Mixed-signal modeling using Simulink based-C

Shoufeng Mu
Qualcomm Inc
4243 Campus Point Ct.
(858)651-7600
smu@qualcomm.com

Michael Laisne
Qualcomm Inc
4243 Campus Point Ct.
(858)651-4164
mlaisne@qualcomm.com

ABSTRACT

This paper presents a top-down mixed-signal modeling approach that relies on Simulink converted-C modeling. This modeling flow is divided into four fundamental operations:

- Build the behavioral model using Simulink.
- Generate the portable stand-alone C code using RTW.
- Use FLI to link the generated C code with HDL.
- Write HDL wrappers.

The resulting discrete-time C-based HDL model can seamlessly integrate with the digital and mixed-signal simulation environments without any special requirements from the simulator. With this approach we can truly achieve chip level or system level mixed-signal verification and analog vector generation.

1. INTRODUCTION

With the ever increasing complexity in highly integrated mixed-signal *system on a Chip* (SOC) or *system in a package* (SIP) solutions, accurate mixed-signal simulation and test vector generation have become a challenging task in modern designs. A major contributor to this complexity is the different design methodologies required for designing its digital and analog components. These are important considerations because of the importance of thorough verification prior to tapeout, meeting important time to market requirements, and managing yield with well controlled tests once the device is in production. Traditional strategies involve partitioning the ASIC at the Analog-Digital boundary, and creating separate simulation environments. However these strategies typically leave holes in testing the signal interactions between the analog and digital domains. In order to achieve true full chip mixed-signal functional verification, or generate mixed-signal test vectors for SOC or SIP, there is an urgent need to build an accurate behavioral model of the analog blocks, which can seamlessly integrate with the digital portion of the design. On the other hand, with the increasing number of transistors (millions) on a chip, performance is another bottle neck for the mixed-signal simulation. Currently there is no industry standard methodology and approach for handling such issues arising in mixed signal designs. SPICE-like mixed-signal simulations, although very accurate, are excruciatingly slow and often impractical, especially for large circuits.

This paper introduces a new approach using Simulink[1] based C models for the analog and mixed-signal elements of the design to achieve reliable, fast and accurate chip level or system level mixed-signal simulation. The Simulink based C model is a discrete model that can interface with the design database in VHDL through an industry standard FLI (foreign Language Interface) [2] or Verilog through PLI (Program Language Interface) [2]. The Simulink model can be simulated with digital simulators such as Modelsim, Verilog XL, NC Verilog, etc. or mixed-signal simulators such as Advance-MS from Mentor Graphics [3] or Discovery AMS from Synopsis [4]. This new approach is different from traditional C modeling which requires developing the C code by hand. Manual coding is error prone, has a long development cycle and is hard to maintain. The Simulink based C model approach is also different from HDL mixed-signal modeling, such as VHDL-ams, Verilog-ams [5] and Verilog-A [6,7]. These are code based modeling languages, and require dedicated mixed-signal simulators, such as Advanced-MS or Nanosim. Another advantage of the proposed new modeling approach is that it is a top down. It can provide various degrees of abstraction in selecting the appropriate model for the application. At the end of the design cycle, this high abstraction-level behavioral model is verified against the analog transistor level block to ensure functional equivalence. The Simulink model, therefore, offers a consistent top down development methodology through the product development cycle.

Another important aspect of this Simulink based C approach is that the model is a discrete time model. Discrete-time behaviors are generally expressed as logical Boolean equations or as communicating processes that are triggered by events, while continuous-time behaviors are generally expressed as differential algebraic equations. On one hand, with this discrete C-model, simulation speed increased 10 to 100 times over RTL design or SPICE[8] and this model is reasonably accurate at the architecture level. For example, a sigma-delta converter model will share frequency characteristic similar to its transistor design. On the other hand, this discrete model is an ideal functional model and is not intended to capture behavior such as temperature coefficients, bias current sensitivity, resistor noise or other circuit characteristics.

