April 4th 2003

VHDL Mixed Signal Modeling and Simulation

Shaylesh Mehta (265690) Group 90
Supervisor: Professor G. Wainer

Abstract

VHLD Mixed Signal Modeling and Signal Modeling and Simulation

To facilitate simulation of mixed signal HDL models within a DEVS simulator, generic DEVS models and HDL to DEVS conversion procedures are required. These models and conversion procedures are designed for a subset of VHDL created for this project named sAMS-VHDL, and are targeted toward the CD++ DEVS simulation toolkit. Hierarchical models written in sAMS-VHDL that utilize Processes, Signals and Simultaneous Statements may be simulated in CD++ by elaborating the model, and converting the model hierarchy into an equivalent CD++ coupled model composed of Process, Signal and Integrator models. These Process, Signal and Integrator models and their associated conversion procedures were designed and then tested in CD++ using a number of characteristic sAMS-VHDL models.

Acknowledgments

Special thanks to Professor G. Wainer for all of his help, Akara Corporation for all of their tutelage and Dhanu and Bobby for their support.

Table Of Contents

List of Figures

Introduction

Today's technology business climate requires hardware designers be fast; not only when designing new technology, but throughout the design and maintenance cycle. Digital designers have known for some time, that thorough modeling and simulation of designs reduces the number of design bugs, reduces the number of integration errors, eases product maintenance and saves money. Design and simulation of digital logic with HDLs (Hardware Descriptor Languages) is a well-proven methodology; digital designers have a rich toolset available for designing and verifying logic before manufacturing. Such robust toolsets for analog and mixed signal design and simulation have yet to be developed, those currently available have many limitations and do not exhibit desirable performance. A suitable mixed signal simulator would give the designer the ability to optimize, debug, and verify designs with lower simulation tool costs, lower risk on manufacturing investment and faster turn around time.

Problem Motivation

The key design challenges for a mixed signal simulator are firstly, providing desirable performance while maintaining accuracy of continuous time signals, and secondly, concurrently executing the simulations of the discrete time digital and continuous time analog models [1]. A proposed solution to these challenges presented in [1] is to simulate mixed signal HDL models in a simulator that implements the DEVS (Discrete EVent Simulation) formalism (see Section 3). This solution requires the HDL model to be converted to a semantically equivalent model that may be executed using a DEVS simulator.

Problem Statement

To simulate a mixed signal HDL model within a DEVS simulator, generic models that capture the semantics of the constructs within the HDL model must be developed for the DEVS simulator. A conversion procedure must then be developed to capture the structure and semantics of a given mixed signal HDL model within the DEVS simulator using the developed generic models.

Proposed Solution

I propose to select and/or invent a set of VHDL-AMS(**V**ery High Speed Integrated Circuit **H**ardware **D**escriptor **L**anguage **A**nalog **M**ixed **S**ignal) constructs for digital and analog elements and, design the necessary generic DEVS models and conversion procedures to simulate designs utilizing these constructs in the CD++ toolset which implements the DEVS formalism(see 3.1). This set of VHDL constructs with analog extensions will be referred to as **sAMS-VHDL** (simple Analog Mixed Signal VHDL).

Accomplishments

The sAMS-VHDL language was specified, borrowing many constructs from VHDL-AMS. Generic models for simulation of sAMS-VHDL Processes, Signals and Simultaneous Statements were developed in the CD++ toolkit, as well as procedures for converting hierarchical models written in sAMS-VHDL using these constructs into hierarchical CD++ models. The conversion procedures and CD++ models were tested by simulating manually converted sAMS-VHDL models in the CD++ toolkit.

1.0 Overview of Report

This report begins with a formal specification of the sAMS-VHDL language in section 2.0, which includes the language grammar as well as a description of the semantics of the various sAMS-VHDL constructs. Section 3.0 and its subsections are an overview of the atomic and coupled DEVS formalisms and explain their implementation in the CD++ toolkit. The process of specifying and simulating a coupled or atomic DEVS model in the CD++ toolkit is also included in this section. Section 4.0 first provides an overview of the process for sAMS-VHDL model to CD++ coupled model conversion. Following this the procedure for sAMS-VHDL model hierarchy conversion is explained in detail in section 4.2. Sections 4.4 and 4.5 are detailed descriptions of the CD++ Process and Signal models respectively. These sections also include explicit mapping rules for their respective constructs. This is followed by an in depth discussion of ODE integration via a number of methods in sections 4.6.1 to 4.6.4. This is directly followed by a description of the design of the CD++ Integrator model and its associated conversion procedure in section 4.6.5. Appendix A contains CD++ models and simulation results for a few example sAMS-VHDL models.

2.0 sAMS VHDL Language

This section outlines the sAMS VHDL language, and explains the syntax and semantics of it's constructs. sAMS VHDL is targeted toward register transfer level modeling of digital circuits with limited behavioral modeling and analog constructs. sAMS VHDL integrates many of the features of VHDL-AMS and explicitly includes some of the types and functions defined by the IEEE 1164 standard logic package.

2.1 Entity

A design entity declaration describes the interface to a sAMS VHDL design or design unit. The entity declaration contains a list of ports, each of which is assigned a type and an optional mode. Ports of type std_logic or std_logic_vector (a standardized type for digital logic) are used for digital signals while ports of type electrical are used for analog signals. In the case of digital signals, ports will have mode **in**, **out**, **inout** or **buffer**. Analog ports do not require a mode. The syntax of an entity declaration is as follows:

```
entity  entity_name is
    port ( [signal |  terminal | quantity] identifier {, identifier}: [mode |  signal_type |
electrical]
       {; [signal | terminal | quantity ] identifier {, identifier}: [mode | signal_type |
electrical]);
    ) ;
end [entity] [entity_name] ;
```

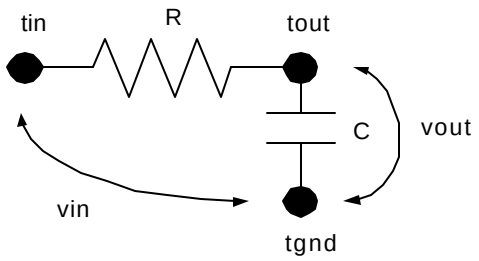
An entity declaration of a digital d flip-flop is listed below:

```
entity  d_flip_flop is
     port(
       d, clk : in std_logic;
       q: out std_logic;
      ) ;
end entity d_flip_flop ;
```

d and clk are input ports of type std_logic, and q is an output port of type std_logic. In addition to the basic std_logic type, vectors of std_logic signals may be declared using the std_logic_vector type. This allows busses of digital

signals to be operated on by only referencing one signal name. A declaration for an analog low pass filter entity is listed below:

Figure 1 Low-pass Filter




```
entity LPF is
        port (
         terminal tout, tin, tgnd:
electrical
        );
end entity LFP;
```

tout, tin and tgnd are each ports of type electrical which represent the nodes tout, tin and tgnd in Figure 1 respectively.

2.2 Architecture

A design architecture describes the functionality of a design or design unit; this may be a structural, dataflow or behavioral description. A single architecture is associated with exactly one entity. The syntax of an architecture declaration is as follows:

```
architecture architecture_name of entity_name is
    signal_declaration
    | constant_declaration
    | component_declaration
begin
    {process_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
    | simultaneous_statement}
end [architecture] [architecture_name] ;
```

The body of an architecture is made up of statements that may be categorized as concurrent, sequential or simultaneous. These statements operate on signals and quantities that are declared within the scope of the architecture and ports that are declared in the entity the architecture is associated with.

