AN OPEN-SOURCE TOOL FOR SYSTEMC TO VERILOG AUTOMATIC TRANSLATION

J. CASTILLO, P. HUERTA and J. I. MARTÍNEZ

Universidad Rey Juan Carlos,
C/ Tulipan S/N, 28933, Mostoles, Spain,
{javier.castillo, pablo.huerta, joseignacio.martinez}@urjc.es,

***Abstract*— As the complexity of electronic systems increases, new ways for describing these systems are proposed. One actual trend involves the use of system level languages that allows the description of the whole system in a higher abstraction level. This type of methodology helps a designer to obtain an appropriate Hw-Sw partition, where the Sw is compiled to the target platform and the Hw is refined to bring it down to a lower level of abstraction in order to be synthesized. This last step usually requires the use of a translation tool that from a description of the system in a system level modeling language, converts it to an equivalent one in a standard Hardware Description Language, usually Verilog or VHDL. This works presents a tool that from a SystemC RTL description generates its equivalent Verilog code ready to be synthesized by any standard Verilog Synthesis Tool.**

***Keywords*— SystemC, Verilog, Translation.**

I. INTRODUCTION

The increasing complexity of the electronic systems has made necessary the exploration of new solutions in order to reduce its development time.

One of this solutions is to use new description languages (OSCI, 2002; Celoxica, 2005; Xilinx,2006; Pellerin and Thibault, 2002) which allow the designer to describe the system in a higher level of abstraction.

From this high level description of the system, a tool must provide a flow to reach the final silicon implementation. Following a traditional hardware-software codesign flow (Chiodo *et al*, 1994) (Fig.1), the system level description is profiled, and an appropriate hardware-software partition is proposed. Then the software has to be compiled to the targeted microprocessor. The hardware high level implementation can be directly converted into a working hardware using a proprietary synthesizer, like Handel-C or CatapultC. Another approach consist of rewriting it to obtain a lower and more detailed level of abstraction appropriate for hardware synthesis tools. This step is commonly made by hand with the problems it represents (time cost, prone to errors, etc.).

This work presents an open-source tool that taking as input a hardware module described using a high level description language, SystemC, gives as a result an equivalent description in Verilog. This Verilog description can be synthesized using any standard RT synthesis tools.

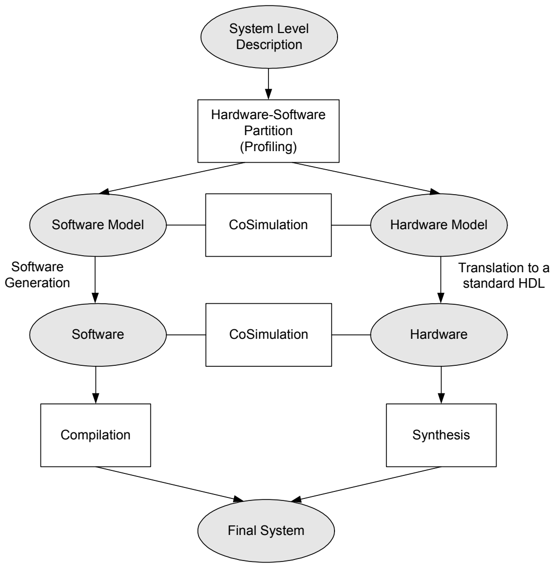

Figure 1. Traditional Hardware-Software CoDesign Flow

II. SYSTEMC

A. SystemC Overview

Nowadays hardware is usually described using HDLs such as VHDL and Verilog. However, software and system designers develop their code in C/C++. Hardware/software codesign becomes a very hard task due to the use of different languages at each abstraction level, or even in the same level (IP exchange). In this context, a single language that can be used in all the design stages is needed.

SystemC is a library of classes for C++ and a simulation kernel that provides all the features needed to describe a system in all its abstraction levels and a reference platform for IP exchange. SystemC also provides a library called the SystemC Verification Standard (SCV, 2002). This library provides classes and methods to build a verification methodology called Transaction Model Style (TLM) based on the use of transactors (Yuri *et al*, 2005). This verification methodology can be used with designs made with other HDLs. Many EDA vendors provide tools that allow SystemC to be mixed with different languages.

B. SystemC Design Flow

Estimations say that the main bottleneck of a traditional design flow is the verification stage (Semiconductor Industry Association, 2001). It is considered that at least 60% of the design effort is made in the verification stage. Traditional verification schemes (Fig. 2) also have the problem that the System Level Verification is the last stage of the design, extending the critical path and making architectural redesign almost impossible.

