Automated SystemC to VHDL Translation in Hardware/Software Codesign

Christian Côté
Industrial/Consumer Division
Teradyne Inc.
Boston, MA, USA
Email : Christian.Cote@teradyne.com

Zeljko Zilic
Dept. of Electrical and Computer Engineering
McGill University
Montréal, QC, Canada
Email: zeljko@macs.ece.mcgill.ca

ABSTRACT

Recent advances in electronic circuit technology have enabled the creation of "system-on-chip", which comprise both hardware and software components. Codesign takes advantage of this opportunity by considering both components as a whole throughout the design process. While an increasing number of tools are being offered, most concentrate on the simulation of systems, with little or no support for their implementation. This paper describes a translation algorithm developed to act as a bridge between simulation and implementation by translating SystemC code to VHDL.

1. INTRODUCTION

In recent years, the hardware/software dichotomy has been seen as a limiting factor in the effectiveness of a design. The codesign approach to system design was introduced to overcome this limitation. By considering both hardware and software jointly throughout the design cycle, it is possible to better simulate and validate the whole system and save valuable time by limiting duplication in the design process.

One of the key aspects of the codesign approach is the ability to describe both hardware and software components using a single system behavior description. Since software has traditionally been used to model and simulate system components, a number of existing codesign tools employ a variant of C/C++ to serve this purpose. Such an approach, however, leads to certain problems, especially in system implementation. Hardware components often have to be rewritten by a time consuming and error-prone process.

This paper documents our attempts at automating the software-to-hardware translation process in a codesign context. First, we will motivate the need for such a translation algorithm. We will then present our architecture for a translation process between SystemC and VHDL. The translation algorithm will then be described, along with some experimental results.

2. BACKGROUND

The aim of codesign is to bridge the gap between hardware and software by combining both aspects early on in the design process. The idea of integrating multiple aspects of the design process in a uniform framework is not new. *Concurrent engineering* is an approach that, among others, promotes increased exchange of information between various design groups. Codesign can be seen as a refinement of this concept. [7] The central ideas of codesign can be summarized as follows:

- *System-level description independent of hardware or software concerns, ideally through a unique language.*
- *Partitioning between hardware and software must be performed as late as possible in the design process*
- *Communication between blocks must be abstracted enough to be independent of the actual implementation*
- *Co-simulation of the system as a whole should be possible throughout the design process*

The issue of communication is of major importance in codesign. Indeed, most systems are composed of a number of functional units exchanging information through internal communication mediums. To preserve design flexibility and implementation independence, it is common to use a layered approach to communication synthesis [2].

Co-simulation is one of the great strengths of a codesign approach. Typically, system-level simulation must be performed using high-level models of the components. This of course introduces a huge overhead for creating and maintaining the models, and a certain uncertainty on their accuracy. Given the right tools, it should be possible to perform co-simulation at all steps prior to implementation.

2.1. Hardware/Software Description Language

The use of a single uniform language in the codesign of a system allows designers to produce a system-level description and simulate it within a single framework. Design exploration can also be facilitated by allowing functional units to be quickly mapped to a hardware or software structure for simulation purposes, which can facilitate the choice of the optimal partition

A number of approaches can be taken to achieve these objectives. First, it is possible to invent a completely new language specifically for the purpose of codesign. Such a language would be better suited for codesign tasks, and be more efficient. However, it would also suffer from a steeper learning curve and a lack of supported tools.

The second alternative consists in extending an existing description language, either hardware or software, by adding the required constructs for codesign. While this provides a much more stable and familiar base

0-7803-7596-3/02/$17.00 ©2002 IEEE

for the designer to work with, it also yields an imperfect solution. For instance, extending a hardware description language would make system-level simulation impractical. Conversely, extending software languages to represent hardware constructs would limit the value of the simulation, especially with respect to timing.

The debate as to which approach, and which language, is the most appropriate for the codesign of systems is ongoing. [11], [14] However, a number of tools have emerged recently, which use variants of programming languages, and especially C/C++, as codesign languages. Still, they are not fully established as a standard, and special system design languages, while not covered here, still retain their value.

2.2. SystemC

SystemC [12] is a set of C++ classes that are used to model hardware behavior. Its syntax is a mix between software (C++) and hardware (VHDL). The use of C++ classes means that hardware and software components can be developed and simulated within a single uniform framework. Since no proprietary extensions were made to the language, the whole system can be compiled using any generic C++ compiler. Further, it is an open standard developed by a number of EDA vendors, and is supported by an increasing number of codesign tools.