2.3.1 Signal and Quantity Declaration

Signals and quantities are declared in the declarative region of a design architecture, this is the region between the architecture statement and the begin clause. These signals and quantities belong to the scope of the architecture in which they are declared and may only be referenced within that architecture. Signals and quantities are assigned types similar to ports in the entity declaration. The types std_logic and std_logic_vector are used for digital logic, these are declared using the signal keyword whose syntax is listed below:

signal signal_name : **std_logic_vector** (upper_bound **downto** lower_bound) | **std_logic** ;

Analog quantities are declared using the quantity keyword, and have type REAL, Voltage, Current or Charge. The syntax for a quantity declaration is listed below:

quantity identifier : **REAL** | **Voltage** | **Current** | **Charge** ;

Quantities may also be declared relative to terminals in an entity declaration. These quantities may either be across or through quantities. Across quantities represent the voltage at the free terminal relative to the reference terminal. Through quantities represent the current from the free terminal into the reference terminal. The syntax for a terminal relative quantity declaration is listed below:

quantity identifier {, identifier} **across** identifier {, identifier} **through** free_terminal **to** reference_terminal ;

2.3.2 Concurrent Statements

Concurrent statements are statements within an architecture body that execute concurrently. These include Process Statements, Simultaneous Statements, Concurrent Assignment Statements and Conditional Concurrent Assignment Statements.

2.3.2.1 Conditional/Unconditional Concurrent Assignment Statement

The conditional concurrent assignment statement assigns the target signal the value of expression1 if condition is true otherwise target signal is assigned the value of expression2.

target_signal <= expression1 **when** condition **else** expression2;

The unconditional concurrent assignment statement always assigns the value of the source signal to the target signal.

target_signal <= source_signal;

2.3.3 Process Statement

A process executes the statements between begin and end process when an event occurs on a signal in it's sensitivity list. All signals modified by the process are not updated until the process body is completed. The statements that appear between the begin and end clauses are referred to as sequential statements, as the name implies, these statements are executed in sequence until the end clause is encountered.

```
[process_name:]
process (sensitivity_list)
   { type_declaration
   }
begin
  {signal_assignment_statement
  | if_statement
  | case_statement
  end process [process_name] ;
```

2.3.3 Sequential Statements

Sequential statements appear within the body of a process, these statements are executed sequentially from the process begin clause to the process end clause.

2.3.3.1 If-Then-Else Statement

This statement has identical semantics to that of an if-then-else statement in C/C++

```
[ if_name: ] if condition then
        sequence_of_statements
    {elsif condition2 then
        sequence_of_statements }
    [else
        sequence_of_statements ]
    end if [ if_name ] ;
```

2.3.3.2 Case-When Statement

This statement runs the sequence of statements that are listed under the when clause whose expression matches that of the expression in the case statement.

```
[ case_name: ] case expression is
            {when identifier | expression | discrete_range | others =>
              sequence_of_statements}
            {when identifier | expression | discrete_range | others =>
              sequence_of_statements}
        end case [ case_name ] ;
```

2.3.3.3 Sequential Assignment Statement

This statement assigns the value of the driver signal to the target signal. When executed from within a process, the target will not get the value of the driver until the end of the process.

```
[ label: ] target <= driver ;
```

2.3.4 Simultaneous Statements

Simultaneous statements are generally used for describing Differential Algebraic Equations, and may consist of quantities or signals. For the purposes of this project sAMS VHDL will only support Ordinary Differential Equations in simultaneous statements. A minimum of one quantity must appear in a simultaneous statement. Simultaneous statements may appear anywhere a concurrent statement may. The order of simultaneous statements is unimportant. An example of simultaneous statements is given below:

```
x1'dot'dot == -f*(x1 – x2) / m1;
x2'dot'dot == -f*(x2 – x1) / m2;
...
```

The 'dot notation denotes the derivative with respect to time of the quantity listed before the 'dot. For example signal'dot is the first derivative wrt time of signal, while signal'dot'dot is the second derivative wrt time of signal.

2.4 Components and Component Instances

Components facilitate hierarchical design within sAMS VHDL models. A component instance is a copy of the named entity and it's associated architecture that interacts with the architecture it is instantiated within. The port map clause specifies which ports of the entity are connected to which signals in the enclosing architecture body. The syntax for a component instantiation statement is listed below:

```
Instantiation_label :
entity entity_name
port map (
        port_name => signal_name | expression | variable_name | open
        {, port_name => signal_name | expression | variable_name | open}
);
```

3.0 The DEVS Formalism and CD++

DEVS is a theoretical approach, which allows the definition of hierarchical models that can be easily reused (Zeigler 1976, Zeigler at al. 2000). DEVS models may be described as a set of communicating atomic or coupled submodels.

3.0.1 Atomic DEVS

An atomic DEVS model is formally described by:

$$M = <X, S, Y, d_{int}, d_{ext}, l, D>$$

Where

X: the input events set
S: the state set
Y: the output events set
$\mathbf{d_{int}}$: internal transition function
$\mathbf{d_{ext}}$: external transition function
l: the output function
D: the elapsed time function

Each atomic model is provided with input and output ports that allow the model to communicate with other models. The input events set is made up of all possible inputs that may occur on the input ports, similarly the output set consists of all possible outputs the atomic model may have. The external transition function is invoked when an event occurs on an input port; this function determines what state change if any is required as a result of the event and the current state. The model remains in its current state for an amount of time determined by the elapsed time function, when this time has expired the output function is invoked, which sends output events from the output set on the output ports based on the current state. Following the invocation of the output function, the internal function is immediately

invoked, which determines which state change if any is required as a result of the current state.

3.0.2 Coupled DEVS

A coupled DEVS model is composed of a set of atomic or coupled sub-models. They are formally defined as:

$$\mathbf{CM = <I, X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\} >}$$

Where

I: the models interface
X: the input events set
Y: the output events set
D: an index for the components of the coupled model
$\mathbf{M_i}$: is a basic DEVS (atomic or coupled) model
$\mathbf{I_i}$: the set of influencees of model I
$\mathbf{Z_{ij}}$: i to j translation function

Coupled models are defined by a set of basic components which are interconnected through their model interfaces. The influences set determines which components should receive the outputs of each component. The translation function converts the outputs of one component to the inputs of other components.

3.1 CD++ Atomic and Coupled Model Definition

Atomic models are created within the CD++ toolkit by creating C++ classes that are derivatives of the class *Atomic*. This new class must then overload the initFunction, externalFunction, internalFunction and outputFunction methods within the *Atomic* class. Each Atomic model

instantiates ports which are unidirectional, either input or output. These ports are used to exchange event messages between different atomic models.

InitFunction: This method is invoked when the simulation starts, it performs the method body as well as setting the model state to passive and setting the time for the next scheduled event to infinity.

externalFunction: This method is invoked when an even occurs on one of the atomic models ports.

internalFunction: This method is invoked when the next event timer(D) has elapsed and after the outputFunction has been invoked.

outputFunction: this method is invoked when the next event timer(D) has elapsed.

The following primitives can be used to define the Atomic model behavior:

holdIn(state,time): instructs the model to remain in **state** for the specified **time**, following which the output and internal transition methods will be invoked.

passivate(): equivalent to holdIn(passive,infinity)

sendOutput(time,port,value): sends a message out on port:**port** at time:**time** with value:**value**

state(): returns the current state of the model.

After creating a new Atomic model class, the class must be registered with the simulator by invoking the SingleModelAdm::Instance().registerAtomic method from within the MainSimulator::registerNewAtomics() method.

Following registration the new atomic model should be added to the simulator makefile, and this makefile should be executed. This will compile the simulator and all new atomic models.