Simulink based C modeling flow can be divided into the four fundamental steps:

- Build Simulink model with Real-Time Workshop (RTW)[8] constraints
- Generate stand-alone C code using RTW
- Modify C code to interface with VHDL or Verilog
- Write HDL wrappers.

Each step will be discussed in details in the following sections. Examples of behavioral models of RC filter, Sigma-delta A2D converter and synchronization block will be given also.

2. MIXED-SIGNAL MODELING USING SIMULINK

Simulink is a program that runs as a companion to MATLAB[1]. These programs are developed and marketed by the MathWorks,

Inc. Simulink and MATLAB form a package that serves as a vehicle for modeling dynamic systems. Simulink provides a graphical user interface (GUI) that is used for building block diagrams, performing simulations, as well as analyzing results. In Simulink, models are hierarchical, and models can be discrete, continuous or hybrid. For a mixed-signal model the Simulink model will be hybrid.

In order to convert the mixed-signal Simulink model to C code smoothly using RTW, the Simulink model should be built with the following constraints:

1) The following blocks are supported by RTW, and should not be used in the model:
   a. Matlab FCN
   b. Algebraic loops
   c. Simulink s-function
   d. Variable step solver.
2) The models should be discrete. If the model is built in continuous time domain, such as sample-hold circuit, integrator, or continuous time transfer function, use discretizer in Simulink to discretize the model.
3) All input values that can change should be ports, not parameters. Parameters can sit in a modelname_init.m file.
4) In order to simplify the c code, all input and output ports should be of type "double". This will translate into a "double" in C, and into type "real" in VHDL. This may not be the most efficient way to handle the data when crossing between different domains, but it simplifies the automation scripts.

3. SIMULINK MODEL OF BBRX

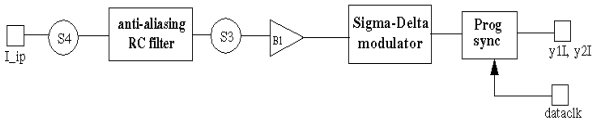

Figure 1. Top-level block diagram of a single-channel BBRX

Figure 1 is a simplified example of the top-level Broad-band Receiver (BBRX) block diagram which normally implemented in the analog die or mixed-signal section of a design. This block diagram is only used to demonstrate the methodology of mixed-signal modeling, not the functionalities of the Broad-Band Receiver. The incoming analog baseband signal I_ip passes through an analog RC filter and, optionally, through a gain buffer. Then the signal passes to a Sigma-delta A2D converter. The modulator converts the resulting signals into digital bit-streams (single bit or multiple bits); in this example, two bit-streams - Y1 and Y2 are used. The data is then passed through a synchronization block and then output to the digital block or digital chip. This is a typical of a mixed-signal design. It has a continuous time block (the RC filter), a mixed-signal block (the Sigma-Delta modulator) and a digital block (the synchronization unit). In this section the BBRX is used as an example to demonstrate the procedures that one must follow to build the mixed-signal models using Simulink.

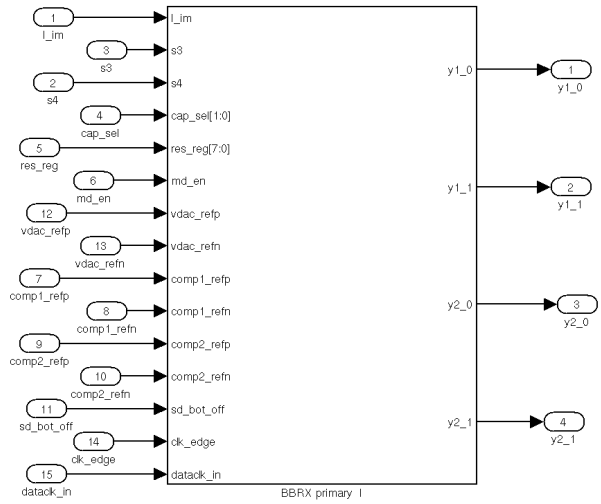

Figure 2 BBRX top-level port list

The top-level port list of the BBRX is shown in Figure 2. The top-level port list of the Simulink model should match the top level port list of the design block. At this level, "match" means the port names and dimensions should match the schematic design. The data type of the port in Simulink should be double; this will convert to double in C and real in VHDL through the FLI (Data type conversion will be discussed in wrapper section). Additionally, the appropriate sampling time will need to be defined for the ports. For the input ports, a reasonable sampling time should be defined. For the output ports sampling time can be inherited from the block driving the port or redefined.

