Construction of VHDL-AMS simulator in Matlab™

M. Zorzi, F. Franzè, N. Speciale

DEIS, University of Bologna
Viale Risorgimento 2 40136 Bologna, Italy
e-mail: mzorzi@deis.unibo.it

Abstract

In this paper we describe the digital kernel implementation of the simulator S.A.M.S.A.[1], a tool for the simulation of Vhdl-Ams systems in Matlab™. The digital kernel was validated by simulating different systems. In particular we will show the simulation of a low-power multistage decimator for a wideband Δ-Σ Analog to Digital Converter (ADC). This example was correctly simulated given the same results as other VHDL commercial tools.

Keywords: VHDL-AMS, CAD tool, Circuit simulation, Δ-Σ ADC, Behavioral Modeling

1 Introduction

System validation by using the circuit simulation is mandatory to manage the increasing design complexity. New CAD tools, based on analog and mixed-signal hardware description languages and simulation [2][3] provide a unifying tool to link the various design steps. Moreover, the use of behavioral languages like *Vhdl-Ams* gives to the designer a degree of freedom not available by using a circuit simulator like Spice [4]: it is possible to define a new model or a system not present in the simulator model library. Although adding new models to the simulator can be accomplished also in Spice [5][6], the use of behavioral languages developed on recent years represents a more appealing solution, and allows to describe systems containing a mix of analog/digital blocks described at different levels and in different physical domains.

In this work we present the digital engine implemented in *S.A.M.S.A.*, a Matlab™ [7] tool for *Vhdl-Ams* simulation.

This work is organized as follows: in the next Section we explain how the *Vhdl-Ams* is compiled and mapped into a C++ intermediate form. Section 3 describes the implemented digital simulation algorithm. Section 4 describes a validation example: a low-power multistage decimator for a wideband Δ-Σ Analog to Digital Converter (ADC). Finally, conclusions will be drawn in Section 5.

2 *Vhdl-Ams* language compilation

The compilation is a two step procedure: the *Vhdl-Ams* code is converted into C++ code and then compiled with a standard C++ compiler like GNU gcc. The final output is a set of dynamically linked library which will be used by the simulation engine. C++ thus acts like an intermediate language, which is anyway fully human-readable and provides an object oriented approach that is more effective to translate *Vhdl-Ams* syntax. We intend to show how it is possible to find a natural mapping between a large set of *Vhdl-Ams* elements and C++. We focus now on the most important elements of *Vhdl-Ams* subdividing them into some independent sets: *i*) the type system; *ii*) statements and expressions; *iii*) entity and architectures.

The mapping for statements and expressions is quite simple and trivial, while finding a mapping for the type system requires a stronger effort.

The *Vhdl-Ams* has a strong and rich type system: we have scalar and non scalar types, it is

possible to define new types and subtypes, among them we can define multidimensional arrays and some complex operators like slicing are provided.

Moreover *Vhdl-Ams* defines different kind of objects which are manipulated by expressions and statements. Like any procedural language it has constants and variables, but it has also quantities and signals which are the basic elements for an analog and digital simulation respectively.

The building blocks of a system described by *Vhdl-Ams* are components. A component is described by an entity which defines its interface to the environment, and by one or more architectures which define its behavior. A good mapping for this is to describe each entity with a class which defines only ports and generics, and each architecture with a derived class. Using inheritance each class defining an architecture has access to generics and ports defined in the parent (entity) class.

The translation of each component generates a directory with the name of the associated entity containing a set of C++ files: a *primary.hh* file containing the class describing the entity, a set of files *entityname_architecturename.cpp*, one for each architecture for that component, a set of dynamic linked libraries *entityname_architecturename.dll* (note extension *.dll* in windows, in linux we have a different extension) one for each architecture for the component. The *primary.hh* file containing the declaration of an entity class will be included by all the component architectures which use that component.

A well defined directory hierarchy is mandatory to follow the hierarchical system organization of *Vhdl-Ams* libraries, e.g. we can use the same name for different components located in different libraries. We defined also a more generic abstract class called component. This is used by the simulation engine to handle the whole system architecture to be simulated. The simulation engine cannot access specific components, but only anonymous components. A key method of the simulation engine is the *getComponent* method, that allows the engine to dynamically load every component (or a component to dynamically load every inner components). By using this method the engine recursively loads the components hierarchy which defines the system to be simulated. The dynamic load of all components of the system is not OS dependent and Matlab™ can handle it without any problem. This modular approach allows for the development of large systems: it is possible to build and test single components individually, to reuse old components previously generated, to glue up the whole system compiling only the new added components.

