code from the Simulink models. Both these tools are targeted towards the DSP code development.

To provide link to the vendor-specific tools, there are two so called *embedded targets* – one for Motorola MPC555 microcontroller and one for the TI C6000 DSP.

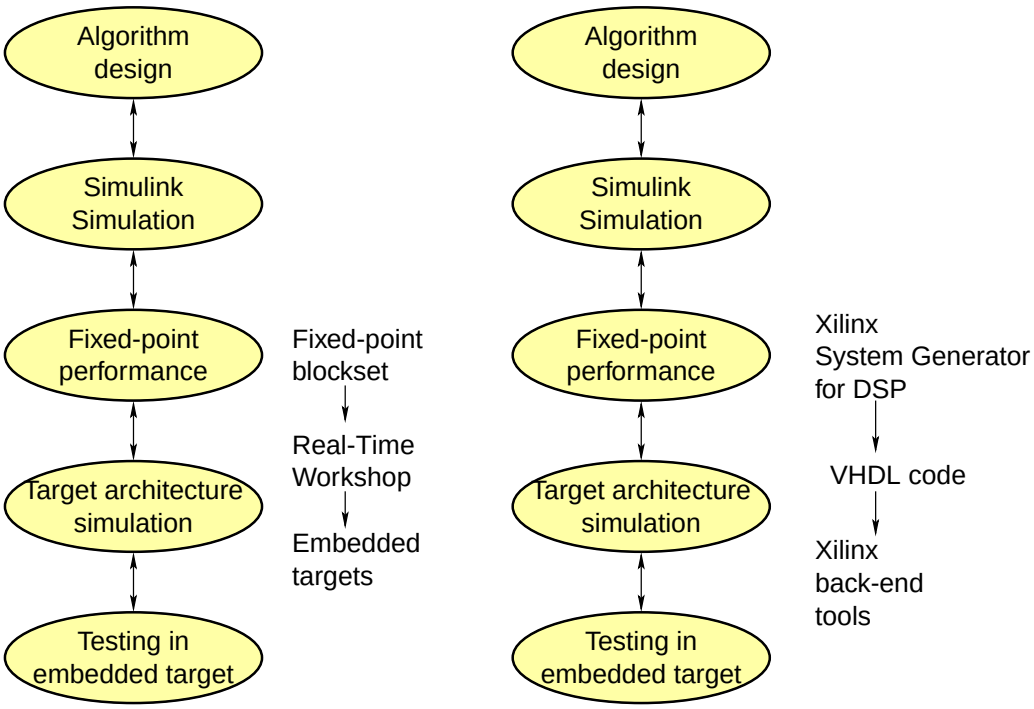
In relation with the FPGA-oriented development, the *Xilinx System Generator for DSP* can be mentioned – this tool addresses the fixed-point testing and partly the target architecture simulation. It also provides the translation to the VHDL hardware-description language.

The design flows for these tools are summarized in Figure 2.

## 3 System C simulation language

A possible alternative for the FPGA- and ASIC-oriented prototyping is the SystemC language.

It is a C++ class library that can be used to create bit-exact and cycle accurate models of algorithms, hardware architectures and system-level designs. The library provides constructs to model hardware timing, concurrency and reactive behaviour.




(a) DSP tools

(b) FPGA tools

Figure 2: Simulink extensions for the DSP and FPGA development

At the simulation level, the library can be used with the standard C++ development tools, while there also exist (proprietary) tools for hardware synthesis. Hence, it is possible to go from the algorithmic to the implementation level using one gradually refined description.

Since the library is open-source and not bound to one particular platform, it is gaining popularity among the developers. The development of the language is steered by the Open SystemC Initiative[1].

### 3.1 Features of the language

Let us show some of the basic features of the language on two simple code snippets – one showing the typical structure of a SystemC *module* (see Figure 4), second showing the structure of the *main* function and the "glue" code (see Figure 3).

The library contains constructs for creating modular hardware description. In the examples shown in Figure 3, we can see simple definitions of ports, description of module sensitivity to signals like clock, timeout or strobe signal, etc. The modules are connected using "wires" – signals. The `sc_trace()` command given in the first example is a wave-dump command; the `sc_start()` command is used to start simulation and to run it for a certain period. The simulation can be also initialized and single-stepped by separate commands.

## 4 Using Matlab as a testbench for SystemC

This is the simplest – case of linking Matlab with SystemC. In general, a testbench is used to provide stimuli to the tested design and to check results generated by the design.

