Creating a Complete Low Power Verification Strategy using the Common Power Format and UVM

Robert Meyer
Medtronic, Inc.
8200 Coral Sea Street NE
MS MVC61
Mounds View, MN 55112
robert.j.meyer@medtronic.com

Joel Artmann
Medtronic, Inc.
8200 Coral Sea Street NE
MS MVC61
Mounds View, MN 55112
joel.b.artmann@medtronic.com

ABSTRACT
Given the operational constraints that low power products face, designs must manage every joule of available energy with the utmost care. Furthermore, as chips move to progressively smaller nodes, it is critical to create circuitry whose role is to minimize the effects of leakage current and other such power-stealing phenomena. This circuitry presents a sophisticated verification challenge which needs to be addressed by an equally sophisticated solution.

This paper describes an approach that was used to create a complete low power verification methodology. This methodology verifies the power control logic embedded in the SoC firmware in concert with the SoC's power control hardware so that it includes all of the low power control feature in the SoC. This methodology then back-annotates low power verification results into a verification plan to dynamically track progress and quantitatively measure the verification results. All aspects of low power verification are integrated into this methodology including verifying control signal behavior, low power intent of the design and then tying simulation results back into a verification plan to measure verification results. We started with a Common Power Format (CPF) file, which was used to generate a compatible Universal Verification Methodology (UVM) Verification Component (UVC) and PSL assertions devoted to verification of power domain isolation & retention. We then extended these verification components to include other power related features such as clock gating and firmware control. This methodology has cut design and verification cycle times by 50%, while simultaneously significantly increasing the probability of an error-free design. The code created by this approach: the UVCs and assertions, can easily be reused on successive design projects.

Categories and Subject Descriptors
B.5.2 [REGISTER-TRANSFER-LEVEL IMPLEMENTATION]: Design Aids –optimization, simulation, verification

General Terms
Hardware Description Languages, Optimization, Simulation, Verification

Keywords
Electronic System Level, Design, Verification, Mixed Language Flow, CPF, UPF, UVM, Low Power, Control Logic

1. INTRODUCTION
The pressure put on today's SoC designs to conserve power has resulted in new, complex design circuitry related to power control. Verification of these power control features presents equally complex challenges to the Verification and Design teams, such as: Do all device features work correctly when portions of the device are in a low power mode? Does the device transition correctly from one power mode to another? How can we be sure that all hardware and firmware interactions related to low power have been verified before design manufacture?

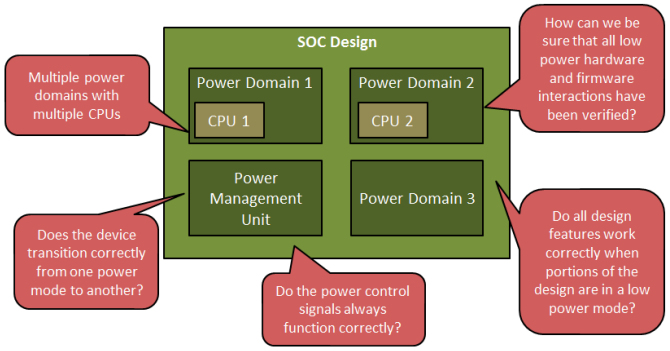

Figure 1. The low power verification challenge

A complete low power verification solution is needed to address these challenges and reduce the complexity and manual effort required to verify a low power SoC design. There are three components to a complete solution:

- Verify proper operation of the power control signals. These signals must be shown to reach the proper state in the proper order at all times.
- Verify the low power "intent" of the design. The design must be shown to transition to all low power states only when expected.
- Simulation results must be quantified and show that all possible combinations of low power behavior have been executed.

A complete solution can be reached in an accelerated manner with less manual effort by automating significant parts of the verification process. We did this by capturing the low power intent of the design in a Common Power Format (CPF) file and using this file to automatically drive much of the verification effort. Whereas using the CPF file to drive portions of the low power verification effort is an extremely helpful and time saving strategy, it is by no means complete. There is much that remained to be done to augment the verification code that was generated from the CPF file to insure an error-free low power implementation.

This paper will show that verification code that can be directly derived from the CPF file in an automated fashion. It will go into detail to show how this code was then augmented by us to greatly enhance the quality of the low power verification effort. The paper will describe four aspects of the verification flow:

- Generation of PSL assertions to check power control signals and power mode behavior. Augmentation of these with custom assertions to create a more complete low power checking strategy will be described.
- Inclusion of CPF file constructs in waveform viewers for debug.
- Creation of an 'e' language compatible UVM Verification Component to capture low power feature coverage in dynamic simulations. The paper will demonstrate how the automatically generated UVC can be extended to include low power concepts that are not defined in the CPF file such as clock gating and to add checking for low power behavior.
- Generation of a low power verification plan for inclusion in the chip or system verification plan. This plan can then be used to tie simulation results back to the verification plan and provide low power feature coverage traceability.