In a previous work [1] we focused on the analog domain: in this paper we aim to show the main architecture of the digital part.

The digital domain introduces the problem of some *Vhdl-Ams* specific elements mapping. We can summarize them as: *i*) signals; *ii*) signal assignments statements; *iii*) wait sequential statements; *iv*) process concurrent statements.

Signals are the basic elements manipulated by a digital system. We map a signal to a *Signal* class containing a value of a given type and a set of specific signal informations, e.g. a flag telling if an event has raised for the signal and the list of processes sensible to that signal. Signal assignment statements are mapped into transactions, i.e. a signal assignment generates a new transaction which will be handled by the digital solver. Each concurrent statement of a component, and specifically the process statement, is mapped into a class *Process* plus some methods related to the process. The *Process* class contains some information about the status of the process, a phase which acts as a program counter and tells where the execution has to resume for a resumed process. Moreover it contains also a pointer to the run method.

The run method contains the sequential code that each process has to execute. The other methods related to the process are: *i*) the init method; *ii*) the register info method. The former initializes the process status and phase, the latter notify the process to all signals to which it is sensible.

The main problem to handle was to restart the execution of a resumed process from where it has stopped. We solved this by using the phase vari-

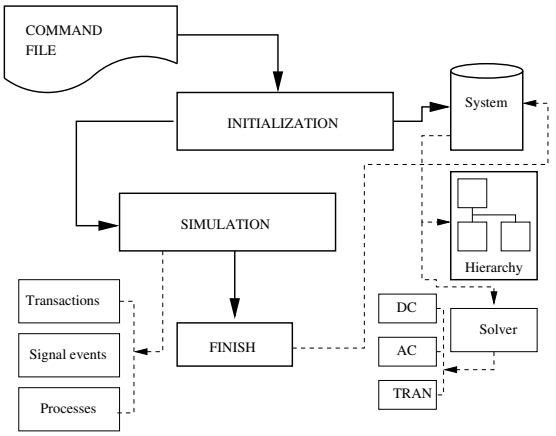

Figure 1: Schematic diagram of the simulation algorithm.

able and a switch of goto at the beginning of the run method which allows to jump to a given label inside the process code. The wait statement is thus simply mapped with some lines of code which set the status of the process to SUSPEND, define a return for the run method, and set a label after the return statement to indicate where the execution has to restart when process will resume. All inner data of a process are maintained by the component containing the process, e.g. we cannot place process variables inside the run method since we have to maintain the value during process execution-resume phase. We avoided name clash by redefining local names by attaching a prefix indicating an unique identifier for the process.

3 Simulation algorithm

The digital simulation algorithm is based on the IEEE standard specification; in this work we will focus on some implementative approaches we have adopted.

A class called *System* stores the global simulation variables visible to all components. Simulation steps are the following (Fig. 1): *i*) command file elaboration, initialization phase; *ii*) *System* creation, hierarchy root retrieve and solver creation; *iii*) simulation phase; *iv*) deallocation phase. Some of these steps will be explained in the following sections.

3.1 Initialization phase

The command file stores everything should be known to simulate a specified architecture of a primary unit. In this file it is also indicated the simulation end time, the simulator options and the variables that should be exported to the Matlab™ workspace or to the output file. After the command file elaboration, the *System* class is created and the component root is retrieved. The dynamically linked library for each component is loaded in memory, and a cache list with loaded components is updated. Some *Vhdl-Ams* systems can have a lot of instances of the same component, and the cache avoid to load the same library many times and speed up this simulation phase. When an instance is created, the component constructor is called, the component internal data and digital processes are initialized. Each component will also load its instanced units and update the hierarchy tree. When loading instances the port and generic mapping is updated.

The solver is created and initialized according to the simulation type. Each solver (AC, TRAN, DC, DIGITAL) is derived from a class *Solver*. By calling the *Solver->simulate()* method the simulation is performed, as briefly explained in the next section.

3.2 Simulation phase

The simulation is performed according to the IEEE directives for the digital simulation [3]. It is not the purpose of this work to give a detailed explanation of the whole simulation algorithm, but we will focus on some implementation aspects.