Figure 3 shows the top-level Simulink model of the BBRX block. It consists of an anti-aliasing RC filter, Sigma-Delta modulator (SD modulator), synchronization block and other control logic. Details of the RC filter, Sigma-delta modulator and synchronization blocks will be discussed in the following sections.

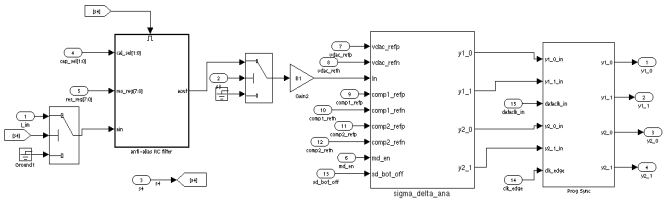

Figure 3 Top-level model of BBRX

3.1 Simulink Model of RC Filter

The schematic of a first order RC filter is shown in Figure 4. In this case, the single-ended RC filter consists of a variable input resistor (R_{bias}), a feedback resistor (R_{feed}), a variable pole capacitor (C_{pole}), and an operational transconductance amplifier (OTA). The OTA gives transconductance amplification. That is, it takes the voltage difference from its inputs and converts it to a proportional current. The coefficient of proportionality, defined as the transconductance of the OTA, is $g_{m;}$ the transfer function of the low-pass filter can be deducted as:

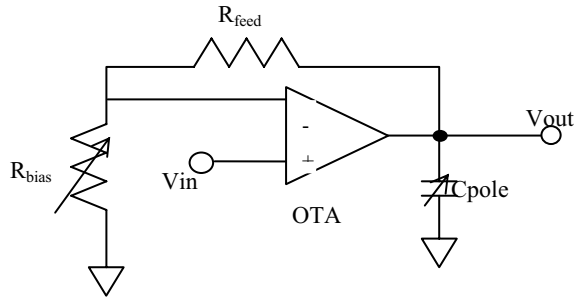

Figure 4 Block diagram of a single-ended RC filter

$$H(s) = \frac{V_{out}}{V_{in}} = \frac{g_m(R_{bias} + R_{feed})}{1 + g_m R_{bias}} \frac{1}{1 + \frac{(R_{bias} + R_{feed})C_{pole}S}{1 + g_m R_{bias}}}$$

$$= g_m R_{eq} \frac{1}{1 + R_{eq} C_{pole} S}$$

Where

$$R_{eq} = \frac{R_{bias} + R_{feed}}{1 + g_m R_{bias}}$$

R_{bias} is a programmable resistor, and its value can be programmed through an 16 bit register. C_{pole} is a programmable capacitor; the user can select eight different configurations using 8 capacitors with values of 1pf, 2pf, …, 7pf and 8pf through c_ctl_reg[2:0].

Figure 5 shows the implementation of the RC filter, which includes the continuous time integrator. In this model, both R_{bias} and C_{pole} are programmable. This model is a continuous time domain model which needs to be discretized when converted to c code. In Simulink, under tools, there is a utility called discretizer. The user can use this discretizer to discretize the continuous time domain model. The integration timestep is very important for the conversion. If the timestep is too big, the discretized model may not be able to represent the design to the required resolution. On the other hand, if the timestep is too small, it will affect the CPU time required for simulation.