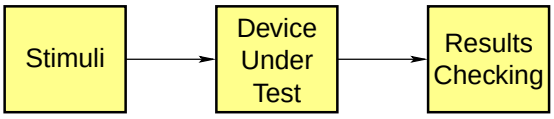
Matlab, with its rich set of signal generators and data visualization tools is ideal for both generating the test signals and for post-processing the simulation results. We shall show on a simple example that a SystemC-based simulation can be very easily plugged into Matlab in a form of a Mex file (Figure 5).

```
// Include signal and module definitions
#include "packet.h"
#include "transmit.h"
#include "receiver.h"

// Definition of main
int sc_main(int argc, char* argv[]) {
  // Definition of "wires" – signals
  sc_signal<packet_type> PACKET;
  sc_signal<bool> TIMEOUT, START;

  sc_clock CLOCK("clock", 20); // clock

  // Module instantiation ...
  transmit t1("transmit");
  // ...and wiring
  t1.timeout(TIMEOUT);
  t1.tpackout(PACKET);
  t1.start_timer(START);
  t1.clock(CLOCK);

  receiver r1("receiver");
  r1.rpackin(PACKET);
  r1.rclk(CLOCK);

  // Tracing:
  sc_trace(tf, PACKET1, "packet1");
  // Simulation length
  sc_start(10000);
  return(0);
}
```

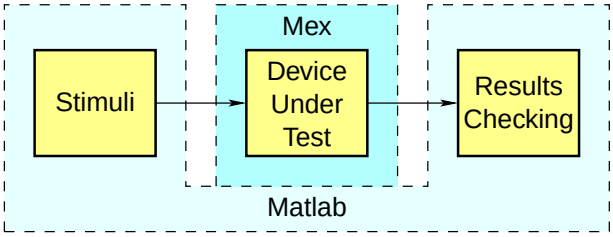
Figure 3: The structure of main

```
// Module: "container" for other definitions
SC_MODULE(module_name) {
  // Port declarations
  sc_in<bool> port1;              // input port
  sc_out<int> port2;              // output port
  sc_inout<bool> port3;           // input/output port
  // Local variables
  int framenum;
  // Method declarations
  void method1();
  // Constructor
  SC_CTOR(module_name) {
    SC_METHOD(method1);           // Method Process
    // Sensitivity to trigger signals, clock, ...
    sensitive_pos << clock;       // Sensitive to the pos. edge of timeout
  }
};
```

Figure 4: SystemC module description




General testbench environment




Testbench using Matlab and SystemC-based mex

Figure 5: Typical testbench environment

## 4.1 Replacing `sc_main` by the mex-function interface

The usual entry point for the SystemC-based simulation is the `sc_main()` function (see Figure 3). In order to plug the simulation as a mex-function into Matlab, this has to be replaced by the standard mex-header (let us recall that the mex-function uses the main() function contained in Matlab self). Looking into the SystemC source code, it was possible to find that `sc_main()` function is called from a standard `main()` function, as defined in the `sc_main.cpp` file (see code snippet in Figure 6). It is straightforward to replace this call by the mex header.

```
int main( int argc, char* argv[] )
{
  int status = 0;
  ...
  status = sc_main( argc, argv );
  ...
  return status;
}
```

Figure 6: `sc_main` utilisation in SystemC

## 4.2 Using the SystemC library in a mex-function

As described in the Matlab documentation, it is possible to link also a C++ based functions into Matlab/Simulink. In this case, the C++ code has to be wrapped by the following code in order to use C-style function calls:

```
extern ''C'' {
  ...
  code
  ...
  }
```

In the case of mex-functions, this wrapper is already defined in the `mex.h` header file; in the case of S-functions, we have to take care of wrapping.

In Figure 7, we give an example showing implementation of a sum of two 4-bit integers implemented using the data types from the SystemC library. The purpose of this example was only to verify that there will be no conflicts between mex and SystemC headers and libraries when compiling the code, and that the SystemC data type can be used in a mex-function.

The code in Figure 7 is the mex-function self. The function arguments from Matlab are assigned to the 4-bit signed integer variables `xd` and `yd`, and the result of sum is stored in a 4-bit variable `zd`. Hence, an overflow is possible and will be demonstrated in the encapsulating m-script.

