Chip-Level Verification of an RF ASIC with CustomSim/VCS

Greg Tumbush, PhD, PE

EM Microelectronic-US, Inc

Colorado Springs, CO, USA

www.emmicroelectronic.com

ABSTRACT

Chip-level AMS verification requires a unique mix of digital methodology with analog expertise. The methodology used at EM Microelectronic to verify our latest RF ASIC utilizes the Synopsys VCS AMS tool combining the CustomSim analog solver with the VCS digital simulator for true chip-level verification. Our methodology features a SystemVerilog testbench wrapping the spice and AMS analog circuits. The chip-level tests are self-checking and executed from the command line allowing the entire test suite to be run as a regression. The SystemVerilog testbench probes currents and voltages directly using the $snps_get_volt and $snps_get_port_current system tasks to ensure specification compliance. The measured voltage/current is passed to a checker function with allowed ranges to account for variability in the measurement and reports pass or fail. The expected passing count for each test is also checked, ensuring that the test completed as expected. The simulations can be run as AMS or digital only through TCL commands to debug the SystemVerilog transactors communicating with the ASIC.

Table of Contents

Table of Figures

1. Introduction

This paper presents the chip-level verification of a mixed signal RF ASIC utilizing Synopsys CustomSim/VCS. The work was completed at EM Microelectronic-US.

The ASICs that EM Microelectronic-US develops are typically very lower power with standby current in the uA range and operating current in the 100's of uA range. Low-power modes are used extensively. Their ASICs are optimized for area and memory is to be kept to a minimum. The ASICs are typically dominated by analog, and the design teams are small, 1 or 2 digital designers, 1 analog designer and a similar number of verification engineers. Their design cycles are short, with concept to tapeout in less than 12 months. Their ASICs must sell in a market that is typified by low cost and high volume.

Chip-level AMS verification requires a unique mix of digital methodology with analog expertise. The methodology used at EM Microelectronic to verify our latest RF ASIC utilizes the Synopsys VCS AMS tool combining the CustomSim analog solver with the VCS digital simulator for true chip-level verification. Our methodology features a SystemVerilog testbench wrapping the spice and AMS analog circuits. The chip-level tests are self-checking and executed from the command line, allowing the entire test suite to be run as a regression. The SystemVerilog testbench probes currents and voltages directly using the *$snps_get_volt* and *$snps_get_port_current* system tasks to ensure specification compliance. The measured voltage/current is passed to a checker function, along with allowed ranges to account for variability in the measurement. In addition to incrementing an error counter, in the case of an error, the checkers also increment a "correct" counter in the case of a correct check. In addition to checking for 0 errors, the expected correct count for each test is also checked, ensuring that the test completed and was checked as expected. Using TCL commands, the simulations can be run as AMS (to verify the chip) or digital (to debug the SystemVerilog transactors communicating with the ASIC.)

1.1 System Description

A block diagram of the RF ASIC is in Figure 1. The analog block contains two oscillators, an always-on slow oscillator and a fast oscillator, which operates only during transmit or receive of RF. The system uses one internal voltage regulator and connections for a 2nd through the V_VDD/BASE/VSS pads. The analog uses a resonator connected through the RES_IN and RES_OUT pads for receiving and transmitting at RF frequencies. The digital is responsible for booting from the Non-Volatile Memory (NVM), Manchester decoding incoming data, Manchester encoding transmit data, and implementing test modes.

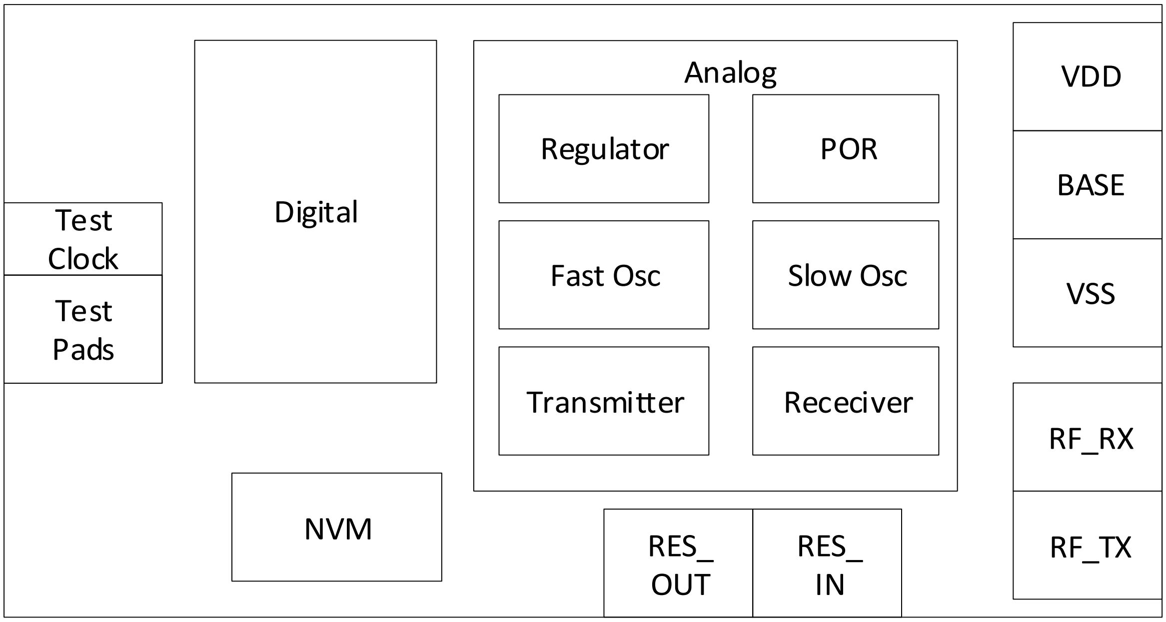

Figure 1: Block Diagram of RF ASIC

2. Methodology

Due to the very long run time of chip-level AMS simulations it is important for all interested parties to agree on what functionality will be tested. The goal is not to verify analog functionality. That is what analog simulations are for. Likewise, there is no reason to verify digital functionality in an AMS simulation. The focus should be on verifying connectivity between digital and analog and device startup.