Following compilation, the new atomic model may be instantiated within a model (MA) file, which defines a coupled DEVS model. A model (MA) file consists of components, atomic model instances and links. A sample model file is listed below:

```
[top]
components : transducer@Transducer generator@Generator Consumer@Consumer
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out
[Consumer]
components : queue@Queue processor@Processor
in : in
out : out
Link : in in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor out
```

The model file is made up of components that contain instances of Atomic models. There must always be at least on component in the MA file, this is sometimes referred to as the top level component or top. A component is specified by inserting a line with the component name surrounded by square

brackets. The example above has two components: Consumer and top. The line following the component name specifies instances of atomic models or components that are used within the component. This line begins with the keyword *component:* followed by a list of atomic model or component instances. Atomic model instances have syntax:

instance_name@atomic_model_name

while component instances have syntax

instance_name@component_name.

The next two lines in the component definition list the input and output ports for the component. The keyword *out:* is followed by a list of output ports for the component. The keyword *in:* is followed by a list of input ports for the component. Finally lines beginning with the keyword *Link:* specify links between ports on any two of the following: ports on atomic model instances, ports on component instances, ports on the component to which the link belongs. The syntax of a link is:

Link **source_port**@instance_name **dest_port**@instance_name

4.0 sAMS VHDL to CD++ Coupled Model Conversion

This section describes the procedures and CD++ models that are used to convert designs written in sAMS VHDL into CD++ models that may be simulated in the CD++ toolkit.

4.1 Simulation Dataflow

The dataflow for the simulation of sAMS VHDL models in CD++ is illustrated in Figure 2. The conversion should begin with a syntax check; to

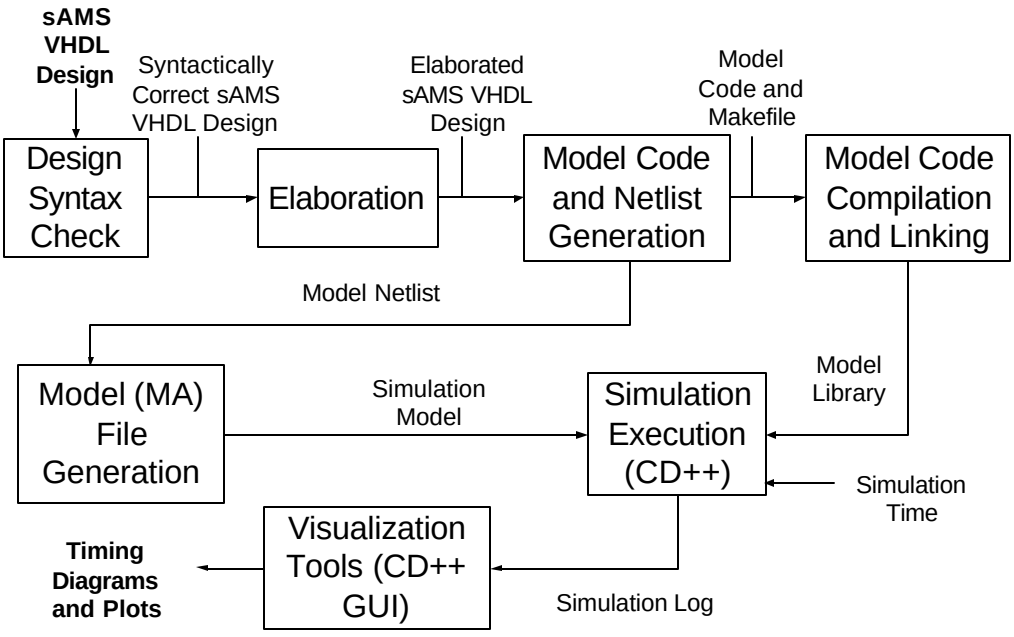


Figure 2 Simulation Dataflow

ensure the sAMS VHDL model is syntactically correct before it is passed on to the elaboration phase. VHDL elaboration is defined by the VHDL specification as the process by which a declaration achieves its effect. The elaboration procedure yields, a description of the structure of each component in the sAMS VHDL design hierarchy. The architecture and entity description for each component in the design is parsed in order to produce a netlist in terms of interconnected integrators, algebraic operators, processes, signals and sub-component instances. This netlist is then passed to the model code and netlist generation procedure. This procedure then generates the CD++ model code for each of the processes in each of the components of the design. The CD++ process models are then compiled into a model library by executing a makefile that lists each of the CD++ process models. Following compilation, the netlist and model library are used by the model (MA) file generation process to yield a coupled CD++ model definition file. This definition contains a coupled model representing each component from the sAMS VHDL design and a top level coupled model. Following model (MA) file generation, the coupled model definition for the sAMS VHDL design may be simulated in the CD++ toolkit.

4.2 sAMS VHDL Design Hierarchies to CD++ Coupled Model Hierarchies

sAMS VHDL models hierarchies are converted to CD++ coupled model hierarchies during the Model Code and Netlist Generation phase of the conversion process. The components that constitute the design hierarchy must first be differentiated based on whether they are a basic component or an aggregate component. Basic components do not contain sub-component instances in their architectures, while aggregate components may have one or more sub-component instances in their architectures. A dependency tree must be generated next; the leaves of the tree will always be basic components, while branches will be aggregate components composed of either basic components or aggregate components, the root will be the top-level model. Figure 5 illustrates a dependency tree for the sAMS VHDL design hierarchy listed in Figure 3. Coupled CD++ models should be defined in the order dictated by the dependency tree, namely from the leaves toward the root. Figure 4 contains a CD++ coupled model definition for the sAMS VHDL design hierarchy of Figure 3, note the order of component declaration begins with the top level model and is followed by models that approach the leaves in the dependency tree.

4.2.1 Structural and Hierarchal Linking of Coupled Models

sAMS VHDL sub-component instances are connected to the architecture in which they are instantiated as defined by the port map clause in their component instantiation statement. This clause will connect either a signal within the architecture or a port on the architectures entity definition to each of the ports on the component instance. In the case of a signal the linking is termed structural, in the case of another port the linking is termed hierarchical. In both cases the mode of the sub-component port specified in the port map clause must be determined prior to generating link statements in the coupled model definition. In structural links if the ports mode is out, it is linked to the input port on the signal model specified in the clause, if the ports mode is in, the output port on the specified signal model is linked to it. In hierarchical links, if the sub-components port mode is out it is linked to the component port, if the sub-components port mode is in, the component port is linked to it. Figure 4 illustrates all four of these cases.

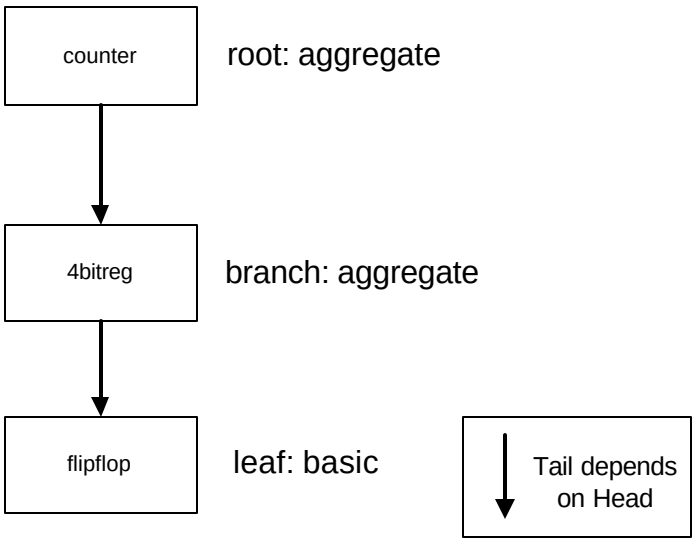
Figure 3 Hierarchical sAMS-VHDL Model

```
entity flipflop is
        port (clk, d : in std_logic;
              q : out std_logic;
             );
end entity flipflop;

architecture rtl of flipflop is
begin
        …
end architecture rtl;

entity 4bitreg is
        port (clk, d0, d1, d2, d3 : in std_logic;
              q0, q1, q2, q3 : out std_logic;
             );
end entity 4bitreg;

architecture rtl of 4bitreg is
begin

        b0: entity flipflop
                port map (clk, d0, q0);
        b1: entity flipflop
                port map (clk, d1, q1);
        b2: entity flipflop
                port map (clk, d2, q2);
        b3: entity flipflop
                port map (clk, d3, q3);

end architecture rtl;

entity counter is
        port ( clk : in std_logic
                …
             );
end entity flipflop;

architecture rtl of counter is
        signal lclk, ld0, ld1, ld2, ld3, lq0, lq1, lq2, lq3;
begin
        4b0 : entity 4bitreg
                port map (clk=>lclk, d0=>ld0, d1=>ld1, … , q0=>lq0,  …);
        …
end architecture rtl;
```