Although SystemC makes it very easy for the designer to codesign and co-simulate a system, even after the hardware/software partition has been determined, it still does not allow the hardware components to be easily implemented. In particular, it does not allow the automatic generation of synthesizable HDL code. However, since SystemC was designed as a full-fledged HDL, it is worthwhile to consider translating a SystemC description into a VHDL one.

2.3. Other Related Work

A number of solutions have been presented that attempt to bridge the implementation gap in a codesign context. Celoxica DK1 [3] and [13] seek to translate software programs, written in C, to hardware. Other projects have focused their efforts on developing object-oriented hardware synthesis methodologies. [10], for instance, uses the '*e*' language for the verification and synthesis of object-oriented hardware, while [8] uses SystemC. Finally, SystemC compilers, such as Synopsys' CoCentric [16] or [6], have been developed to allow the use of SystemC as an HDL.

Most of these projects use compilation techniques to generate hardware synthesis code. This offers the advantage of producing more efficient hardware. However, the relationship between constructs in the source code and the compiled result is hard to establish, which significantly reduces the usability of the output code for debugging and optimization. The approach we propose uses a transparent translation algorithm where the relationship between input and output is easily established. The designer can therefore go back and forth between hardware and software with no adaptation.

3. SYSTEMC TO VHDL TRANSLATION

Compilers that allow the translation from software to hardware already exist. However, they either impose constraints on the input code, or produce unreadable output. Our goal is to develop an algorithm that will translate SystemC code into synthesizable VHDL, while limiting the number of constraints forced on the developer. This should be facilitated by the fact that a number of SystemC constructs can be mapped directly to VHDL.

The translation algorithm we propose to develop is not intended at producing highly optimized code. Indeed, this could only be achieved using more aggressive compilation techniques, and behavioral synthesis approaches in particular. The algorithm presented here is therefore expected to be mostly useful for design exploration, and for quick and dirty prototyping. Further, producing readable code allows the designer to quickly go back and optimize the code, or at least to be able to identify where the bottlenecks are.

3.1. Design Constraints

The most important requirement for the design of our translator should be to preserve the semantics of the SystemC design. Indeed, with Celoxica's DK1, certain optimization or modifications resulted in a change of the meaning of a given construct, which could cause the system to misbehave. Therefore, we need to ensure that the translation maps closely with the original version, even at the expense of inefficiencies.

If our translator is to have any useful application, we must limit ourselves to producing VHDL code that is synthesizable. Indeed, the VHDL language contains a number of software-like constructs that are not supported by all synthesis tools. For example, indefinite loops, wait statements or pointers can be difficult to represent in hardware. We will need to find alternate mappings for these constructs that can easily be synthesized.

Finally, our translator should be able to produce readable output code. Indeed, a designer should be able to easily identify how each construct got translated, and to work directly with the output code. This can be achieved by preserving symbol names and comments, and by naming automatically generated variables such that their origin can be easily traced. While this is mostly an implementation constraint, it does limit the amount of modifications that can be made in the translation process.

3.2. Construct Mapping

Since a number of constructs cannot be directly mapped from software to hardware, we must find a synthesizable representation that is semantically equivalent. However, it is not possible to achieve this goal for all the problematic constructs. For example, pointers, functions or global variables can be somewhat difficult to represent in hardware, and are not supported by our translator. Table 1 presents a list of the differing SystemC constructs and their mapping in VHDL.

Table 1: Mapping of SystemC constructs to VHDL

System C Construct	VHDL Translation
Structure elements (classes, methods, ports, signals, etc.)	Direct mapping (entity, process, ports, signals, ...)
Data types	Direct mapping (except float)
Bit-wise, logical and arithmetic expressions	Direct mapping
Conditional execution (if else, switch)	Direct mapping (If Then Else, Case)
Loops and waits	**Finite State Machine Model**
Functions, pointers, global variables...	Not supported

3.3. Finite State Machine Model

Of particular interest in the above table is the method for the translation of loops and wait statements, which are perfect examples of widely used SystemC constructs that have non-synthesizable VHDL equivalent. Indeed, both are highly sequential in nature, and thus are closer to a computer than to a hardware implementation. We must therefore serialize their execution in hardware to achieve our goal. One way to serialize execution in VHDL is to use a finite state machine (FSM). FSMs effectively allow the designer to break down complex operation in time. This method can thus be used to represent sequential execution constructs like loops and waits. The following code fragments shows the mapping of a SystemC loop:

SystemC

```
expression_block1;
while(condition) {
    loop_body;
}
expression_block2;
```

VHDL

```
case state_var is
  when STATE1 =>
    expression_block1;
    if condition then
      state_var <= STATE2;
    else
      state_var <= STATE3;
    end if;
  when STATE2 =>
    loop_body;
    if condition then
      state_var <= STATE2;
    else
      state_var <= STATE3;
    end if;
  when STATE3 =>
    expression_block2;
    [...]
end case;
```

As can be seen, both fragments are semantically equivalent (assuming *state_var* is initialized to *STATE1*). The loop condition is used to control state transitions, and each loop iteration is assumed to takes one VHDL clock cycle to complete. It is possible to generalize this template to represent multiple loops or nested loop combination. Wait statements can be handled in a similar fashion, except that state transitions have no condition.

This approach, while simple and efficient, does suffer from certain limitations. First, it is only valid for synchronous systems. While a clock signal is present in most designs, it is not true in all cases. Thus, modules that can be translated using this approach must be present in a system where a clock signal is available.

Also, one can note that the timing is not preserved in the translation. Indeed, where the SystemC version would execute in a single clock cycle, the VHDL one would require an additional number of clocks dependant on the number of times the loop is executed. This can be significant, especially if the module being translated must communicate with another one in a timely fashion.

This problem can be addressed by adding wait statements in the SystemC description, both before the loop and at the end of the loop body. These modifications would need to be made by the designer, and thus can be seen as restriction on the formatting of the SystemC code. Both alternatives are semantically equivalent, and should not affect the translation.

Another important limitation of this approach lies in the handling of complex nested structures that comprise both conditional and iterative execution. While the computer can translate such structuresto an FSM easily, they quickly become intractable for a human reader. More importantly, the ability of a synthesizer to correctly map the given structure is degraded as the complexity increases. Finally, the efficiency of such a system is significantly decreased as the number of cycles required to complete the execution increases.

A final limitation of this approach is that it is quite inefficient for handling *for* loops that are counting over signal indices. Such structures can be easily parallelized by unrolling the loop, increasing efficiency. However, to achieve this our translator would need to detect which loops can be unrolled and which can't. In general, it would need to verify if any given signal is assigned to in more than one loop iteration, and unroll it if not.

4. TRANSLATION ALGORITHM

4.1. Parsing

The first step in the translation process consists in reading the SystemC description to memory in a hierarchical list. This is necessary since translation decisions for any given expression may depend on structures encountered later on. A line-by-line translation would be impossible to implement. If loops are detected, a flag is set to indicate that the FSM model must be used.

4.2. FSM transform

This step is where SystemC expressions are reorganized to form the FSM model. This is achieved in multiple stages. In a first stage, the FSM tree is created and states are assigned to different blocks based on their sequential order. Next, state transitions are determined and written in the proper state body. Special care must be taken when determining transitions for nested loops or waits to ensure that the semantics are preserved. For example, a loop that is part of the body of a conditional statement should have its condition evaluated before the loop body is processed.

The next step consists in determining the value of the output ports for each state. Output ports must be driven at all time, and the division into states can have altered this behavior. By storing a local value of the output signals in a register, and by driving the output ports from it, we can avoid potential problems. The register is updated only where the original program assigned to the output port.

4.3. Processing

Before the actual translation can take place, complex expressions must be broken down to make them easier to synthesize. This is easily achieved by defining a number of intermediate signals or variables needed to obtain a value. Operations on multiple literals can also be simplified by computing their values ahead of time. Signals that are assigned more than once in a single clock cycle must also be broken down to meet synthesis requirements. Dependencies must be checked to make sure that the right values are used to generate intermediate signals.

5. EXPERIMENTAL RESULTS

We have used this algorithm to convert simple floating-point adders and multipliers, implemented in SystemC, to VHDL. They both take as input the sign, mantissa and exponent of two IEEE standard floating-point numbers, and output the result. The algorithms were implemented sequentially, and no efforts were made to exploit parallelism. The translation to VHDL produced a five state FSM for the adder and a three stage FSM for the multiplier. The VHDL code was compiled using Altera's *Max+plus II* tool. The results obtained are compared to the *DFPADD* [4] and *DFPMUL* [5] commercial IP cores from Digital Core Design. All versions were synthesized for the Flex10KE-1 FPGA from Altera.