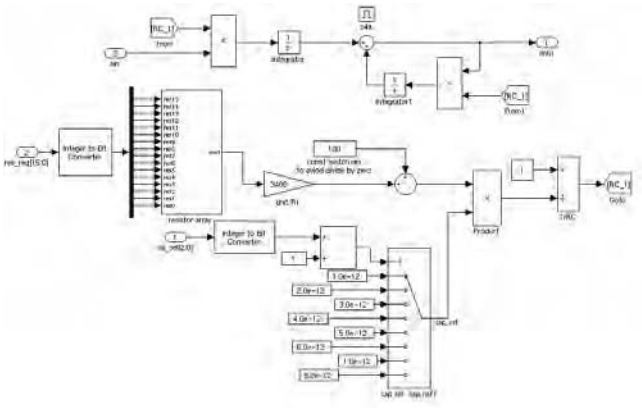

Figure 5 Model of RC filter in Simulink

Table 1 shows the relationship between the timestep, performance, and simulation time. These simulations were performed on a 32 bit Linux machine with RC time constant T_{RC} =1.9e-7s. From the table we can see, when the timestep is too big, the algorithm wouldn't be able to converge. When the timestep< $0.1T_{RC}$, if reduces the time steps, it greatly increases the CPU time, and the performance of model doesn't improve much. Therefore, user should choose the integration timestep wisely.

Integration timestep (s)	CPU time(s)	SNR(signal noise ratio in DB)
6.0e-8		Not converge
5.0e-8	151	-171
4.0e-8	166	-171.5
2.0e-8	194	-172.5
1.0e-8	226	-172.5
5.0e-9	331	-172.8

Table 1 Relationship between integration timestep, performance and CPU time

3.2 Sigma-delta modulator

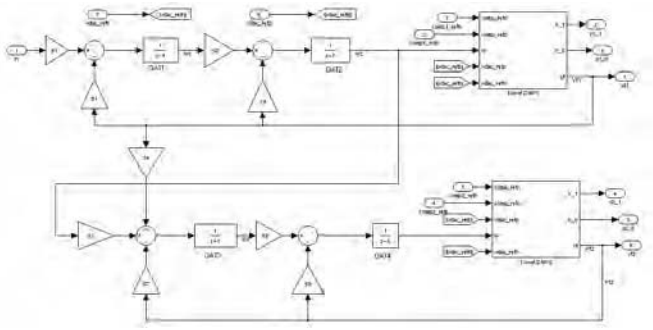

Figure 6 Simulink model of 4-level 4th order ΣΔ modulator

Figure 6 shows the Simulink model of a generic 4th order 4 level ΣΔ A2D. There are two 4-level comparators and two A2D converters. In this model, the sampling time of the delay block should be based on the over sampling clock frequency. The reference voltages for comparator 1, comparator 2, DAC 1 and DAC 2 are programmable, and are generated from the reference block.

3.3 Signal Synchronization

Figure 7 shows the Simulink model of the synchronization block. Dataclk, an input to this block, is used to synchronize the bit stream from Sigma-delta modulator. The synchronized data be will sent to the digital MSM chip. There is another control signal, clk_edge. With this signal the user can choose the digital data synchronized to the rising edge or the falling edge of dataclk.