Another important decision, is at what level will the analog blocks be represented. For example, transistor-level models of oscillators lead to long simulation times due to the continuous and high frequency oscillations. Keeping as many oscillators as possible in the digital domain will result in greatly improved simulation times. Usage of transistor-level memory models may also lead to long simulations times or non-convergence. An RTL-level model of the digital logic will simulate faster than a gate-level representation. For a startup test, though, perhaps all analog blocks and pads will be represented as transistors.

It is not necessary to go through the startup sequence for the voltage regulator, power-on-reset (POR) circuitry, etc. for every test. Instead, use an initial condition to set the regulator output to the expected regulated voltage and toggle POR.

Many tests were required to verify the myriad of test modes. The test modes are entered by a lengthy sequence by the Test Transactor on the Test Bus as shown in Figure 2. Due to the simulation time being dominated by these transactions, over 90% for some scenarios, it is imperative to get them correct prior to running an AMS simulation. Be sure to have a switch to run a quick digital simulation to debug these transactions. All chip-level tests were executed at the command line and were self-checking, a necessity for regression testing.

The goals of the verification methodology are:

- Follow the chip-level test plan
- No hand hacking of netlists, scripts, etc.
- Tests can be run as an unattended regression

- Tests are self-checking as much as possible

To this end the workflow is:

1) Generate a top-level Spectre netlist from the schematic in Cadence Virtuoso
2) Execute a perl script to modify the top-level analog netlist. This includes:
    a. Fix paths to the Physical Design Kit (PDK)
    b. Include the proper NVM content
    c. Add a top-level sub-circuit definition so the top-level netlist can be instantiated in the Spectre wrapper.
    d. Comment out all Spectre simulator options
3) Execute a top level run script with a test name and a seed

3. Environment

The chip-level verification environment is depicted in Figure 2. The Test Transactor puts the ASIC into the desired test modes, sets fields, and reads status. The RF transactor acts as the radio to provide RF communication, albeit at a slower modulation rate than specified. The PnP circuit is used as an external voltage regulator and is specified in the customer's Bill of Materials (BOM). The ability to overdrive the PnP regulator by a DC source, controlled by *vdd_overdrive*, is provided in order to avoid simulating the PnP circuit, thereby improving simulation performance. A spectre wrapper connects all analog components.