Figure 4 Hierarchical CD++ Model

```
[counter]
components : 4b0@4bitreg lclk@Signal ld0@Signal ld1@Signal lq0@Signal …

in: …
out: …

link : out@lclk clk@4b0
link : out@ld0 d0@4b0
link : out@ld1 d1@4b0
…

link : q0@4b0 in@lq0

[4bitreg]
components : b0@flipflop b1@flipflop b2@flipflop b3@flipflop

in : clk d0 d1 d2 d3
out: q0 q1 q2 q3

link : clk clk@b0
link : clk clk@b1
link : clk clk@b2
link : clk clk@b3

link : d0 d@b0
link : d1 d@b1
link : d2 d@b2
link : d3 d@b3

link : q@b0 q0
link : q@b1 q1
link : q@b2 q2
link : q@b3 q3

[flipflop]

in : clk d
out : q

…
```

Figure 5 Model Dependency Tree



4.3 Elaboration of Concurrent Statements

Concurrent statements in a sAMS VHDL design are converted to a process statement containing one or more sequential statements during the elaboration phase. For example the following concurrent conditional assignment statement:

```
x<='1' when (a=b or c='1') else '0' ;
```

Is converted into this equivalent process statement:

```
my_proc: process (a, b, c)
begin
        x<='0';
        if (a=b or c='1')
                x<='1'
        end if;
end process my_proc;
```

This conversion is possible for any and all concurrent statements in sAMS VHDL. As a result, the problem of converting concurrent statements to CD++ models is solved by first elaborating all concurrent statements to convert them into equivalent process statements and then converting those process statements to CD++ models.

4.4 CD++ Process Model

During conversion a CD++ Process Model is generated for each process model in the design hierarchy. The CD++ process model captures the semantics of a sAMS VHDL process statement by converting it's sequential statements to C++ code and instantiating ports for every signal that is read or driven from within the process and for every signal in the processes sensitivity list. Figure 6 illustrates the CD++ process model for the following flip-flop process:

```
flipflop: process (clk)
begin
   if( rising_edge(clk) )
      q<=d;
   end if;
end process my_proc;
```

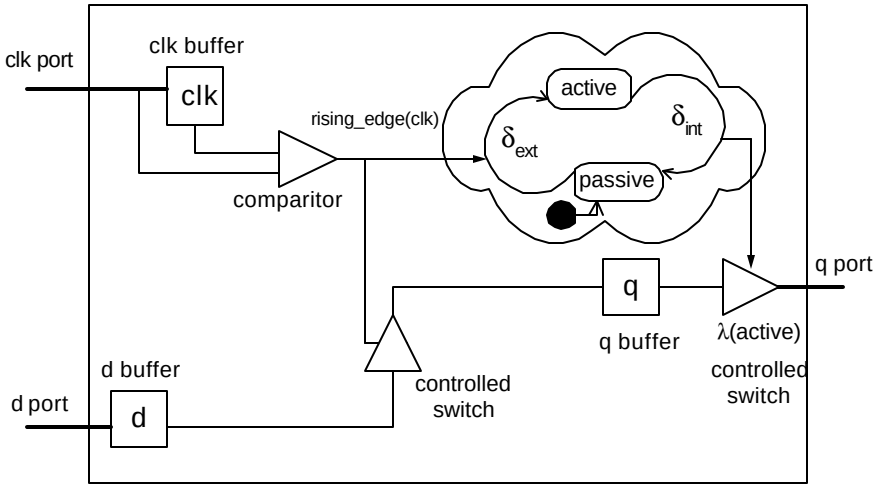


Figure 6 CD++ Process Model

The process body is implemented within the external transition function of the CD++ process model. The values received from all external events generated on the input ports that represent read and sensitivity list signals are buffered within the model. This is accomplished by inserting the following block of C++ code within the external transition function for each read or sensitivity list signal:

```
if (msg.port()==[signal name]) {
        [signal buffer]=msg.value(); //buffer inputs to processes
}
```

A special case exists if the process body contains a reference to the rising_edge(signal_name) or falling_edge(signal_name) operation. This operation determines if a rising edge or falling edge has occurred on a signal (0 to 1 or 1 to 0). In this case, the values received from all external events generated on the input port that represents the signal referenced in the rising_edge or falling_edge operation must be have a buffer of length two within the model. This maintains the previous value and current value of the signal. The buffering is accomplished by inserting the following block of C++ code within the external transition function for each signal referenced in a rising_edge or falling_edge operation.

```
if (msg.port()==[signal name]) {
        [old signal buffer name]=[new signal buffer name]; //keep last value of signal
        [new signal buffer name]=msg.value(); //buffer inputs to processes
}
```

Buffers are also created for each output port on the CD++ process model. The output ports on the model represent all the signals that are driven from within the sAMS VHDL process. The values that are assigned to these buffers will be output on their respective ports when the model schedules an output event.

The sequential statements in the process body can be converted directly to C++ code and inserted into the external transition function since they are sequentially executed and are semantically equivalent to C++ statements. sAMS VHDL If , case and assignment statements are converted directly into C++ if, switch and assignment statements. The boolean expression which refers to read and sensitivity list signals in the sAMS VHDL if statement is replaced with an equivalent boolean expression that refers to port buffers for those signals. This same procedure can be used to convert the expression in a sAMS VHDL case statement to an equivalent one in C++. In the case where the rising_edge or falling_edge operation appears within an if or case statement in the process body, rising_edge and falling_edge may be substituted with the following code in the C++ equivalent if or case statement:

```
![old signal buffer name] and [new signal buffer name] // rising edge
[old signal buffer name] and ![new signal buffer name] // falling edge
```

If the condition within the sAMS VHDL if statement contains a sensitivity list signal, the last piece of code within the C++ if condition body should instruct the process model to change to the active state for a time of zero. This will cause an output event and internal transition to be triggered. The output event will update all of the driven signals by sending the value of each output port buffer out on each output port. The internal transition will cause the model to return to the active state. The sAMS VHDL code for a process used in a four bit counter is listed below:

```
Counter: process (clk) is
begin
        if(rising_edge(clk))
                q1<=not d1;
                q2<=d1 xor d2;
                q3<= d3 xor (d1 and d2);
                q4<=d4 xor (d1 and d2 and d3);
        end if;
end Counter;
```