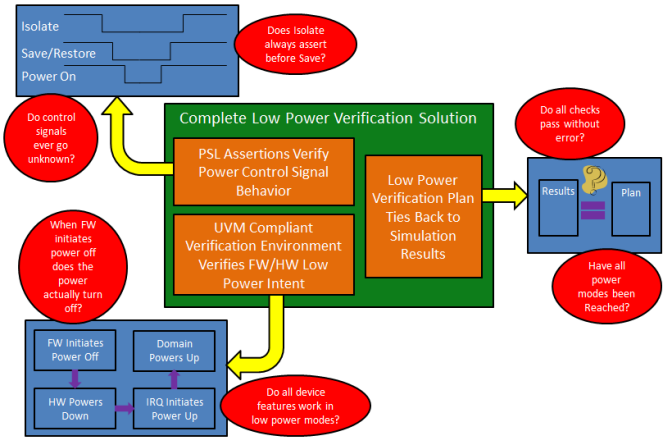

Figure 2. The complete low power verification solution

2. CREATING A CPF FILE HIERARCHY FOR SIMULATION

A CPF file is very closely associated with its corresponding design and contains hierarchical references to structures within the RTL. Tools used for place and route and formal verification require all design references to use the top level module of the design as the root of these references. The simulator, however, requires the addition of a testbench to instantiate the top level design module. This will have the effect of throwing off all of the hierarchical references within the CPF file.

The solution to this problem is to create a second CPF file for use in simulation only. Don't panic, it is not necessary to duplicate the information contained in the design CPF file. Instead, the simulation CPF file simply identifies the testbench module name and refers to the design CPF file. The simulator then inserts the testbench module into the hierarchical references called out in the design CPF file.

Below is an example of the content of a simulation CPF file:

```
set_cpf_version 1.1
set_hierarchy_separator "/"

set_design testbench -testbench
   set_instance soc_design_top
   include soc_design.cpf
end_design testbench
```

As you can see, there is not much to it and all of the key low power information remains in the design CPF file.

3. GENERATING PSL ASSERTIONS

Low-power assertions were written as well as auto-generated to provide a portion of the needed checking for the low power feature of the design. Assertion Based Verification (ABV) is a general verification technique that can be used to provide temporal checking of potentially any design behavior. In our case, we used these assertions to verify proper temporal power control signal and power mode behavior.

3.1 Assertions Generated from the CPF File

Table 1 contains a list of the power control signal assertions that are created for each power domain defined in the CPF file. These assertions will serve to verify proper power control signal temporal behavior as the design drives them throughout the simulation.

Assertion Name	Description
Always_isolated_when_shutoff	The isolation signal is always asserted whenever shutoff is asserted.
Isolate_off_follows_pwrup	The isolation signal is eventually de-asserted following power up.
Isolation_signal_not_x	The isolation signal never goes unknown.
Shutoff_follows_isolate	The shutoff signal is eventually asserted following assertion of the isolate signal.
Shutoff_signal_not_x	The shutoff signal is never unknown.
*Always_save_before_restore	When the save signal is asserted it is eventually followed by assertion of the restore signal.
*No_restore_while_shutoff	The restore signal is never asserted while the shutoff signal is asserted.
*No_save_while_shutoff	The save signal is never asserted while the shutoff signal is asserted.
*Restore_signal_not_x	The restore signal never goes unknown.
*Save_signal_not_x	The save signal never goes unknown.
*Shutoff_follows_save	The shutoff signal is eventually asserted following assertion of the save signal.

*This assertion is only included if save and restore signals are defined for the power domain.

Table 1. Assertions generated automatically from the CPF file

3.2 Custom Low Power Assertions

Whereas the auto-generated assertions do a good job of covering the temporal relationships between the power control signals, the assertions do not form a complete low power verification scheme. For instance, there is no assertion which verifies that when the power shutoff control signal is asserted for a given power domain that the power within the domain actually shuts off. In order to close this gap, it is necessary to augment the automatically generated assertions with additional, designer created assertions and pull them into a simulation. There are two assertions which can be added for every power domain:

- Whenever the power shutoff signal is asserted, the power net in the corresponding power domain shall be powered down
- Whenever the power shutoff signal is de-asserted, the power net in the corresponding domain shall be powered up

Below is an example vunit for a power domain module which contains the assertions described above.

```
vunit my_design_module_vunit (my_design_module) {

// if power shutoff is asserted then power net is off (1'bX)
LPV_POWER_NET_OFF : assert always ({power_ctrl == 1} |->
{power_net === 1'bx}) @ (posedge(system_clk));

// if power shutoff is de-asserted then power net is on (1'b1)
LPV_POWER_NET_ON : assert always ({power_ctrl == 0} |->
{power_net === 1'b1}) @ (posedge(system_clk));

}
```

The behavioral model used in simulation for a power domain may or may not be designed to be sensitive to the value driven onto the power net. In other words, depending on how the behavioral model is written, it may operate in simulation independently of the power net value in simulation. This means in operates in the same fashion whether the power for the domain is turned on or not. If the behavioral model description is independent of the power net value, it is possible for a bug in the power shutoff feature to go unrecognized in simulation. The only way to catch it under these circumstances is with the custom assertions described above.