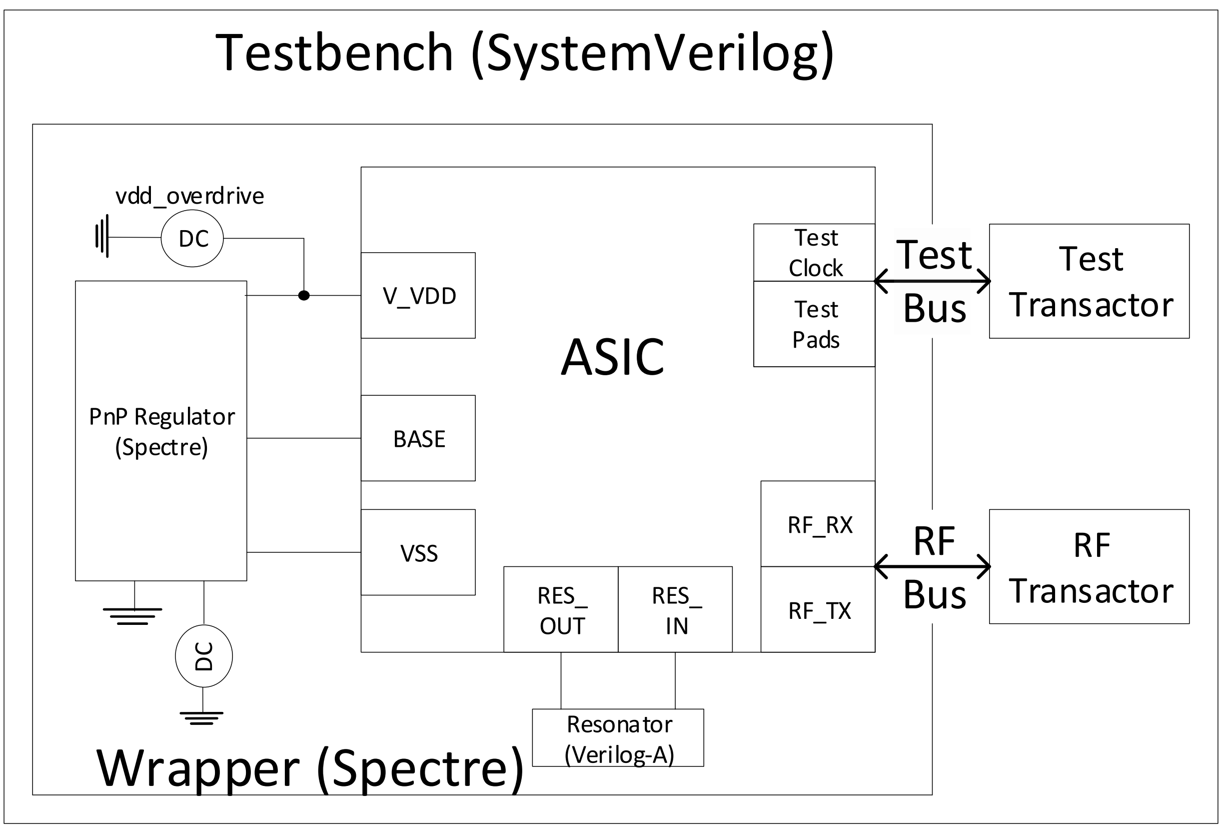

Figure 2: Chip-Level Testbench

A high-level overview of the scripting environment, which will be explained in more detail in later sections, is in Figure 3. The test name and seed are provided to the top level run script, *runVcsAms*, which modifies the top-level analog netlist and builds and executes the VCS command line. The VCS command line specifies the mixed signal simulation control file, by default *vcsAD.init*. The *vcsAD.init* script specifies the characteristics of the digital to analog (D2A) and analog to digital (A2D) converters and builds and executes the XA command line. The CustomSim/VCS simulator uses XA to solve the analog portions of the circuit while VCS solves the digital. The XA command line specifies the CustomSim control file, by default *xa.cfg*. The *xa.cfg* file specifies the simulation level (i.e. speed and model complexity trade-off) and the signals to be probed. At this point the simulation runs, a *vcs.vpd* file is created for viewing digital probes, and the *xa.fsdb* file is created for viewing analog probes.

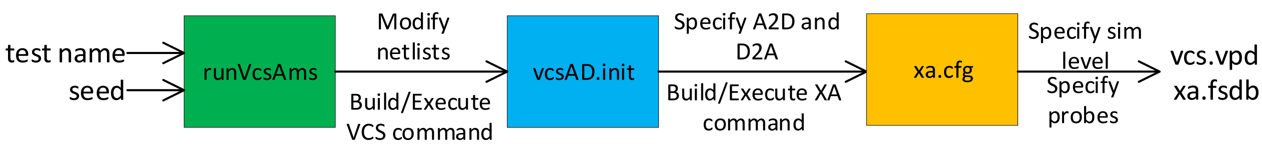

Figure 3: Scripting flow

3.1 Run script

A top-level C-shell (CSH) run script, *runVcsAms,* is used to invoke a simulation. The script only has two options, the seed and test name. These options are passed to the Synopsys VCS command by using the –*pvalue* option as seen in Figure 4. The syntax to specify the testname is a bit tricky because it is desired for the test name to remain a string. The CSH syntax necessitates 3 backward slashes to maintain the quotes.

```
set vcs_command = ($vcs_command -pvalue+chip_top_tb.SEED=$seed);
set vcs_command = ($vcs_command -pvalue+chip_top_tb.TESTNAME=\\\"$test\\\");
```

Figure 4: Specifying the seed and command line to VCS

3.2 Mixed-Signal simulation control file

The mixed-signal simulation control file, by default *vcsAD.init,* serves the following purposes in the chip-level verification environment:

- Specify the bus format of vectors
- Specify the analog solver and options
- Specify the restore file
- Specify the characteristics of the D2A and A2D converters

Each of these tasks are elaborated in the following subsections.

3.2.1 Specify the bus format of vectors

The top-level analog netlist uses the vector format <signal_name>\<index\>, for example, a 2-bit vector called *my_trim* would be represented as the following:

```
my_trim\<1\> my_trim\<0\>
```

To properly interpret these vectors the bus format must be declared in the mixed-signal simulation control file with the following statement:

```
bus_format <%d>;
```

3.2.2 Specify the analog solver and options

The mixed-signal simulation control file uses the *choose* command to specify the analog solver and to pass command line options to the solver. For example, to use the XA solver, as was desired in this project, the following command is used:

```
choose xa -tcl -mt 4 -spectre <spectre file> -c xa.cfg
```

The *–tcl* option specifies that the TCL interpreter should be used for the CustomSim control file, *xa.cfg*, specified by the *–c* option. The *–mt <#>* option specifies the number of CPU cores to use. Note that using extra CPU cores might require extra licenses. The *–spectre <spectre file>* option specifies the name of the input netlist in Spectre format.

3.2.3 Specify the restore file

If a restore file was saved, as will be described in subsection 3.3.3 it can be used to restore the simulation by adding the following option to the XA command line as described in subsection 3.2.2 . Note that even though the *.ic* file is specified the *.ic.sup* file is also needed.

```
-restore <state file>.ic
```

3.2.4 Specify the characteristics of the D2A and A2D converters

The digital logic shown in Figure 1 operates from a regulated voltage, typically 1.65V. All digital outputs should be converted by the D2A conversion to approximately 1.65V for a logic-1. However, when observing the voltage of a logic-1 digital output in a simulation it was found to be 3.3V as shown in Figure 5 where *[digital output]* is the digital signal and [*v(digital output)]* is the analog signal after D2A conversion.