Basically, the digital simulation is made by making some scheduled transactions which can change a signal value or activate suspended processes. After the activation, processes are executed and new transactions are scheduled. The challenge is to perform these tasks in an optimized way. In the first developed digital engine the simulation procedure was composed by the following steps: *i*) make transactions; *ii*) look for active processes by walking through the hierarchy tree and calling a specific *checkActive* method for each process; *iii*) walk through the hierarchy tree and call all active processes; *iv*) clear signal

events. In spite of its simplicity, this procedure resulted to be very slow, basically due to the tree walking for each simulation cycle. A most appealing solution was analyzed and implemented on current digital kernel.

It is based on the reduction of the hierarchy tree access time, which is due to the process activation-execution phase. Our approach to handle process execution after a signal event or a transaction activation is the following:

- during the instance initialization each process pointer is stored on signals that can activate that process;
- a list with all signals that have the event flag raised and an active process list are kept in memory. The signal list and the process list are updated by the same signals when an event occurs;
- we walk through the active process list at the current simulation time, and the process run method is called.

This solution had the effect to reduce the simulation time to the 10% of the first digital kernel simulation time we implemented.

4 Simulation example

Many tests where performed to validate the digital simulation engine. In this section the VHDL [8] implementation of a low-power multistage decimator [9] for a wideband Δ-Σ analog to digital converter (ADC) is presented. More exactly, the output of a 2-bit 2nd order Δ-Σ

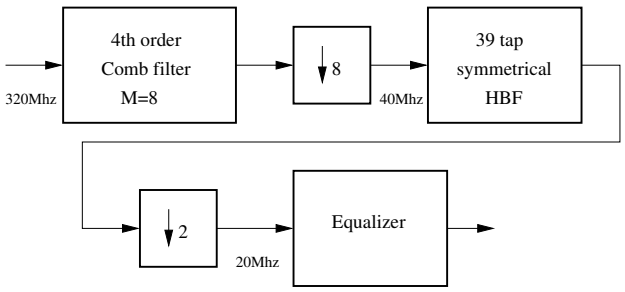

Figure 2: Block diagram of the implemented decimator.

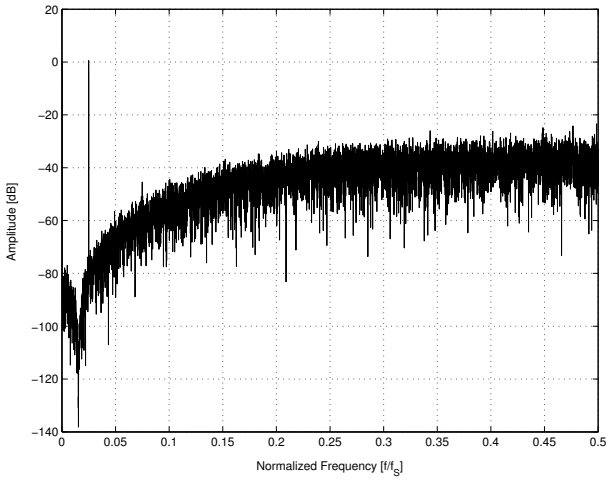

Figure 3: Δ-Σ modulator output power spectral density.

Modulator (DSM) with optimized Noise Transfer Function (NTF) is supposed to be sampled at Fs=320Ms/s having an oversampling ratio OSR=Fs/2fB=18.82 where fB=8.5MHz is the baseband frequency of the signal of interest (like could be for example an OFDM signal in the standard IEEE802.11). The simulated decimator down-samples the output of the converter to 20Ms/s applying a decimation factor of M=16 and meeting requirements like almost same SNR at the output of the decimator as in input (SNR=60[dB] or 10 equivalent bits), low power consumption (less than 10mW) and latency delay less than $1\mu s$. To meet these constraint a cascade of a 4th order Comb Filter with decimation ratio M=8, 39 tap symmetric Half Band filter are used. To compensate the drop in the baseband due to the comb filter an equalizer as final stage is required. Figure 3 shows the power spectral density of the signal taken from Δ-Σ Modulator. Simulation results are shown in Figures 4 and 5. These results where compared with the results obtained with other commercial simulators, giving a perfect agreement.

The implemented and simulated decimator has a total number of 74 different components and 607 instances, and a total number of 4478 processes.