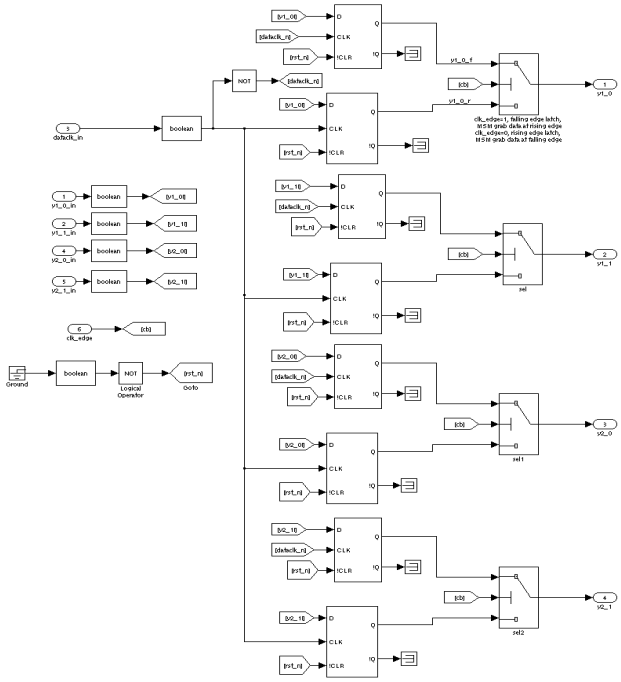

Figure 7 Simulink model of the synchronization block

4. CONVERT SIMULINK MODEL TO C

Real-Time Workshop is an application of Simulink. It generates optimized, portable and customized ANSI C code from Simulink models. It automatically builds programs that execute in Real-Time or Stand-alone non-real-time simulation. In this paper, only Stand-alone non-real-time C code will be used.

The following configuration parameters should be set correctly before generating the C code:

1) Simulation start time and stop time. Normally the simulation start time is set to 0, stop time to 'inf' for infinite. The end of the simulation will be controlled from outside the c environment.
2) The solver option. The solver type should be set to fixed step discrete. Variable step solver is allowed in Simulink, but not supported by RTW.
3) Step size option. For C code generation, the step size can be set to 'auto'. When customized C code, users can decide the step size through FLI interface. For the simulation case, the step size should be set to a reasonable value, such as half of the fast clock period.
4) Solver mode. The solver model should be set to single-tasking. The C code generated from single-tasking mode is much simpler than that from the multi-tasking mode.
5) Workspace IO option. Make sure the "save output" and "save time" options are off. These options log both the results and the timestamp respectively. During C code generation, these options should be off since there is no valid input to the model.
6) System target file. Select the Generic Real-Time Target (grt.tlc). This will generate Stand-alone non-real-time C code.

Parameters in the model should be initialized if there are any. Then it is ready for the code generation. Code generation is simple. You can issue Matlab command make_rtw to generate the C code. Once the code generation is complete the following files will be generated in the directory ./X_grt_rtw (where 'X', the model name, is embedded in both the directory and most of its associated files):

- **X.c**: the stand alone C code that implements the model
- **X_data.c**: initial parameter values used by the model
- **X.h**: an include header file containing definitions for parameters and state variables
- **X_types.h**: a file containing forward declarations of data types used in the code.
- **X_private.h**: a header file containing common include definitions
- **Rtmodel.h**: a master header file for including generated code in the static main program.

There is another file used in the c code compilation to create the executable, grt_main.c. This file is located in the directory /installation/mathworks/version/rtw/c/grt/ (/installation is the directory where Mathworks products are installed). The file, grt_main.c, contains the main() C routine that is used to initialize all the variables, allocate memory, invoke the functional routine that implements the model and terminate the program. Any display statements, interrupt handling with external clock or timing mechanism must be added in this file. The rest of the files remain untouched except the header file.

5. CUSTOMIZED C CODE

The C code generated in step 4 can interface with VHDL or Verilog. For mixed-signal designs, Verilog 1995 doesn't have real data type and cannot handle analog signals. Therefore, for mixed-signal modeling, VHDL is chose to interface with C code. But this modeling flow can be used for digital modeling, including DSP, ALU, etc. Under these circumstances the C code can interface with either VHDL or Verilog. In this section, we mainly discuss how to modify the C code to interface with VHDL.

The grt_main.c file needs to be modified to interface with VHDL. The interface will vary depending on the simulator used. Here the Modeltech FLI interface is shown as an example. Figure 8 illustrates the philosophy behind partitioning the C main function and the VHDL entity and architecture. The "x_init" in the VHDL architecture is the name of the initialization function in the main.c file. The "x.so" is the name of the shared object file generated from the C code.