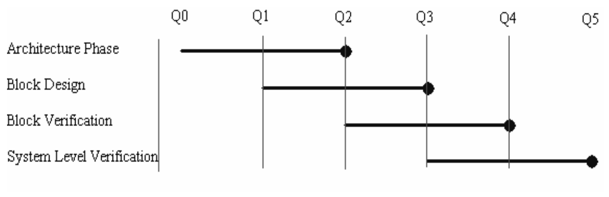

Figure 2. Traditional design flow

New solutions are proposed to overcome this problem. One is the Transaction Level Modeling Style (Fig. 3) described in the SystemC Verification Standard, that starts the verification effort begins in parallel with the system level design.

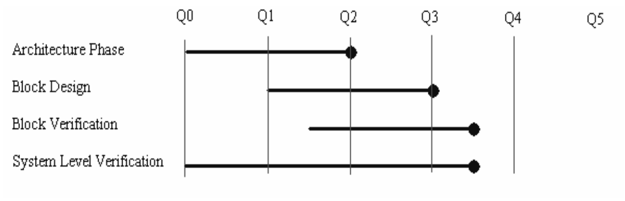

Figure 3. TLM design flow

The components developed in the system level description can be reused in the block verification step and in the final verification step, where all the blocks are replaced with their synthesizable models. Also, new architectural optimizations can be evaluated with low effort.

Using this methodology the verification effort can be reduced significantly (Castillo *et al*, 2004) but still there is a need for translating the SystemC hardware modules to a synthesizable HDL after the block design and verification stages. Even though there are in the market some tools that allow synthesizing SystemC, like Forte Cynthesizer (Forte, 2004) or Synopsys Cocentric (Synopsys), they are not free tools, which is especially relevant for educational environments.

For an Open Source Language like SystemC seems necessary to have the possibility of closing the design flow using free tools. In the following sections an Open Source Tool that produces a Verilog translation of the SystemC RT hardware modules will be described.

III. SYSTEMC TO VERILOG TRANSLATION

The first version of the translator was entirely developed using ANSI C. It was able of translating very simple designs written under very strict rules. But when more features were needed a new approach was necessary. The version 0.1 of the translator was entirely rewritten using Lex and Yacc (Levine *et al.,* 1992) parsing tools.

Lex and Yacc tools help to extract the structure of the SystemC files. Lex splits the source file into tokens. These tokens are collected by Yacc, which recognizes the tokens and executes the C code assigned to this token appearance.

The translator inputs are two files, called header and implementation that describe a module. Only one module can be declared in each header file.

The translation process is divided in two steps. In the first step the implementation file is parsed. In the second step the information from the first step is combined with the information obtained by parsing the header file to get the final Verilog translation.

A. Implementation File Parsing

In this first step the translator takes the module implementation file and uses the translatable RT subset defined in (Synopsys, 2002). The implementation file contains the implementation of the module and all the auxiliary functions required. The tool converts all the SystemC code and functions inside the processes into Verilog. The local variables inside the modules and their references are renamed, adding to the names the name of the process in which they are declared. This is because Verilog doesn't support local variables insides the process, whilst in SystemC is not unusual to have different processes using local variables with the same name, for instance, `aux`

Another task that is carried out in this first step is to convert the structures data type into something Verilog can understand because it is a data type not supported. Structures are translated into local variables changing the name of each element of the structure for the same name plus the structure's own name. All the references to the structures inside the process code are also changed to the new names.

Translating SystemC assignments to Verilog can be difficult because SystemC can not distinguish between signal blocking and non-blocking assignments. In SystemC the only way to assign a value to a sc_signal is to use the .write() method of the sc_signal class. The first thing to analyze is that if this method produces a blocking or a non-blocking assignment to the signal. A simple test to understand the behavior of the .write()method was designed:

```
sc_signal < sc_uint < 8 > > a,b,c;

void test::test ()
{
if (reset.read ()){
      a.write(1);
      b.write(2);
      c.write(3);
  }else{
      a.write(c.read());
      b.write(a.read());
  }}
```

Running this simple piece of code shows that a is 3 and b is 1 after the first cycle signal. That means that the .write() method has a non-blocking behavior and all assignments to a sc_signal in the Verilog translation should use the non-blocking assignment operator (<=).

Another test was necessary to check the behavior of the assignments to local variables:

```
void
test::test ()
{
 sc_uint<8> a,b,c;

 a=1;
 b=2;
 c=3;

 a=c;
 b=a;
}
```

As expected, the result was a is 3 and b is 3, therefore, the variable assignment states as a Verilog blocking assignment (=).