This process has one sensitivity list signal (clk), four read signals (d1, d2, d3 and d4) and four driven signals (q1, q2, q3 and q4). The process body contains an if sequential statement with a boolean expression that contains the rising_edge operation acting on signal clk, and 4 sequential assignment operations. The C++ code that represents the body of the above process is listed below. o_clk, n_clk, _d1, _d2, _d3 and _d4 are input port buffers, _q1, _q2, _q3 and _q4 are output port buffers. _1164and, _1164not and _1164xor are functions that implement and, not and xor operators in CD++.

```
if (msg.port()==clk) {  //since clk is in the trigger list
{
        o_clk=n_clk;
        n_clk=msg.value();
}
```

```
... //port buffer code for d1 d2 d3 d4

if(o_clk==0 && n_clk==1) { // if rising_edge(clk)
{
            _q1=_1164not(_d1);
             _q2=_1164xor(_d2,_d1);
             _q3=_1164xor(_d3,_1164and(_d1,_d2));
             _q4=_1164xor(_d4,_1164and(_d3,_1164and(_d1,_d2)));

             holdIn(active,0);
 }
```

4.5 CD++ Signal Model

The CD++ signal model is not an implementation of sAMS VHDL signals in CD++. Signals in the sAMS VHDL design are used to determine how the ports on the many process model instances must be interconnected for each component, this information is then used during model file generation to create links. Thus, a signal model is not needed for the purpose of interconnecting processes, instead the CD++ signal model is used to implement transport delay on messages sent between process model ports.

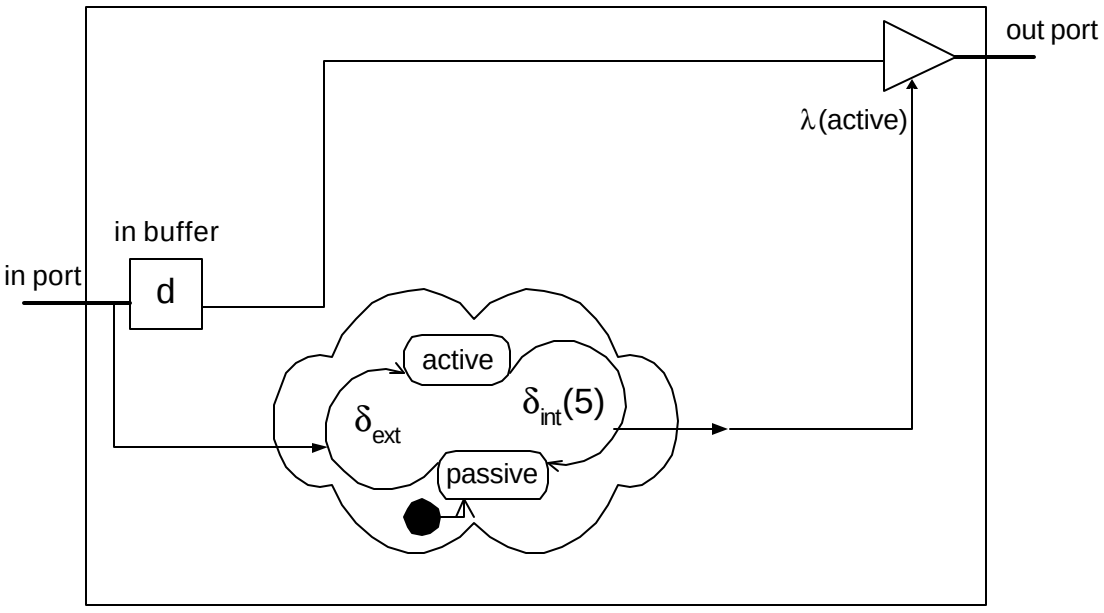


Figure 7 CD++ Signal Model

The implementation of the CD++ process model does not allow assignment statements to have transport delay, since the output events for all driven signals must occur simultaneously. To remedy this shortcoming, transport delay is implemented in the CD++ signal model. The signal model simply receives and buffers data on it's input port, enters the active state for the time specified by the assignment statement transport delay, then outputs the buffered data on it's output port.

Figure 7 illustrates the CD++ Signal Model for the sAMS VHDL assignment statement below:

```
signal my_signal, x, y, z: std_logic;
...
my_signal<=x after 5ns;
```

4.6 Simultaneous Statements and DAE Simulation

Simultaneous statements in sAMS VHDL allow the definition of continuous time systems through differential algebraic equations (DAE). For the purposes of this project, the problem of solving differential algebraic equation systems was confined to solving ordinary differential equation systems with initial conditions.

4.6.1 ODE Simulation via Integration

The problem of simulating an n^{th} ordinary differential equation is solved by reducing the n^{th} order ordinary differential equation into a set of first order differential equations. For example:

$$\frac{d^2y}{dx^2} + p(x)\frac{dy}{dx} = q(x)$$

can be written as two first-order differential equations:

$$\frac{dy}{dx} = z(x)$$
$$\frac{dz}{dx} = q(x) - p(x)z(x)$$

In general, an n^{th} order ordinary differential equation of form:

$$F(t, y, y', y'', ..., y^{(n)}) = 0 \quad (1)$$

may be decomposed into a set of first order differential equations:

$$\frac{dy_i(t)}{dt} = f_i(t, y_1, ..., y_N), \qquad i = 1, ..., N \quad (2)$$

where each $f_i(t, y_1, ..., y_N)$ is known

A solution for each $y_i(t)$ is obtained for some t >0 and set $y_i(0)$ by integrating each $\frac{dy_i(t)}{dt}$. This is most simply accomplished by replacing each y_i by Δy and each dt by Δt in (2) and then multiplying (2) by Δt. An acceptable level of accuracy is obtained from this approach providing Δt is sufficiently small. This type of integration is known as Euler's method.

4.6.2 Euler's Method Integration

The formula for Euler's method is

$$y_{n+1} = y_n + hf(t_n, y_n) \qquad (3)$$

where y_{n+1} is the value of y at the end of the interval h
and y_n is the value of y at the begining of the interval

As shown in Figure 8 this method extrapolates the solution for y over the interval t_n to $t_{n+1} \equiv t_n + h$ using the derivative at the beginning of the interval $f(t_n, y_n)$. Because the Euler's method uses the slope at the beginning of the

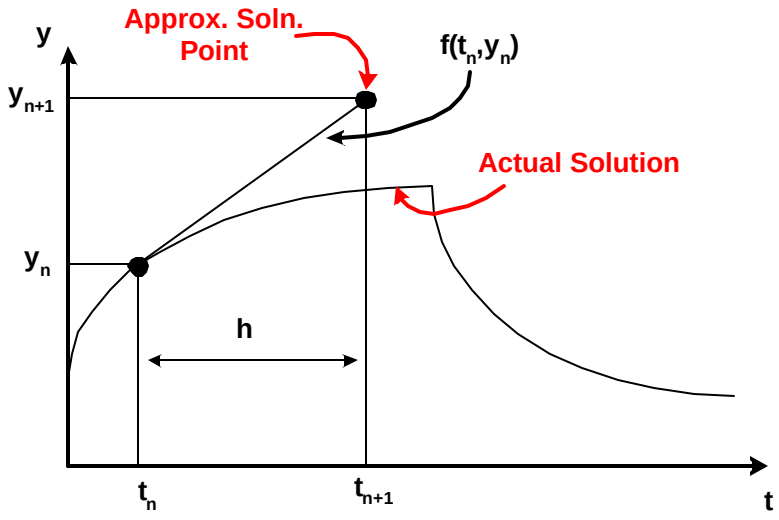


Figure 8 Euler Integration

interval only it is generally inaccurate compared to other methods, and can be unstable for large h.