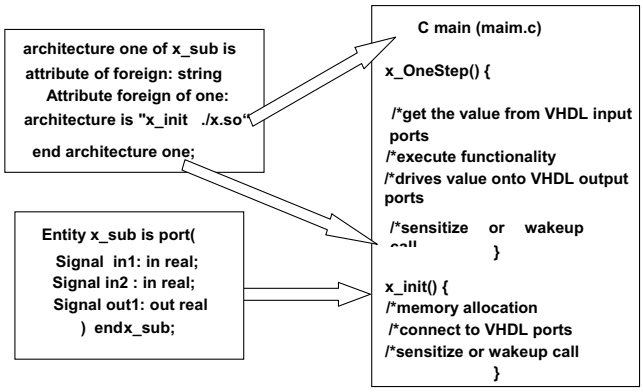

Figure 8 Partition of C main () function

```
void init(mtiRegionIdT region, char *param, mtiInterfaceListT, mtiInterfaceListT *generics,
mtiInterfaceListT *ports)
        {   Xinst_rec    *ip_X;
              ...
              /* memory allocation and function callback */
            ip_X = (Xinst_rec *)mti_Malloc(sizeof(Xinst_rec));
            mti_AddRestartCB(mti_Free, ip_X);
      /* create process and save the handles to the signals in the port list */
          ip_X->procid    =    mti_CreateProcess("p1X",    OneStep,
      in  ip_X->In1 = mti_FindPort(ports, "in1");
          ip_X->In2     =     mti_FindPort(ports,
        "in2");
          mti_ScheduleWakeup(ip_X->procid, 1000);
            ...
            /* create drivers */
      ip_X->Out1 = mti_CreateDriver(mti_FindPort(ports,"Out1"));
      mti_SetDriverOwner(ip_X->Out1, ip_X->procid);
```

Figure 9 Example of init( ) routine

```
static void OneStep(void *param)
{  inst_rec *ip=param;
     ...
       /* get the inputs from VHDL port*/
    mti_GetSignalValueIndirect(ip->In1 ,&(rtU.In1) );
    mti_GetSignalValueIndirect(ip->In2 ,&(rtU.In2) );
     ...
   MdlOutputs(0);   /* execute the C function */
   /* drive the value to the VHDL output port */
    mti_ScheduleDriver(ip->Out1,(int)&(rtY.Out1),0,   MTI_TRANSPORT);
   mti_ScheduleWakeup(ip->procid,1000);/*wakeup call*/
} /* end OneStep */
```

Figure 10 Example of OneStep() routine

Basically the main() function has to be modified with the functions like init() and OneStep(). The init() function has to be changed such that the the VHDL input and output ports connect to the Simulink model. The OneStep() function has to be modified with the addition of FLI commands to read in the VHDL values and to drive the c outputs back to the VHDL output ports. Figure 9 shows an example of the x_init() routine. This routine is the entry point into the foreign C model. This initialization function typically allocates memory, saves the handles to the signals in the port list, creates drivers on the port that will be driven, creates one or more processes (a C function that can be called when a signal changes) and sensitizes each process to a list of signals. In this example, it uses schedule wakeup instead of sensitize call. Figure 10 gives an example of the OneStep( ) routine. This routine reads in the VHDL value (input ports in VHDL entity) through FLI command mti_GetSignalValue (for any scalar signals except real or time) or mti_GetSignalValueIndirect (for real signals or time), and pass the value to the input of foreign C-model. The input port of the foreign C model is under rtU data structure, and can be accessed as rtU.in1, rtU.in2, etc. Then rt_OneStep() routine executes the C routine (MdlOutputs(0)), and drives the C output back to VHDL output ports. The output port of the foreign C model is under rtY data structure, and can be accessed as rtY.out1.