The module implementation file could also contain a set of directives for the translator. These directives allow to set up and tune the behavior of the translator, indicating which sections of the code should or shouldn't be translated (translate on/off) or specifying code that has to be copied without translation into the Verilog file (verilog begin/end).

Apart from translating processes and functions, this first step also stores some information needed in the final assembly step. This information concerns the signals that have been written inside the processes and will be used later on to decide whether a signal is a reg or a wire in the Verilog translation.

The output of this step is a set of files with the following information:

- *name*.sc2v and *name*_reg.sc2v: the Verilog equivalent code and registers declaration of the processes and functions of the module
- file_defines.sc2v: the C preprocessor calls inside the module
- file_writes.sc2v: the signals written inside the module

B. Header File Parsing

The second step of the translation takes the header file of the module and the information generated by the first step as input and generates the Verilog file.

The translator second step reads all the information of the header file first using another Yacc parser and stores it into linked lists data structures. Each of these lists can include other lists. For example, the translator has a list for all the processes of one module, where each entry of that list includes information of its name, type (sequential or combinational) and another list with the signals in the sensibility list of the process.

There is a list for each of the following objects:

- Module Ports
- Module Signals
- Processes
    - Sensitivity List of the Process
- Instantiated Modules
    - Port Bindings
- Enumerated Types
    - Enumerated members
- Auxiliary Functions
    - Inputs and outputs for each function

When the parser has finished collecting information, the generation of the Verilog file can start.

After writing the module name, all the ports with its associated types have to be written. The translator has to decide for each output if it has an associated register or not. If the output is directly connected to an instance, the output should not have an associated register. If not, this output will be written to from a process and should have the associated register. This is very easy to analyze just looking at the instances list and the bindings of each instance.

After the ports are declared the enumerated data types have to be translated. This feature was incorporated at the request of many users and has been included in version 0.2 to ease the design of state machines. Verilog doesn't allow enumerated data types but as in the case of structures it is not difficult to mimic its behavior, using parameters in this case. For example the SystemC enumerated variable:

```
    enum {S0,S1,S2,S3} state;
```

is translated into:

```
    parameter S3=0, S2=1, S1=2, S0=3;
    reg [1:0] state;
```

The translator automatically calculates the register length to fit the number of elements of the enumerated data type. It is necessary to be careful with enumerated types because using the same element name in more than one enumerated type will produce an error.

The signals used by the module are declared next. It is compulsory to decide if the SystemC *sc_signal* should be translated into a Verilog *wire* or *reg* signal type. The way of doing this is to look at the file_writes.sc2v file generated during the implementation file parsing. If the signal has been written inside any process, the signals will be translated into a *reg* type. If not, the signal was declared to be connected to an instance and it is translated into a *wire.*

After finishing with all the signals and ports, the next step is to write the hierarchy instantiations. This step requires to read the list with all the information about the instances and its bindings and write the equivalent code to the Verilog file.

Next, the file_defines.sc2v which has all the preprocessor directives (defines, conditional compilation directives and macros) is read and written to the translation.

Finally the functions and processes code are written down in the translation from the files generated during the implementation file parsing, concluding with a Verilog equivalent file ready to be simulated and/or synthesized.

IV. DESIGN OF AN SYSTEMC AES-DES CRIPTOPROCESSOR USING SC2V

This section describes the complete design flow of an AES/DES crypto processor, from the System Level specification of the design, to the final implementation on a prototype board using the SystemC to Verilog translation tool.

A. System Design

The first step of the design process is to develop a functional model which is a behavioral description of the system. This model has no time and no implementation details about the final system. It only reflects the required functionality of the whole system, and it will be used in later verification stages as the golden model for the designed blocks. This kind of model is called an "Untimed Functional Model" (UTF) for the reasons explained before.

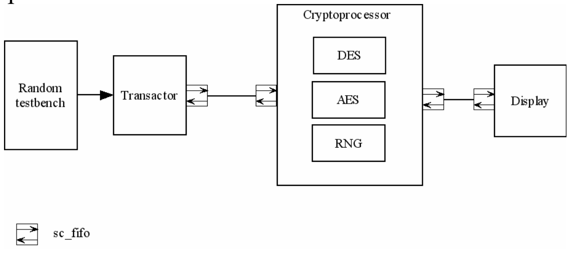

Figure 4. Functional model

One of the most important parts of the verification methodology is the testbench generation. Using SCV features, a testbench that generates random keys and data for the AES/DES models was designed. The transactor takes the stimuli generated by the random testbench and applies them to the model. If the abstraction level of the model is changed, the same testbench can still be used by simply changing the transactor.