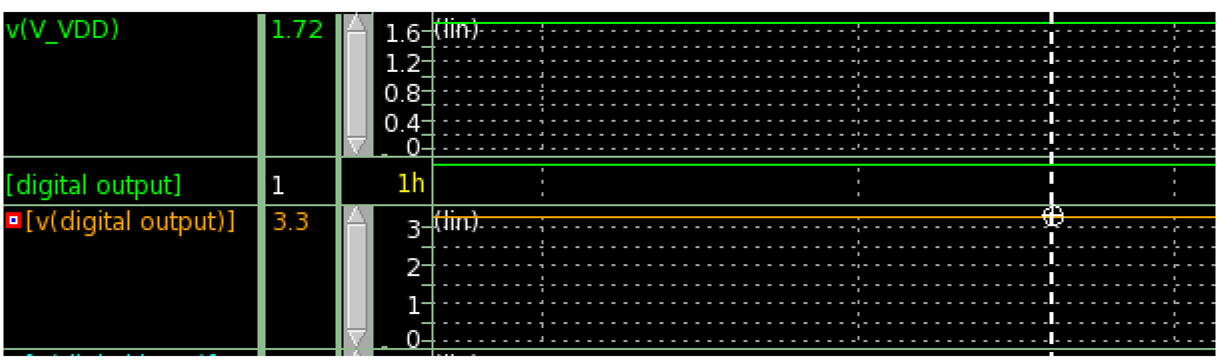

Figure 5: Digital output using defaults for D2A converter

Examination of the *simv.msv/interface_element.rpt* report revealed that 3.3V was chosen as the logic-1 voltage, as seen below, because the CustomSim tool assumes a 3.3V supply if it cannot trace a *d2a* net to an ideal supply

```
d2a hiv=3.3v lov=0v node=<path to digital output>
```

To fix this issue it is necessary to specify *vdd* for the digital output D2A converters in the mixed-signal simulation control file as shown below.

```
d2a vdd=<path to digital vdd pin> vss=<path to digital vss pin> hiv=100%
    lov=0% minv=0.1 minv_analog=0 inst=<path to digital instance> port=*;
```

The *vdd* and *vss* paths specify the reference supply to associate to logic-1 and logic-0 values. In this way, the conversion from logic-1 or logic-0 to voltage, uses the actual regulated supply or ground to the digital. The *hiv* and *lov* parameters specify that a logic-1 is 100% of the *vdd* voltage and a logic-0 is 0% of the *vdd* voltage. The *minv* parameter specifies the *vdd* voltage at which the D2A conversion turns off, in this case 0.1V, saving simulation time. The *minv_analog* parameter specifies the analog output when the D2A converter is off, in this case 0V. After adding this command, the digital output voltage is correct as shown in Figure 6 and the *simv.msv/interface_element.rpt* report is as expected.

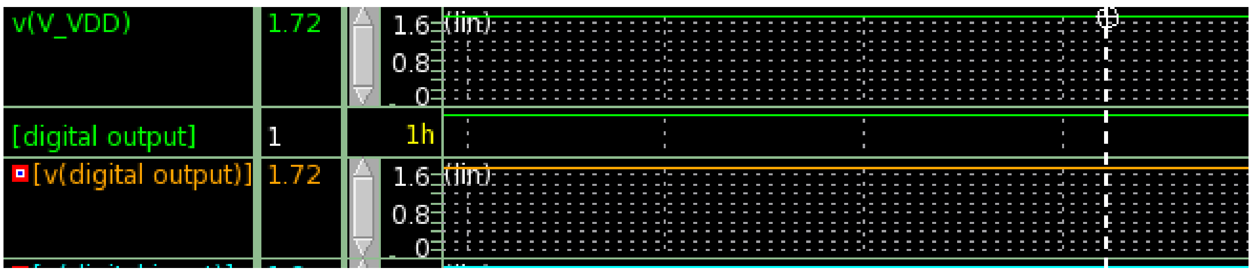

Figure 6: Digital output specifying range for D2A coverter

The digital logic shown in Figure 1 operates from a regulated voltage, typically 1.65V, so a voltage above 70% of the regulated voltage should be converted to a logic-1 and a voltage below 30% of the regulated voltage should be converted to a logic-0. Voltages in-between should be converted to Z. However, according to [3], if the parameters of the A2D conversion are not specified, a voltage above 50% of VDD is a logic-1 and a voltage below 50% of VDD is a logic-0. The default behavior does not model a CMOS standard cell well. In addition, as was the case for D2A conversion, the supply is assumed to be 3.3V as seen in the *simv.msv/interface_element.rpt* report below.

```
a2d loth=1.65v hith=1.65v node=<path to digital input>
```

The default behavior results in the erroneous behavior shown in Figure 7 in which an analog input to the digital is slowly ramping from 0V to 1.6V. However, the A2D conversion does not see a voltage over 1.65V and converts the signal to a logic-0. In Figure 7, [*v(digital input)]* is the analog signal and *[digital input]* is the digital signal after A2D conversion.