6. VHDL WRAPPERS

There are two wrappers for the c model. The "inner" wrapper has only real inputs and outputs, and its architecture calls the c library with a foreign attribute. The "outer" wrapper takes care of the conversion of the data format, as well as separating the bits in a digital bus. The name and data type of input/output ports of the "outer" wrapper must match the pin name of the analog schematic exactly. It can be useful for the "inner" layer to have a different number of inputs/output pins for debugging purpose; this is allowable

Figure 11 is an example of the inner wrapper. In this example, the data types of the input and output ports are real, and its architecture calls the c library with a foreign attribute.

```
entity x_sub is
  port (signal PortNameIn1      :  in   real;
        signal PortNameIn2      :  in   real;
        signal PortNameIn3      :  in   real;
        signal PortNameOut1     :  out  real;
        signal PortNameOut2     :  out  real );
end x_sub;
architecture one of x_sub is
   attribute foreign : string;
 attribute foreign of one : architecture is
"x_init ./x.so";
   end architecture one;
```

Figure 11 Example of Inner wrapper

```
ENTITY ModelTop IS
          port (signal PortNameIn1      : in   real;
            signal PortNameIn2    : in   std_logic;
            signal PortNameIn3a   : in   std_logic;
            signal PortNameIn3b   : in   std_logic;
            signal PortNameIOut1  : out  real;
            signal PortNameOut2   : out  std_logic );
        END  ModelTop;
        ARCHITECTURE one OF ModelTop IS
        --internal signals for data type conversion
       signal PortNameIn2_real       : real : = 0.0;
      Signal PortNameIn3_vector:unsigned(1 downto 0) := "00";
        signal    PortNameIn3_real      : real := 0.0;
        signal    PortNameOut2_real     : real;

        BEGIN
          PortNameIn2_real <= to_real(PortNameIn2     );
          PortNameIn3_vector<= PortNameIn3a & PortNameIn3b;
          PortNameIn3_real<= to_real(PortNameIn3_vector);
          PortNameOut2  <= to_std_logic(PortNameOut2_real);
           …  (port map )
        END one;
```

Figure 12 Example of outer wrapper

Figure 12 shows an example of the outer wrappers. For the outer wrapper, the data type of input port PortNameIn2 is std_logic, and its corresponding inner wrapper type is real. This requires a data type conversion – converting the outer wrapper's std_logic to real. The input ports PortNameIn3a and PortNameIn3b in the outer wrapper are mapped to the input port PortNameIn3 in the inner wrapper. In this case, the user needs to combine the input port PortNameIn3a and PortNameIn3b to a std_logic_vector, and convert the std_logic_vector to a real signal, then map this real signal to PortnameIn3 in the inner wrapper. For the output port PortNameOut2, users need to convert the real data type to std_logic by the function call to_std_logic.

7. AUTOMATION SCRIPTS

Customizing C code and writing VHDL wrapper manually seems time consuming and tedious. This process can be automated by shell scripts and Perl scripts. The following section outlines the major steps of the automation scripts.

1) Perl scripts to customize the C code generated by RTW to interface with VHDL. There are four major steps for this script. First, create a customize template grt_main.c file based on the grt_main.c file provided by Mathworks. This template file should have the proper format for FLI/PLI interface. Next, get the input/output port names and types of the generated C code

(Simulink model). This information is available in X.h file which was generated in step 4. Third, connect the VHDL inputs/outputs to the Simulink model's input and output ports respectively in the Init() routine. If the names of the Simulink model match the names of the VHDL wrapper, it will simplify the Perl script. The last step, add the FLI commands to read in the VHDL values to the c code and drive the C output back to the VHDL output ports in OneStep() routine, and add the sensitize command or schedule wakeup command in the Init() and OneStep() routine.

2) Perl scripts for VHDL inner and outer wrapper. For the inner wrapper, the input/output port names should exactly match its corresponding name in the Simulink model, and it is available in X.h file. For the outer wrapper, there are various ways to generate the outer wrapper. One easiest way is to annotate the inner wrapper (for VHDL, use "—"which will be interpreted as comment). This annotation specifies the new port name and data type in the outer wrapper.

3) Perl script to generate the C compile command which can be used to generate C executable for various platforms. Currently we support Solaris, HP, Linux.