4.6.3 Fourth-order Runge-Kutta Method Integration

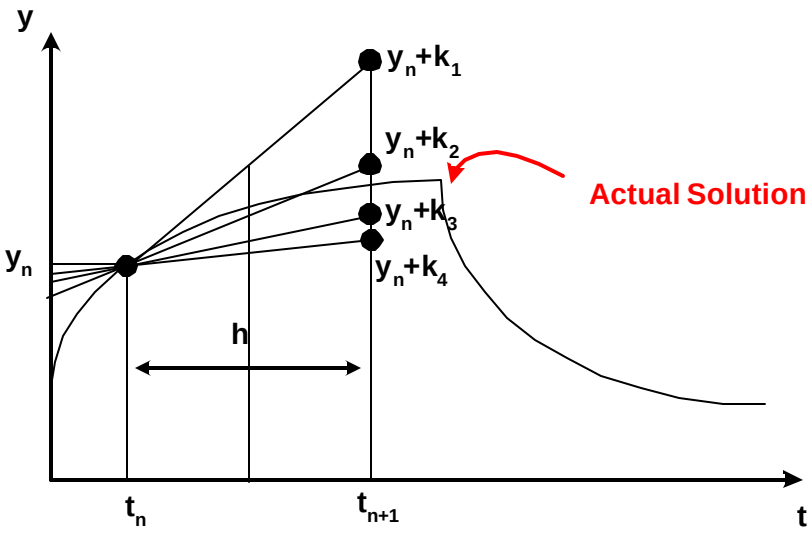
The Fourth-order Runge-Kutta method is generally accepted to be more accurate and stable when compared to the Euler's method for a given step size. The Fourth-order Runge-Kutta method does not rely on the derivative at the beginning of the interval only, but rather uses the derivative at the beginning of the interval, the derivative at two trial midpoints and the derivative at a trial end point. The equations for the Fourth-order Runge-Kutta method are as follows, and the method is illustrated in Figure 9:

$$k_1 = hf(t_n, y_n)$$
$$k_2 = hf(t_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$
$$k_3 = hf(t_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$
$$k_4 = hf(t_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \qquad (4)$$

where y_{n+1} is the value of y at the end of the interval h
and y_n is the value of y at the begining of the interval

Figure 9 Runge-Kutta Integration



The Fourth-order Runge-Kutta method first extrapolates the derivative at the beginning of the interval (t_n) to the end of the interval (t_{n+1}) to determine k_1. Then a trial step is taken to the midpoint of the interval, and the derivative is evaluated at this point using $y_n + \frac{k_1}{2}$, this derivative is then used to extrapolate the solution to the end of the interval to determine k_2. A second trial step is taken to the midpoint of the interval, and the derivative is evaluated at this point using $y_n + \frac{k_2}{2}$, this derivative is then used to extrapolate the solution to the end of the interval to determine k_3. A final trail step is taken to the end of the interval, and the derivative is evaluated at this point using $y_n + k_3$, this derivative is then used to extrapolate the solution to

the end of the interval to determine k_4. Finally a weighted sum of k_1, k_2, k_3 and k_4 is added to y_n to determine y_{n+1}.

4.6.4 Quantized State Systems with Runge-Kutta Integration

Continuous time ODE systems with initial conditions have traditionally been simulated by discretizing the time domain, and solving the ODE over each discrete time interval. An alternative approach introduced by Zeigler et al.[12], suggests discretizing the state space of the solution rather than the time domain to solve the ODE system. Systems using this alternative approach are termed quantized state systems. This approach requires a fundamental shift in thinking about the system as a whole. Instead of determining what value a dependant variable will have (it's state) at a given time, we must determine at what time a dependant variable will enter a given state, namely the state above or below it's current state. This approach may yield results as accurate as a discrete time approach under the condition that the quantum size of the state space is sufficiently small. Figure 10 illustrates the mid-read quantization of a continuous time signal, note that quantum state changes occur when the continuous time signal is ±Q/2 from its current quantum state.

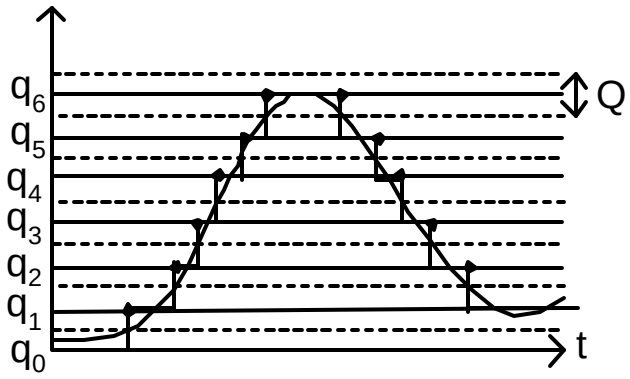


Figure 10 Signal Quantization

The Fourth-order Runge-Kutta integration method presented in the previous section uses a discrete time approach to integrate a first order DE over an interval h to determine y_{n+1}. In order to use the Fourth-order Runge-Kutta method in a quantized state system, equation (4) must be modified to determine h when $y_{n+1} - y_n = \frac{Q}{2}$ where Q is the quantum size. The following is a derivation of the Fourth-order Runge-Kutta method for a quantized state system.

let Q be the quantum size

substitute $k_1 = \frac{Q}{2}, k_2 = \frac{Q}{2}, k_3 = \frac{Q}{2}$ and $k_4 = \frac{Q}{2}$ in (4) to get h_1, h_2, h_3 and h_4

$$h_1 = \frac{\frac{Q}{2}}{f(t_n, y_n)}$$

$$h_2 = \frac{\frac{Q}{2}}{f(t_n + \frac{h_1}{2}, y_n + sign(h_1)\frac{Q}{4})}$$

$$h_3 = \frac{\frac{Q}{2}}{f(t_n + \frac{h_2}{2}, y_n + sign(h_2)\frac{Q}{4})}$$

$$h_4 = \frac{Q}{f(t_n + h_3, y_n + sign(h_3)\frac{Q}{2})}$$

Rearrange sum in (4) and substitute for k_1, k_2, k_3 and k_4

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

$$y_{n+1} - y_n = \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

$$\left|y_{n+1} - y_n\right| = \frac{Q}{2}$$

$$\frac{Q}{2} = \left| h\left[\frac{\frac{Q}{2}}{h_1 6} + \frac{\frac{Q}{2}}{h_2 3} + \frac{\frac{Q}{2}}{h_3 3} + \frac{\frac{Q}{2}}{h_4 6}\right]\right|$$

$$h = \left|\left[\frac{1}{h_1 6} + \frac{1}{h_2 3} + \frac{1}{h_3 3} + \frac{1}{h_4 6}\right]^{-1}\right| \qquad (5)$$

Equation (5) determines at what time relative to the present time the integral of the first order differential equation will enter the quantum state above or below it's current quantum state.

4.6.5 Fourth-order Runge-Kutta Quantized Integrator Model

In order to simulate an ODE system written in sAMS VHDL simultaneous statements, the ODE must first be decomposed into a set of first order differential equations as outlined in (2). Each of these first order differential equations will then be converted into a Fourth-order Runge-Kutta Quantized Integrator model during Model Code and Netlist Generation, and will be instantiated and linked to other Fourth-order Runge-Kutta Quantized Integrators during Model (MA) File Generation.

The conversion process must first determine which quantities and signals are exogenous to the ODE system, and which are endogenous to the ODE system. Endogenous quantities will be the quantity on the left hand side of the simultaneous statement as well as all quantities on the right hand side of the simultaneous statement with the same quantity name as the left hand side quantity. All other quantities or signals will be exogenous. For example the following simultaneous statement describes a first order low-pass filter with input voltage vin and output voltage vout:

```
vout'dot = (1/(R*C))*(vin-vout);
```

In this statement vin is an exogenous quantity, while vout and vout'dot are endogenous quantities.