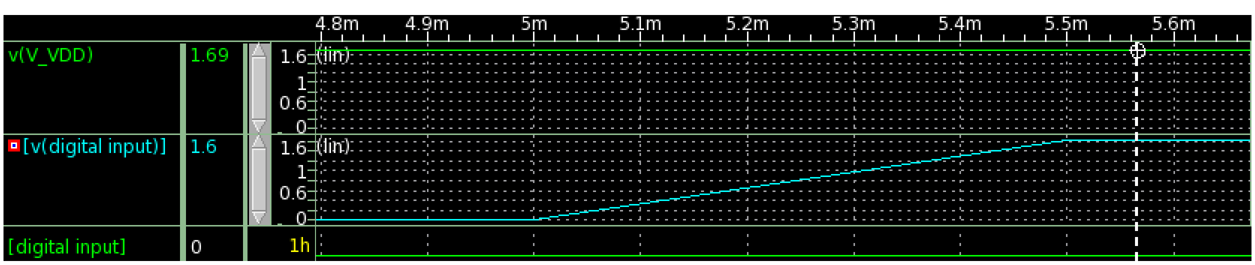

Figure 7: Digital input using defaults for A2D converter

To correct this issue the code in Figure 8 is added to the *vcsAD.init* file. Note that it mirrors the solution for the D2A issue noted above by specifying the digital *vdd* and *vss* pin.

```
a2d vdd=<path to digital vdd pin> vss=<path to digital vss pin>
    loth=30% hith=70% minv=0.1 minv_logic=0
    midv_time=10n midv_logic=Z inst=<path to digital interface> port =*;
```

Figure 8: Addition to vcsAD.init for correct a2d operation

In addition, the logic-0 and logic-1 thresholds are specified as 30% and 70% of *vdd*, respectively. Similar to the *minv_analog* parameter for the D2A converter, the *minv* parameter is used for the A2D converter and specified to be 100mV. This causes the A2D conversion to turn off, saving simulation time, when the analog supply is below 100mV. The *minv_logic* parameter specifies the output of the A2D converter when the *minv* criteria is met, i.e. the analog supply is below 100mV. The default is logic-Z and is changed to logic-0. Lastly, the *midv_time* and *midv_logic* parameters specify the logic value as Z when the analog input to the digital is between 30% and 70% of vdd for greater than 10ns. The default for *midv_logic* is logic-X.

Using the A2D specification in Figure 8 results in the waveform in Figure 9, which meets the requirements for the conversion. The digital input is logic-0 until the input voltage exceeds 30% of *vdd* and remains logic-Z until the input voltage exceeds 70% of *vdd*, at which time the digital input is converted to logic-1.

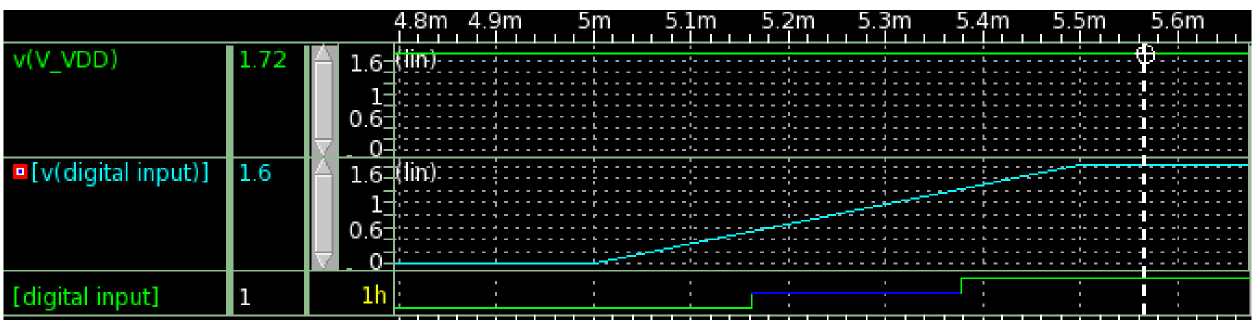

Figure 9: Digital input using correct specification for A2D converter

3.3 CustomSim Control File

The CustomSim control file, by default *xa.cfg,* serves the following purposes in the chip-level verification environment:

- Read a TCL script to specify the digital representation
- Set the simulation accuracy
- Save the state to restore
- Specify currents and voltages to probe

Each of these tasks are elaborated in the following subsections.

3.3.1 Read a TCL script to specify the digital representation

An RTL representation of the digital is used for most tests, but to measure current consumption, including the digital, a post-layout physical netlist with Standard Parasitic Extraction Format (SPEF) is used. A physical netlist contains power and ground connections and the underlying representation of the digital standard cells is a schematic, not a Verilog description of a standard cell. To choose between the two representations on a test-by-test basis the *runVcsAms* script creates a *tcl* file specifying the digital representation. For example, if it is desired to run a gate level representation *runVcsAms* executes the following:

```
echo "set digital gates" >> sim_vars.tcl
```

TCL script *sim_vars.tcl* is then read by the CustomSim control file as seen in Figure 10. The *set_waveform_sim_stat* command dumps simulation statistics to the waveform output file [4]. Figure 11 shows the simulation statistics for a simple startup test. As expected, more memory and CPU resources are utilized when *V_VDD* is ramping and *por_n* releases.