At this stage the cryptoproccesor model is a set of C++ functions with a SystemC wrapper and sc_fifo channels to connect them to the testbenches and the display.

Before the module design phase can begin, it is necessary to go down in the abstraction level. In this level, information about the interfaces of the modules is added, as well as a clock. In this case no accurate time information is added to the model at this stage, because no time specifications exist.

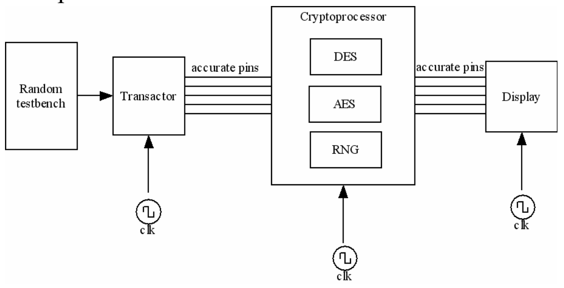

Figure 5. Pin accurate model

After the interfaces and time is added to the model the blocks that compose the systems are described using the SystemC language Synthesizable subset. To have a synthesizable model of the system it was necessary to describe the following blocks:

- Bus interface
- Controller of AES/DES module
- Random number generator
- DES encryption/decryption module
- AES encryption/decryption module

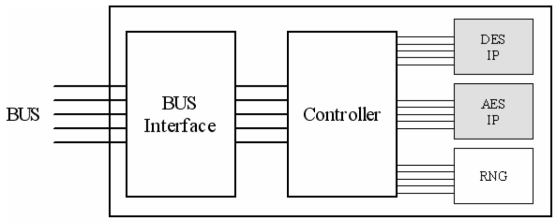

Figure 6. Cryptoprocessor modules

The bus interface depends on the system bus, it connects the crypto-processor to a master. In this case, the selected bus is a Wishbone compatible one.

The controller of AES/DES modules takes the configuration word of the cryptoprocessor and generates the signals to manage the AES/DES modules. It also takes the data and the keys from the data registers and applies them to the modules, writing back the ciphered block in the output registers.

The random number generator is based on the scheme (Tkacik, 2002) below, where an LFSR and a CASR in parallel are used to generate a random number generator with good statistical properties and a cycle length of 2^80. It is important to notice that the seed of the random number generator can be changed writing in the data register of the random generator.

B. System Verification

To guarantee the IP quality, a complete verification environment must be developed.

Three verification levels are proposed:

- Block Level Verification
- Module Level Verification
- System Level Verification

A block is a component of the system that must be verified before being integrated in a module. An example of block could be the key generation block of the DES and AES modules. In order to verify these blocks, classic signal-oriented testbenches were applied.

Another verification level is the module one. In this level the modules that compose the cryptoprocessor are verified using a classic testbench as in the block level, and also a random verification in the case of DES and AES blocks. This module random verification is very similar to the one used in the System level verification. In both cases the random testbench applies stimuli to the RT model to be verified and to the C code used as a golden model.

The outputs of both modules are passed to the checker that compares them. If a mismatch between the data is found, an error is reported and the simulation ends. The test was executed with several different seeds during long periods of time.

In the System Level case the testbench developed for the System Level specification is reused by simply changing the transactor functionality. At this level, the transactor applies the stimuli to the RT synthesizable design and to the C++ model of the cryptoprocessor used as a golden model.

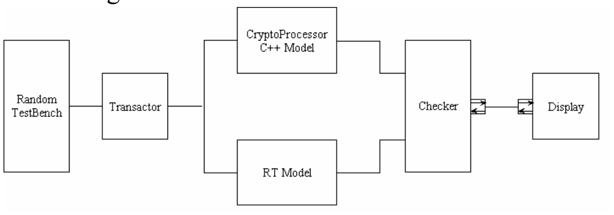

Figure 7. System Level Verification environment

B. Translation and On-Board Verification

After the design is validated using the simulation environment, the design is translated to Verilog using the translator. Now the design can be synthesized by any standard tool, for example, Xilinx XST.

In Table 1, the number of SystemC lines of each module that compose the system is presented. These data offers an idea about the crypto-processor complexity.