Once all endogenous and exogenous quantities and signals have been identified, the ODE specified in the simultaneous statement must be decomposed into a set of first order differential equations as outlined in (2). Each of these first order differential equations is then converted directly into a Fourth-order Runge-Kutta Quantized Integrator model. Each Integrator model must have an input port for each exogenous and endogenous quantity or signal on the right hand side of its first order differential equation, and an output port for the integral of the left hand side of its first order differential equation. For example, the low-pass filter above requires only a single integrator, and this integrator has input ports for vin and vout, as well as an output port for vout.

The Fourth-order Runge-Kutta Quantized Integrator model is simple in design. The model buffers each of the input ports by inserting the following code in the models external transition function:

```
if (msg.port()==[signal name])
   {
      [signal buffer name]=(int)msg.value();
   }
```

Following all port buffer code in the integrators external transition function, the model executes the Fourth-order Runge-Kutta method for a quantized state system if the model is in the passive state. The right hand side

of the first order differential equation is converted to C++, substituting the signal buffer name for the signal name, and multiplying this buffer by the quantum size. The following is the Fourth-order Runge-kutta method code for the low-pass filter presented above:

```
    p1 = (1.0/(C*R))*(_vin*QuantumSize - (_vout*QuantumSize));
    p2 = (1.0/(C*R))*(_vin*QuantumSize - (_vout*QuantumSize +
sign(p1)*(HalfQuantumSize/2.0)));
    p3 = (1.0/(C*R))*(_vin*QuantumSize - (_vout*QuantumSize +
sign(p2)*(HalfQuantumSize/2.0)));
    p4 = (1.0/(C*R))*(_vin*QuantumSize - (_vout*QuantumSize +
sign(p2)*(HalfQuantumSize)));

    h1 = HalfQuantumSize / p1;
    h2 = HalfQuantumSize / p2;
    h3 = HalfQuantumSize / p3;
    h4 = HalfQuantumSize / p4;

    h = 1.0/(1.0/(6.0*h1) + 1.0/(3.0*h2) + 1.0/(3.0*h3) + 1.0/(6.0*h4));
```

The model then transitions to the active state for a time determined by h, which is calculated as in (5). The output function simply outputs the current state of the output buffer plus or minus one, plus one if the slope over the interval was positive, minus one if the slope over the interval was negative. The internal transition function similarly increments/decrements the state of the output buffer depending on the slope over the interval and then sends the model into the passive state.

During Model (MA) File Generation each of the integrator models converted during Model Code and Netlist Generation, are instantiated and

linked together. For each Integrator model instance, each port that represents a given endogenous quantity in the simultaneous statement is linked to all ports that represent that same quantity on itself and on all other Integrator model instances. All exogenous quantity and signal input ports are linked to their respective output ports on a process, component or signal model.

Conclusions and Recommendations

To facilitate simulation of mixed signal HDL models within a DEVS simulator, generic DEVS models and conversion procedures were required. These models and conversion procedures were designed for a subset of VHDL-AMS named sAMS-VHDL and targeted toward the CD++ DEVS simulation toolkit. Hierarchical models written in sAMS-VHDL that utilize Processes, Signals and Simultaneous statements may be simulated in CD++ by elaborating the model, and converting the model hierarchy into an equivalent CD++ model that is composed of Process, Signal and Integrator models.

sAMS-VHDL is limited, and does not support many of the advanced features of VHDL or VHDL-AMS. Moving forward, it is recommended that amendments be made to sAMS-VHDL, and additional models and conversion procedures be developed. Type definition, generate blocks and signal attributes would be useful additions to the sAMS-VHDL language as they would ease model definition. The modularity of the CD++ models developed for this project will facilitate integration of new models in the future.

If mixed signal simulation work is going to continue within the CD++ toolkit, a time management protocol will have to be developed. Since CD++ treats time as integer values, the simulation models developed in this project must scale time by a constant factor to allow simulations to run correctly. The

choice of this constant scalar is dependent on the rates of change involved in the analog simulation, and must be chosen carefully. A time management protocol integrated into CD++ could remedy this problem.

References

[1] Sumit Ghosh and Norbert Giambiasi, *"Breakthrough in Modeling and Simulation of Mixed-Signal Electronic Designs in nVHDL"*, Modeling and Simulation, May 2001.

[2] Press, Teukolsky, Vetterling and Flannery, *Numerical Recipes in C: The Art of Scientific Computing. Cambridge:* Cambridge University Press, 1988-1992.

[3] Kloos and Breuer, Ed., *Formal Semantics for VHDL*. Dordrecht: Kluwer Academic Publishers, 1995.

[4] Ashenden, *The Designer's Guide to VHDL*. San Francisco: Morgan Kaufmann Publichers, 1996.

[5] Stroustrup, *The C++ Programming Language Third Edition*. Reading: Addison-Wesley 1997.

[6] Skahill, *VHDL for Programmable Logic*. Menlo Park: Addison-Wesley 1996.

[7] Christen et al., *"DAC'99 VHDL-AMS Tutorial"* presented at 36th Design Automation Conference, New Orleans, June 21-25, 1999.

[8] Rodriguez and Wainer, *CD++ User's Guide*. Universidad de Buenos Aires, Argentina, 1999.

[9] *IEEE 1076 Standard VHDL Language Reference Manual,* Design Automation Standards Committe (DASC). Piscataway: IEEE-SA Standards Board, 2000.

Appendix A

This appendix contains the CD++ models and simulation output for a number of test sAMS VHDL models.

Clock Generator

sAMS VHDL for Clock Generator

```
entity clock is
    port(clk : out std_logic);
end entity clock;

architecture top of clock is
    signal temp;
begin
    inv: process (temp) is
    begin
        temp<=not temp after 10ns;
    end process clk_process;

end architecture rtl;
```

CD++ model for Clock Generator

[top]

components : inv@Process_Inv temp@Signal

out : clk

Link : out@temp in@inv

Link : out@inv in@temp

Link : out@temp out

[temp]

Transport_Delay : 00:00:00:010

Simulation Output for Clock Generator

00:00:00:010 out 0

00:00:00:020 out 1

00:00:00:030 out 0

00:00:00:040 out 1

00:00:00:050 out 0

00:00:00:060 out 1

00:00:00:070 out 0

00:00:00:080 out 1

00:00:00:090 out 0

00:00:00:100 out 1

Four-Bit Counter

sAMS-VHDL for Four-Bit Counter

```
entity 4_counter is
    port(bo0, bo1, bo2, bo3 : out std_logic);
end entity 4_count;

architecture top of 4_counter is
    signal b1,b2,b3,b4,clk : std_logic;
begin

    clock: entity clock  -- from example of Clock Generator
    port map (
        clk=>clk
    );

    4count: process (clk)
    begin

        if(rising_edge(clk))
            b1<= not b1;
            b2<= b2 xor b1;
            b3<= b3 xor (b2 and b1);
            b4<= b4 xor (b3 and b2 and b1);
        end if;
    end process 4count;

    bo0<=b1;
    bo1<=b2;
    bo2<=b3;
    bo3<=b4;

end architecture 4_count;
```

CD++ Model for Four-Bit Counter

[top]

components : 4count@Process_4_Counter b1@Signal b2@Signal b3@Signal b4@Signal clock

out : bo1 bo2 bo3 bo4

link : out@clock clk@4count

link : q1@4count in@b1

link : q2@4count in@b2

link : q3@4count in@b3

link : q4@4count in@b4

link : out@b1 d1@4count

link : out@b2 d2@4count

link : out@b3 d3@4count

link : out@b4 d4@4count

link : out@b1 bo1

link : out@b2 bo2

link : out@b3 bo3

link : out@b4 bo4

[clock]

components : inv@Process_Inv sig1@Signal

out : out

Link : out@sig1 in@inv

Link : out@inv in@sig1

Link : out@sig1 out

[b1]