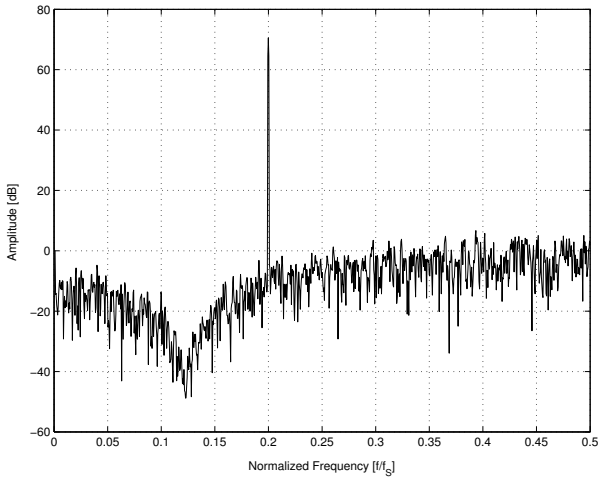

Figure 4: Comb Filter output power spectral density ($N_{FFT} = 2048$ samples).

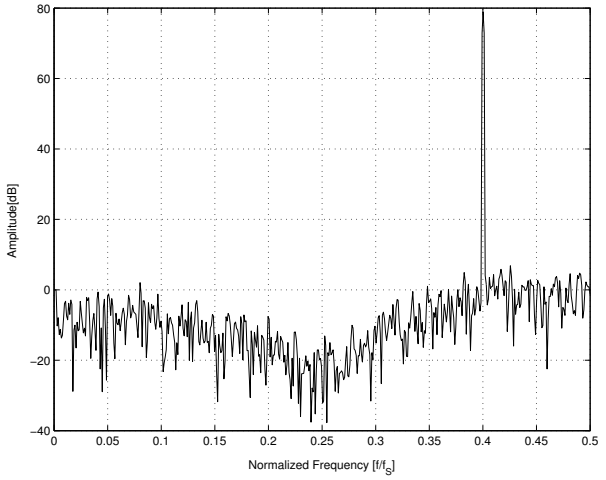

Figure 5: Decimated signal power spectral density ($N_{FFT} = 1024$ samples).

5 Conclusions

In this work we presented the *S.A.M.S.A.* digital kernel implementation, and the simulation results regarding a low-power multistage decimator used to validate our tool. The simulator was able to correctly simulate the whole system giving the same numerical results as other commercial VHDL tools. Moreover, *S.A.M.S.A.* allows for the simulation integration with the Matlab™toolboxes, giving to it a great flexibility and the capability to use all the numerical features of this numerical scientific tool. The compiler actually allows to compile almost all of the clauses defined in the IEEE standard, and in particular it is possible to compile all the standardized libraries as *std_logic_1164*, *std_logic_signed*, *std_logic_unsigned*, *std_logic_arith*.

The proposed tool is part of a research project for the development of a Matlab™ simulation framework for fully analog/mixed signal simulation.

Actually the analog and digital kernel are available in *S.A.M.S.A.*, and a fully mixed-signal engine is on-going.

Acknowledgments

The authors would like to thank Dr. A. Marcianesi for help, work and suggestions about the decimator and Δ-Σ modulator.

6 References

[1] M. Zorzi, N. Speciale, G. Masetti, *"A New VHDL-AMS Simulation Framework in Matlab"* BMAS 2002, Santa Rosa (CA), 6-8 October 2002.

[2] *"SPECTRE HDL Reference"*, Cadence Design Systems, 1998.

[3] *"IEEE Standard VHDL Analog and Mixed-Signal Extensions"*, IEEE Std 1076.1-1999.

[4] A. Vladimirescu, *"The SPICE Book"*, John Wiley & Sons, Inc. 1994.

[5] M. Zorzi, F. Franzè, N. Speciale *"I.M.A.GE.: a new CAD Tool for Device Modeling in Spice"*, Proc. of ECCTD01, Espoo, Finland, pp 241–244.

[6] F. L. Cox, W. B. Kuhn, J. P. Murray and S. D. Tynnor, *"Code-Level Modeling in XSPICE"*, Proc. of ISCAS'92. Vol 2, 1992, pp: 871-874.

[7] *"Matlab Reference Documentation"*, Ver. 6, Mathworks.

[8] P. J. Ashenden, *"The Designer's Guide to VHDL"*, Morgan Kaufmann Publishers, Inc. 1996.

[9] S. R. Norsworthy, R. Schreier, G. C. Temes, *"Delta-Sigma Data Converters: Theory, Design, and Simulation"*, New York: IEEE Press, 1996.