A shell script is used to link the scripts 1) to 3) together. Figure 13 is shown the structure of the convert_c.sh script.

```
#!/bin/sh
#Usage: convert_c.sh diary modelname no_inst wakeuptime
#arch[sol| hp| linux | linux64] modeltech_version

diary=$1; shift; m=$1; shift;
no_inst=$1; shift;
wakeup=$1; shift
arch=$1; shift;
version=$1; shift;
/usr/bin/diary2compile.pl $diary $no_inst $arch $version > ./compile

chmod +x ./compile
/usr/bin/create_main.pl  $m.h  $wakeup $arch $*
 /usr/bin/wrappers.pl $m $no_inst
./compile $arch modelsim
```

Figure 13 an example of the automation scripts

8. INTEGRATION AND SIMULATION

The wrappers of the Simulink based-c models are VHDL codes, therefore the models can be treated as VHDL models from the integration point of view. These models can be integrated with RTL design, gate-level design and mixed-signal design. They can be used with either digital simulators or mixed-signal simulators as long as the simulator supports a c foreign language interface. ModeSim SE is a simulator product from Model Technology that supports VHDL/Verilog mixed-language simulation. We chose Modelsim for our mixed-signal simulation/verification environment due to its full-featured access to Verilog models and VHDL entities, including source code debugging, waveform viewing and hierarchy navigation.

In our verification environment, the behavioral model of the BBRX, Codec and HKADC are integrated with the digital MSM chipset. Figure 14 shows the Modelsim simulation results of the BBRX. At the top of figure 14, it shows the register configuration of BBRX, followed by the analog differential input iip and iin (sinewave). The synchronized 4 bit Sigma-Delta modulator (SD) outputs are shown in Figure 14 followed by the analog inputs. The 4-bit SD outputs pass through a digital filter and the recovered sinewaves are shown at the bottom of Figure 14.

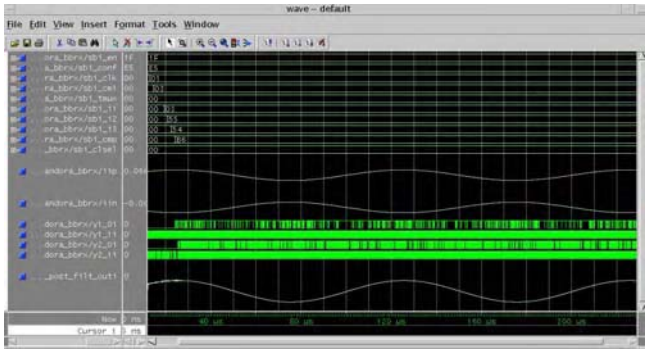

Figure 14 Modelsim Simulation result of BBRX block

9. SUMMARY

This Simulink based C modeling approach has been used successfully in commercial 90 nm WCDMA MSM chip set simulation environments. This method is used to model 12 bit Successive Approximation Housekeeping ADC's, TXDAC's, wide-band CODEC's and Broad Band Receivers, which incorporate anti-aliasing filters, RC filters and Sigma-Delta modulators. With this mixed-signal simulation environment, we're able to achieve full-package (system in a package) mixed-signal simulation and mixed-signal test vector generation. The top down approach to modeling analog portions of the design using Simulink models has proven to be invaluable time to market aid for getting chips reliably from concept to production.

10. REFERENCES

[1] Simulink manual from http://www.mathworks.com/products/simulink/.

[2] ModelSim SE/EE Foreign Language Interface manual.

[3] http://www.mentor.com/products/ic_nanometer_design/news/ams_simulator.cfm.

[4] http://www.electronicstalk.com/news/syn/syn201.html

[5] Francois Pecheux, Christophe Lallement, IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol.24, No.2.

[6] K. S. Kundert and O. Zinke. *The Designer's Guide to Verilog-AMS.* Kluwer Academic Publishers, 2004.

[7] D. Fitzpatrick and I. Miller, *Analog Behavioral Modeling with the Verilog-A Language*, Kluwer Academic Publishers, 1998.

[8] Bill Luo, Jim Lear, *A Unified Functional Verification Approach for Mixed Analog-digital ASIC Designs*, DesignCon, 2003.