Table 2: Translated SystemC code vs. commercial IP cores

	Performance	Logic Cells
Translated adder	39 MHz	572
DFPADD	42 MHz	1161
Translated multiplier	17 MHz*	1995
DFPMUL	33 MHz	2609

The performance obtained from a translated adder is quite close to the commercial result. However, we should note that because the translated code resulted in a five state FSM, at least five clock cycles are required to compute a sum. Also, some states represent loops bodies, and are traversed more than once. The actual performance can therefore be expected to be 10 to 20 times lower than reported. In contrast, the commercial core comprises a four stage pipeline, which ensures a higher throughput.

On the other hand, the performance of the translated multiplier is roughly half that of the equivalent commercial core. This performance hit is caused mainly by the use of a single-cycle multiplier with a 60ns hold time. In fact, the registered performance reported by the timing tool is of 76 MHz, when not accounting for the hold time for the multiplier. Since the translated VHDL code is readable, it would be feasible to optimize it.

As for the area required for each implementation, we can note that while the commercial cores are much more feature-rich than our simple adder and multiplier, the difference in area is still larger than expected. Although we do not claim that the translated code is an optimal solution, we can still assume that the translation does not affect the size as much as it did the performance.

6. CONCLUSIONS AND FUTURE WORK

We have shown that it is possible to design a simple translator that supports common SystemC constructs. While most have direct VHDL equivalent, some, like loops and wait statements, must be given special handling. We have demonstrated that by converting the module to a finite state machine structure, it is possible to obtain an easily synthesizable, semantically equivalent VHDL representation.

The architecture we have presented can support most SystemC description. However, the results obtained are far from optimal. Dependency checks, for instance, could allow internal parallelisms to be exploited. Likewise, complex expressions could be optimized by the common factor extraction. Finally, methods to map unsupported features, such as pointers, functions and global variables, could be developed to further extend the program's functionality. Note that such changes would reduce the readability of the output code, which is our project's main goal.

7. REFERENCES

[1] P. J. Ashenden, *The Designer's guide to VHDL, 2nd edition*, Morgan Kaufmann Publishers, USA, 2001.

[2] I. Bolsens, H. J. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest, "Hardware/software co-design of digital telecommunication systems", *Proc. IEEE*, Vol. 85 Iss. 3, pp. 391 – 418, 1997.

[3] Celoxica Limited, http://www.celoxica.com

[4] Digital Core Design DFPADD datasheet, http://www.dcd.com.pl/dcdpdf/dfpadd_ds.pdf

[5] Digital Core Design DFPMUL datasheet, http://www.dcd.com.pl/dcdpdf/dfpmul_ds.pdf

[6] G. Economakos, P. Oikonomakos, I. Panagopoulos, I. Poulakis, and G. Papakonstantinou, "Behavioral synthesis with SystemC", *Proc. DATE*, pp. 21 – 25, 2001.

[7] D. W. Franke, and M. K. Purvis, "Hardware/software codesign: a perspective", *Proc. Int. Conference on Software Engineering*, pp. 344 – 352, 1991.

[8] E. Grimpel, and F. Oppenheimer, "Object-oriented high level synthesis based on SystemC" *Proc. ICECS*, pp. 529 -535, 2001.

[9] A. Jerraya, "Hardware-software codesign", *IEEE Design & Test of Computers*, Vol. 17 Iss. 1, pp. 92 - 99, 2000

[10] T. Kuhn, T. Oppold, C. Schulz-Key, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai, "Object Oriented Hardware Synthesis and Verification", *Proc. Int. Symp. System Synthesis*, pp. 189 – 194, 2001.

[11] G. Moretti, "Get a handle on design languages", *EDN Magazine*, June 5th, 2000, pp. 60 – 72, 2000.

[12] Open SystemC Initiative, www.systemc.org,

[13] S. Sankaran, R. L. Haggard, "A convenient methodology for efficient translation of C to VHDL", *Proc. Southeast Symposium on System Theory*, pp. 203 – 207, 2001.

[14] P. Schaumont, I. Verbauwhede, R. Chandra, K. Konigsfeld, D. Gajski, G. Berry, D. Dumlugol, "The next HDL: if C++ is the answer, what is the question?", *Proc. Design Automation Conference*, pp. 71 – 72, 2001.

[15] H.-J. Schlebusch, , "SystemC based hardware synthesis becomes reality", *Proc. Euromicro*, p. 434 vol.1, 2000.

[16] Synopsys Inc. CoCentric SystemC Compiler Datasheet, http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC_ds.html