Table 1. Number of lines of translated cores

Core	SystemC Lines
AES-128	2638
DES	4225
RNG	303

The advantages of using a design methodology based in the use of Transaction Level Modeling Style were described earlier. One of the main advantages shown was that the verification environment could be reused in other stages by only changing the transactor functionality. In this work this concept is extended in order to verify the functionality of the physical implementation over a development board.

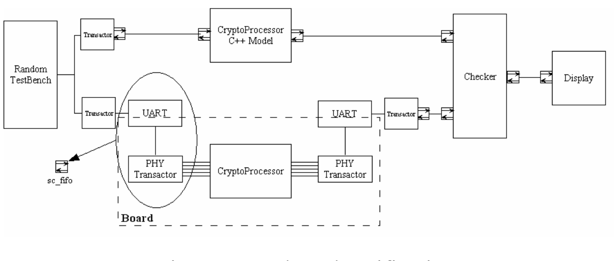

Figure 8. On board verification

The Physical Transactor concept is the main fact introduced in this level of verification.

This kind of transactor converts the data from the UART on the board to the physical signals applied to the cryptoprocessor ports. Another Physical Transactor takes the outputs and send them back to the verification environment through the UART.

This transactor in combination with the UART works as a *sc_fifo* channel that blocks the simulation until a data from the board arrives. This kind of model is equivalent to an UTF model, where the *sc_fifo* channels connected to the physical implementation are exchanged by their equivalent models, made up of the UART and the physical transactor.

As shown in the System Level verification stage, the C++ model of the cryptoprocessor is connected in parallel with the design under verification. The outputs generated by the board are sent back to the verification environment via the UART on the board and compared with the ones generated by the C++ golden model by the checker.

The system was downloaded to a Celoxica RC203 board with a VirtexII FPGA and tested successfully.

V. CONCLUSIONS AND FUTURE WORK

A tool that allows automatic translation from SystemC RT to a Verilog equivalent description has been presented. The tool provides an easy path from a SystemC description to a more appropriate HDL that can be synthesized using any standard RT synthesis tools. This allows closing the design flow using Open Source tools.

A complete design using SystemC design flow which has been successfully implemented on a board using the translator was also presented.

The tool is in a continuous improving process adding new features and correcting bugs reported by the people is using the tool around the world. Some of the new features which would be included in the next releases would be:

- Improved support for C preprocessor macros
- Support for C++ constructions:
    - Templates
    - Classes
- Improve support for structs:
    - Arrays inside structs
    - Nested structs
- Add up users requests!

The tool can be downloaded for free from: http://www.opencores.org/projects.cgi/web/sc2v/overview.

This work has been supported by the Spanish PROFIT and MEDEA+ program under FIT-0700000-2003-930 contract.

REFERENCES

Castillo, J., P. Huerta and J.I. Martinez, “SystemC Design Flow for a DES/AES CryptoProcessor”, *In WSEAS Transactions on Information Science and Applications,* Athens, **1**, 193-198 (2004).

Celoxica, “Handel-C Language Reference Manual”, (2005)

Chiodo, M., P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno and A. Sangiovanni-Vincentelli, “Hardware-Software Codesign of Embedded Systems”. *In IEEE Micro*, 26-36, August (1994)

Forte, “Cynthesizer Datasheet” http://www.forteds.com/products/cynthesizer_datasheet.pdf (2004)

Levine, J., T. Mason and D. Brown, “Lex & Yacc, Second Edition”, Oreilly (1992)

OSCI, “SystemC 2.1”, http://www.systemc.org, (2002)

Pellerin, D. and S. Thibault, “Practical FPGA Programming in C”, Prentice-Hall (2002)

SCV, “SystemC Verification Standard Specification” (2002)

Semiconductor Industry Association, “International Technology Roadmap for Semiconductors”, http://public.itrs.net/Files/2001ITRS/home.htm, (2001)

Synopsys, “Cocentric System Studio Datasheet”, http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html (2006)

Synopsys, “Describing Synthesizable RTL in SystemC” (2002)

Tkacik, T., “A hardware random generator”. *In Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002,* LNCS 2523, 450-452, USA (2002)

Xilinx, “Xilinx System Generator for v.8.1 User Guide” (2006)

Yuri, E., J. Tatsuda, N. Khan and C. Dietrich, “Transaction- based simulation using SystemC/SCV”, http://www.eetasia.com/ (2005)

Received: April 14, 2006.
Accepted: September 8, 2006.
Recommended by Special Issue Editors Hilda Larrondo, Gustavo Sutter.