The `compat.h` header file contains the following definitions, necessary for compatibility:

```
#define   i386
#undef    __GNUC__
#undef    uint64_to_double(u)
```

The code in Figure 8 represents a very simple test-bench. First, two vectors are defined, one containing a constant, the other containing one period of a sinusoid. Then, the mex-function is called in a loop, and finally, the input and output data are plotted. These plots are given in Figure 9.

```
#include <iostream.h>

#include "mex.h"
#include "matrix.h"
#include "compat.h"
#include "systemc.h"


sc_int<4> int_sum(sc_int<4> x, sc_int<4> y)
{
  return x+y;
};

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{

  double      *x, *y, *z;
  sc_int<4>   xd, yd, zd;

  x = (double *) mxGetPr(prhs[0]);
  y = (double *) mxGetPr(prhs[1]);

  plhs[0] = mxCreateScalarDouble(0);
  z = (double *) mxGetPr(plhs[0]);

  xd = *x;
  yd = *y;

  zd = int_sum(xd, yd);
  *z = zd;
  return;
}
```

Figure 7: Function `sc_sum.cpp`

# 5 Conclusions

We have discussed the possibility of using the SystemC library for bit-true simulations in Matlab. The viability of the approach has been demonstrated on a simple example.

```
x=4*ones(1,20);
y=5*sin((1:20)*2*pi/20);

for i=1:20
z(i)=sc_sum(x(i),y(i));
end

x_axis=1:20;

figure(1);
plot(x_axis,x,'b--',x_axis,y,'g-');
title('Input');

figure(2);
plot(z);
title('Output');
```

Figure 8: m-script testbench for the `sc_sum` function

In order to exploit the full strength of SystemC, many problems have yet to be considered: for example linking the SystemC into the S-functions, automatic generation of a synthesisable SystemC code from a Simulink model containing such functions (probably using the Real-Time workshop with appropriate TLC definitions).

# 6 References


[1] Available from the Open SystemC Initiative (URL: http://www.systemc.org).

[2] Xilinx Inc. 'Xilinx System Generator'
(Available online: http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=system_generator).


# 7 Acknowledgements


This work was supported by the Ministry of Education of the Czech Republic under Project LN00B096.


# 8 Contact


Department of Signal Processing
Institute of Information Theory and Automation
Academy of Sciences of the Czech Republic
Pod vodárenskou věží 4
182 08, Prague 8, Czech Republic
E-mail: schier@utia.cas.cz
URL: http://www.utia.cas.cz/ZS

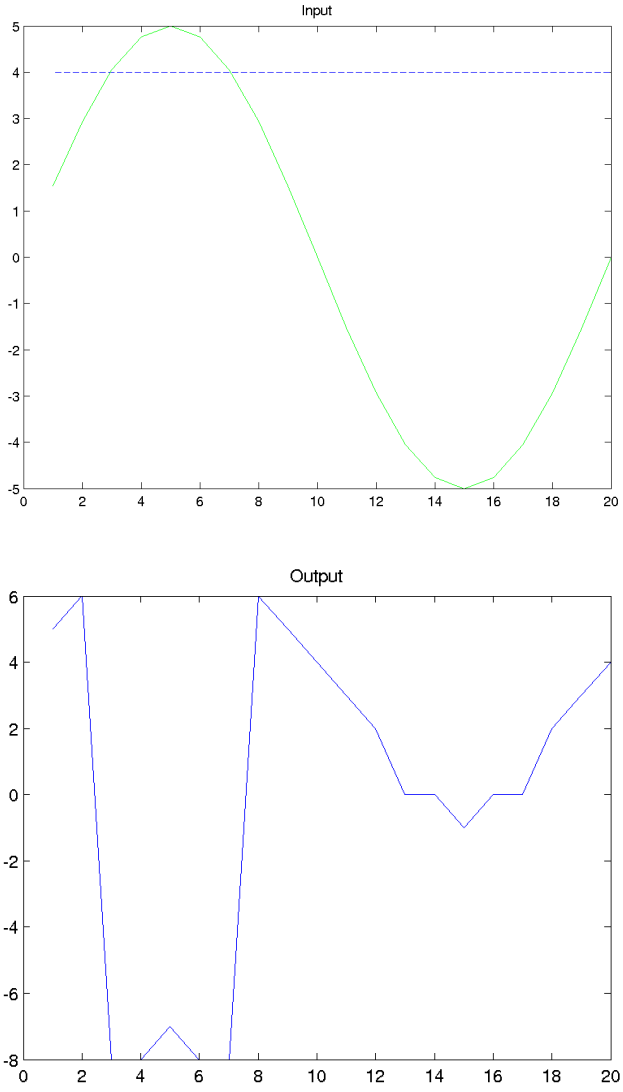



Figure 9: Input and output data