The *load_verilog_file* command loads the physical Verilog netlist. The *load_ba_file* command loads the SPEF file and the *–report_no_ba 2* option will report any missing SPEF nets. A message is displayed if an undefined digital representation is requested.

```
source sim_vars.tcl
if {$digital=="rtl"} {
      puts "xa.cfg: variable digital=rtl"
      set_waveform_sim_stat -type all
} elseif {$digital=="gates"} {
      load_verilog_file -file <path to physical netlist>
      load_ba_file -file <path to SPEF file> -report_no_ba 2
      puts "xa.cfg: variable digital=gates"
} else {
      puts "xa.cfg: variable digital not recognized as rtl or gates"
}
```

Figure 10: TCL CustomSim Control File

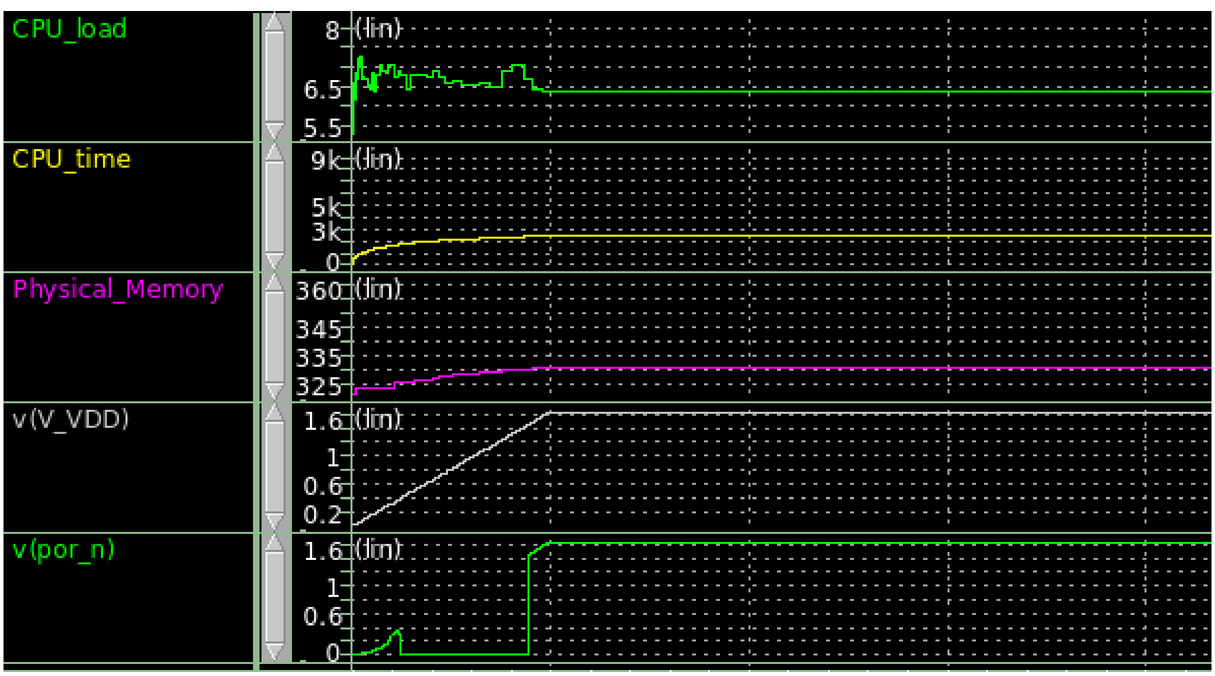

Figure 11: Simulation statistics from set_waveform_sim_stat

3.3.2 Set the simulation accuracy

The speed and model complexity tradeoff for the simulation, i.e. the accuracy, is set globally to 5 by the following statement.

```
set_sim_level 5
```

A sim level of 5 is recommended as a good tradeoff between accuracy and simulation speed and this was found to be true.

The simulation accuracy can be set on a specific instance as well which, if parsed later in the *xa.cfg* file, overrides the global setting. This feature was used to set the simulation accuracy of the digital logic to 3 with the following statement.

```
set_sim_level 3 -inst <path to digital logic>
```

Other simulation accuracy settings are commented out by default. These statements will be uncommented by the *runVcsAms* script when needed, for example, when the RF is being simulated.

3.3.3 Save the state to restore

For many tests the activity up to and including the boot is identical. In these cases, saving the simulation activity to this point in time once and then restoring the activity for each test will reduce the simulation time. In this way, the time up to the boot does not have to be re-simulated. To save the state the following statement was used.

```
set_save_state -time <time to save> -file <state file>
```

The argument to *–time* is the simulation time to save the state. The argument to *–file* is the desired state file name without extension, for example, *state_5ms*. This will create two files, *state_5ms.ic* and *state_5ms.ic.sup*. Both are required to restore the state.

In section 3.2.3 it was shown how to restore the saved state.

3.3.4 Specify currents and voltages to probe

Lastly, the CustomSim control file is used to specify the current and voltages to probe so they can be examined in a waveform viewer. The addition of probes will add to simulation time so an intelligent tradeoff between observability and simulation time must be achieved. Typically, specific signals are probed as opposed to all signals within a block. For example, to probe the slow oscillator voltage the following statement was used. The argument to *–v* specifies the node name:

```
probe_waveform_voltage -v <path to slow osc>
```

Probing currents is similar. For example to probe the current of the supply for the slow oscillator, the following statement is used. The argument to *–isub* is the subcircuit port:

```
probe_waveform_current -isub <path to slow osc supply>
```

To probe voltages for all signals at a specified hierarchy the following statement is used.

```
probe_waveform_voltage <instance path>.* -port 1
```

Setting the *–port* argument to 1 causes a wildcard character to match the sub-circuit port name. The

–*limit* option (not shown) specifies the hierarchy level to which voltages are probed when a wildcard character is used. The default limit is 3. This means that if <*instance path*> is greater than a depth of 3, starting from the testbench, nothing will be probed. If <*instance path*> is a depth of 3 all I/O and internal variables of <*instance path*> will be probed but no hierarchy below will be probed.