Transport_Delay : 00:00:00:000

[b2]

Transport_Delay : 00:00:00:000

[b3]

Transport_Delay : 00:00:00:000

[b4]

Transport_Delay : 00:00:00:000

[sig1]

Transport_Delay : 00:00:00:010

Simulation Results for Four-Bit Counter

00:00:00:000 bo1 0

00:00:00:000 bo2 0

00:00:00:000 bo3 0

00:00:00:000 bo4 0

00:00:00:020 bo1 1

00:00:00:020 bo2 0

00:00:00:020 bo3 0

00:00:00:020 bo4 0

00:00:00:040 bo1 0

00:00:00:040 bo2 1

00:00:00:040 bo3 0

00:00:00:040 bo4 0

00:00:00:060 bo1 1

00:00:00:060 bo2 1

00:00:00:060 bo3 0

00:00:00:060 bo4 0

00:00:00:080 bo1 0

00:00:00:080 bo2 0

00:00:00:080 bo3 1

00:00:00:080 bo4 0

00:00:00:100 bo1 1

00:00:00:100 bo2 0

00:00:00:100 bo3 1

Low-pass Filter

sAMS-VHDL for Low-pass Filter

```
entity LPF is
        port (
         terminal tout, tgnd: electrical
        );
end entity LFP;

architecture top of LPF is
    signal clk : std_logic;
    signal vin : std_logic;
    quantity vout across tout to tgnd;

begin
    vout'dot = (1/(R*C))*(vin-vout);

clk: entity clk
port map (clk=>clk
);

vin<=clk;

end architecture top;
```

CD++ Model for Low-pass Filter

```
[top]
components : int@rkIntegModel clock
out : clk y

Link : y@int y
Link : y@int dydt@int
Link : out@clock clk
Link : out@clock vin@int

[int]
y0 : 0
dydt0 : 0
C : 1.0E-6
R : 1000

[clock]
components : inv@Process_Inv sig1@Signal qm@QuantumMultiply
out : out
Link : out@sig1 in@inv
Link : out@inv  in@sig1
Link : out@sig1 in@qm
Link : out@qm   out

[sig1]
Transport_Delay : 00:00:1:000
```

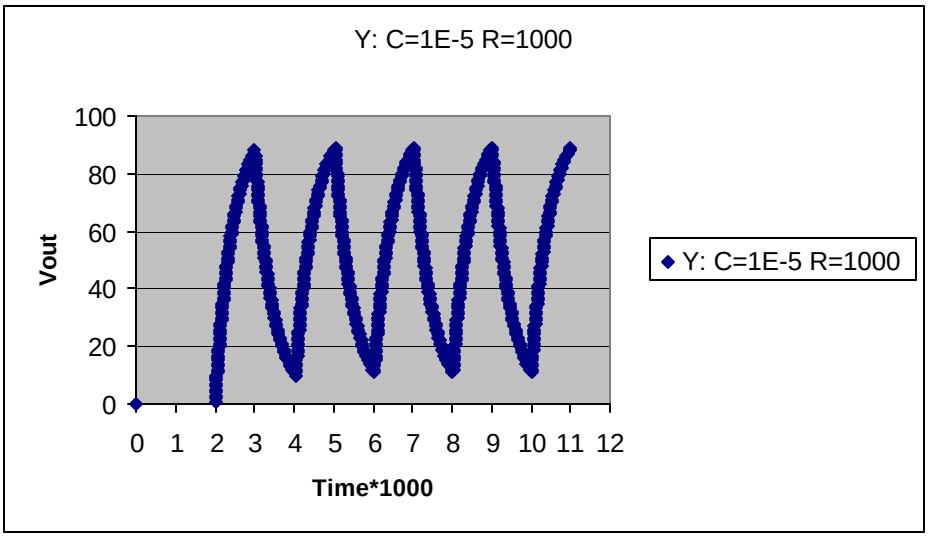
[qm]

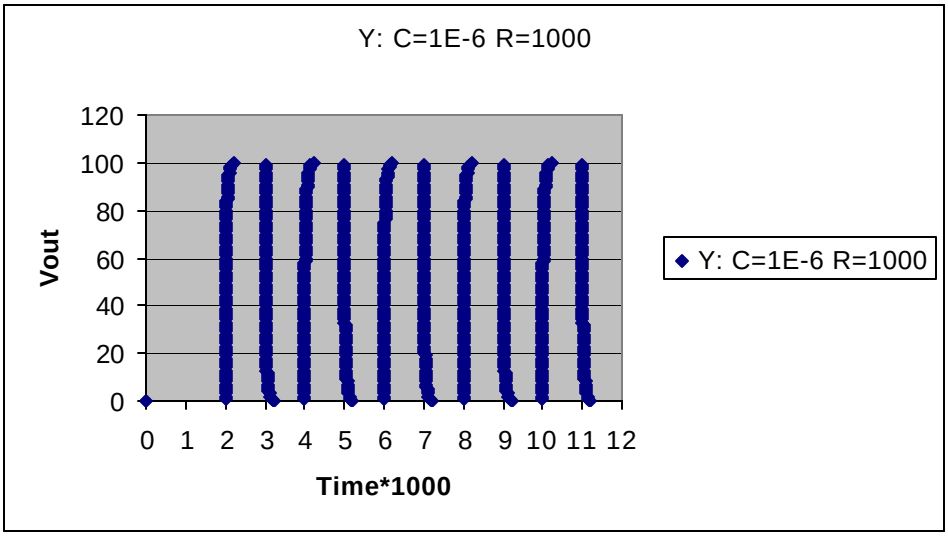
Transport_Delay : 00:00:00:000

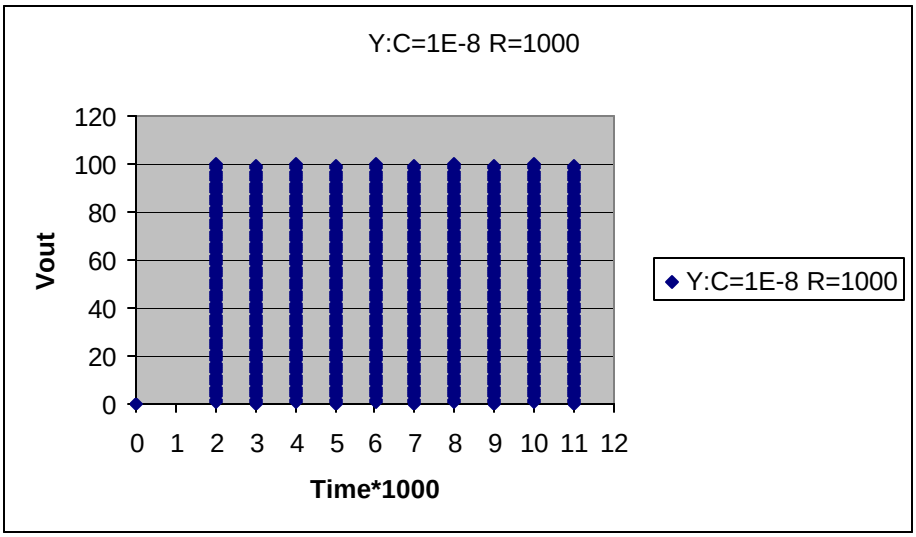
Attenuation : 100

Simulation Results Low-Pass Filter

Simulation results have been graphed in excel below:






Y:C=1E-8 R=1000
120
100
80
60
40
20
0
Vout
0 1 2 3 4 5 6 7 8 9 10 11 12
Time*1000
Y:C=1E-8 R=1000