3.4 Testbench

The testbench was developed in SystemVerilog. In Figure 12 is the testbench module declaration. It has two parameterized arguments, the seed and the test name. The seed has a default value of 0 but the test name does not, requiring the user to supply a test name. The seed and test name are passed in by the top level CSH script.

```
module chip_top_tb #(
              parameter bit [15:0] SEED = 16'h0000,
              parameter            TESTNAME = "null"
              );
```

Figure 12: Testbench module declaration

The testname is then used in a case statement to call the appropriate task. The task executes the test. For example, the code in Figure 13 accomplishes the following tasks.

- Uses the SystemVerilog *$timeformat* statement to specify that time shall be printed in units of 10^{-9} seconds, 1 digit to the right of the decimal point, string to print after the time value, and minimum field width. [1][2]
- Displays the test to be run along with the seed
- From the string *TESTNAME* call the appropriate task, in this case only test/task *trim_test* is shown
- Displays an error if an unrecognized *TESTNAME* is specified
- Calls *$finish* to exit the simulation

```
initial begin
   $timeformat(-9, 1, "ns", 0);
   $display("Running test %s with seed=0x%4h", TESTNAME, SEED);
   case (TESTNAME)
     "trim_test": trim_test();
     ....
     default     : begin
        $display("ERROR!: Test name of %s not recognized", TESTNAME);
     end
   endcase
   $finish;
end
```

Figure 13: Calling the selected test task

The testbench also automatically checks for specification compliance to accomplish a self-checking testbench. Throughout the test calls are made to *$snps_get_volt* and *$snps_get_port_current* as seen below.

```
check_real_volt(.checked("V(por_n)"), .allowed_low(1.7),
                .allowed_high(1.75),  .actual($snps_get_volt(<path to por_n>)));
check_real_curr(.checked("I(Bias)"),  .allowed_low(20E-9),
                .allowed_high(21E-9), .actual($snps_get_port_current(<path to bias>)));
```

The functions *check_real_volt* and *check_real_curr* are shown in Figure 14. The string input *checked* is used to display what is being checked. Note that these functions track both the error and correct count. The correct count is useful to ensure the test has executed fully and checked what was expected.

```
function void check_real_volt(input string checked, input real allowed_low, allowed_high,
                              input real actual);
   if ((actual > allowed_high) || (actual < allowed_low)) begin
      $display("%0t ERROR: %s %1.3f !< %1.3f !< %1.3f", $time, checked, allowed_low,
                                                              actual, allowed_high);
      error++;
   end
   else begin
      $display("%0t SUCCESS: %s %1.3f < %1.3f < %1.3f", $time, checked, allowed_low,
                                                              actual, allowed_high);
      correct++;
   end
endfunction // check_real_volt

function void check_real_curr(input string checked, input real allowed_low, allowed_high,
                              input real actual);
   if ((actual > allowed_high) || (actual < allowed_low)) begin
      $display("%0t ERROR: %s %eA !< %eA !< %eA", $time, checked, allowed_low,
                                                        actual, allowed_high);
      error++;
   end
   else begin
      $display("%0t SUCCESS: %s %eA < %eA < %eA", $time, checked, allowed_low,
                                                        actual, allowed_high);
      correct++;
   end
endfunction // check_real_curr
```

Figure 14: Functions to check a real voltage and real current

3.5 Replacing the PnP regulator

The testbench has the ability to drive voltages to the Spectre wrapper. For example, if the test wanted to speed up the simulation by removing the PnP regulator and use the *vdd_overdrive* source instead, the following would occur:

1) The *runVCSAms* script uncomments the following line in the Spectre wrapper

```
VDD_OD (V_VDD 0) bsource v=v(vdd_overdrive, 0)
```

2) The test task in the testbench ramps *vdd_overdrive*. For example to ramp V_VDD to 1.65V in 1ms the following code is used:

```
while (vdd_overdrive <= 1.65) begin
   vdd_overdrive += 0.0000165;
   #10ns;
end
```

3) The *vdd_overdrive* real variable is connected to the wrapper in the SystemVerilog testbench:
```
real vdd_overdrive;
wrapper wrapper(
    ....
    .vdd_overdrive(vdd_overdrive),
    ...
                  );
```
4) The *runVCSAms* script replaces the connections to the PnP regulator with DUMMY connections in the Spectre wrapper. It is important to not simply comment out the PnP regulator because any references to the instantiation would cause a compile error.
```
i_pnp_reg (V_EMITTER DUMMY_VDD DUMMY_BASE DUMMY_VSS) pnp_reg
```

3.6 Supporting digital only tests

To debug Test Transactions quickly it is desirable to be able to run a digital-only test using the VCS engine only, not XA. To accomplish this a separate script, *runVcs,* was developed. In this script *vlogan* is called with *+define+DIG_ONLY*. The SystemVerilog testbench uses this define to a) not compile any analog-specific code and b) instantiate the digital block only.

3.7 NVM stuffing

Operational settings are read from the NVM at boot. To verify different settings the ability to "stuff" the NVM was developed. A perl script takes as input the netlist of an un-programmed NVM and the desired bits to change. The perl script then modifies the netlist to define the state of the desired memory cell bit(s).

3.8 RF Verification

The RF receive path was verified with a 100MHz amplitude modulator. This frequency is well below the specified modulation frequency, but was a good tradeoff between simulation speed and stimulating the RF. The instantiation of a Verilog-A model of the AM modulator is below. Input *rf_rx_voltage* is the input signal and is driven in the SystemVerilog testbench as type *real*.

```
i_am_modulator (rf_rx_voltage RF_RX) am_modulator f_carrier=100M vin_max=2
                                     vin_min=0 unmod_amp=250m mod_depth=1
```

Similarly, to verify the RF transmit path the resonator shown in Figure 2 was also operated at 100MHz.

4. Results, debug, and improvements

In this chapter, the overall results are presented along with debug hints, what did not go well, and future improvements.

The chip-level test plan was followed and resulted in a first pass success. The current as simulated closely approximated the silicon measurement. Approximately 40 AMS tests were written. Re-running all simulations on a 4-core compute server took about 3 weeks. The self-checking features implemented were invaluable. Re-checking 40 tests visually is error-prone and excessively time consuming.

4.1 Debug using the simv.msv directory

The *simv.msv* directory contains a number of invaluable files for debug [3]. This section will explain the most noteworthy ones.

4.1.1 interface_element.rpt

This report lists all signals crossing the analog/digital interface and requiring an A2D or D2A converter along with the parameters controlling the conversion. This file was used to debug the issues described in subsection 3.2.4 . In addition, bidirectional interface nodes are also listed in this file. The *real* type values in the SystemVerilog testbench, such as *rf_rx_voltage* described in section 3.8 , use a *r2e* converter to convert from a *real* type to an *electrical* type. Signal *rf_rx_voltage* is a bidirectional signal.

4.1.2 names_map.rpt

This report lists how signal names are mapped when a child Verilog or VHDL module is instantiated in a parent SPICE netlist. This is particularly useful for determining the analog signal to probe that is connected to a digital I/O. For example, consider the following Verilog module *my_dig* instantiated as shown in a Spectre netlist. The *names_map.rpt* file would contain *<path to slow_osc>:<path to net60>*. For a module with many signals and vectors this is very helpful.

```
// Verilog
module my_dig(input slow_osc);

// Spectre
i_my_dig(net60) my_dig
```

4.1.3 port.rpt

This report is similar to the *names_map.rpt* report but reports the mapping between modules and sub-circuits at the same level, i.e. child to child, as opposed to parent to child.

4.2 What did not work well

In this chapter is an explanation of issues that were encountered and workarounds.

4.2.1 Usage of snps_above() or snps_cross() slows the simulation

The *snps_above()* and *snps_cross()* statements were initially used in the SystemVerilog testbench. These commands generate a digital event that allows the testbench to react when the voltage of the specified signal has gone above or below the specified threshold. The event can be used in a Verilog *always* block to trigger execution. The author found that these statements had a large negative impact on simulation performance. This is expected, considering that the simulator is simply sampling the signal and looking for the event criteria. The sampling rate cannot be specified to *snps_above()* and *snps_cross()*. To work around this limitation of *snps_above()* an alternative was developed. The SystemVerilog code and how to call the task is in Figure 15. The event *above_thresh* is triggered and the forever loop exits when the specified node is above input *threshold*, otherwise time is advanced by input *sample_period*. If the specified node does not rise above input *threshold* by the time specified by input *timeout*, all threads are killed and the task exits.

```
sample_until_above(.threshold(<real threshold>), .sample_period(<sample period time>),
                   .timeout(<timeout time>);

task sample_until_above(input real threshold, input time sample_period,
                        input time timeout);
   fork
      forever begin // thread 1
         if ($snps_get_volt(<path to node>) > threshold) begin
            -> above_thresh;
            break;
         end
         else
           #(sample_period);
      end
      begin // thread 2
         #(timeout)
         $display("Error, threshold %fV not reached in %t", threshold, timeout);
      end
   join_any;
   disable fork;
endtask
```

Figure 15: Sampling task for voltage above threshold

4.2.2 Confusing probe_waveform_* statements

The *–limit* option to a *probe_waveform_** statement should start at the level of the path specified. For example *–limit 0* would probe all signals at the instance path specified. The user should not have to count the hierarchy.

The *–vall <instance name>* option to *probe_waveform_voltage* is documented to specify the name of the instance at which the voltage of all terminals are probed[4]. However, the following statement in the CustomSim control file resulted in no signals being probed and no skipped signal messages.

```
probe_waveform_voltage -vall <instance name> -limit 99
```

Similar issues were encountered with *probe_waveform_current*. It was difficult to probe multiple currents with a single statement.

4.3 Improvements

The first improvement would be automatic generation of the top-level netlist. This was the only manual task required in order to run a simulation. In addition, automatic alerts when a schematic changes would also be helpful.

The second improvement would be to collect functional coverage through SystemVerilog *covergroup's*. Functional coverage would be an additional check to ensure the tests were behaving as expected and to obtain a more complete picture of the verified scenarios.

5. Conclusions

As was stated previously, the CustomSim/VCS simulator helped to ensure a first pass success. The current measurements correlated well to silicon measurements. The simulations were of sufficient speed to complete the chip-level verification plan.

A big thanks to Joe Perttu who was invaluable in the success the author experienced with CustomSim/VCS and for encouraging the author to write this paper.

6. References

[1] *IEEE Standard for SystemVerilog*, IEEE Computer Society, Etc., 2012

[2] *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, G. Tumbush and C. Spear, 3rd Edition, 2012

[3] *Mixed-Signal Simulation Users Guide*, Synopsys, Version M-2017.03-SP2, June 2017

[4] *CustomSim™ Command Reference,* Synopsys, Version M-2017.03-SP2, June 2017