4. VIEWING CPF FILE CONSTRUCTS IN A WAVEFORM VIEWER

We were able to view some of the low power constructs associated with the CPF file in a waveform viewer. These include the low power assertions and power domain state information. This is a powerful tool and a vital step in debugging errant power control behavior.

First, when running the simulation we probed the low power information. This can be done by either waving the entire design or by waving the ALPV_MODEL module which is created by inclusion of the CPF file. This module appears at the same level of hierarchy as the top level of the design.

Analyzing the simulation results using post-processing, we started the waveform viewer and loaded the waveform database as per usual. The twist for low power analysis was to also load a simulation snapshot in order for the low power state information to be loaded into the waveform viewer.

Once inside the simulation tool, the power mode and power domain information is loaded into a hierarchical, tree view display. See Figure which shows the power display for a project.

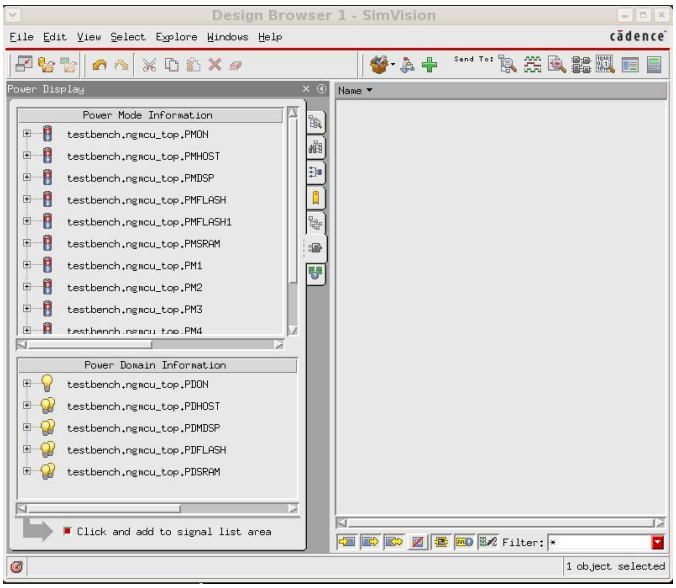

Figure 3. Low power information displayed in a design browser window

To load the state information for one of the domains, right click on a domain listed in the "Power Domain Information" panel and select "Send to waveform Window". There is a great deal of information recorded for each power domain, including information on many of the signals in the domain related to isolation, save and restore. What we found most helpful were the State, Mode, Nominal_Condition and Voltage entries. Figure shows this information being displayed for the PDMDSP and PDFLASH domains of a design. Each of the domains is being shut off then turned back on again in the figure.

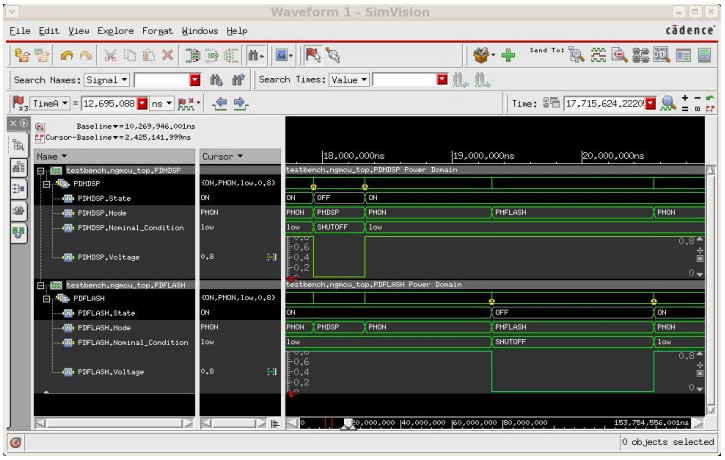

Figure 4. Key low power information displayed in ta waveform window

It is also possible to view the low power assertions in an assertion browser window. Specifically, all of the probed assertions in the design will be listed in the assertion browser window. We were able

to see the number of times each assertion passed and failed. As shown in Figure 3 below, filtering by "*LPV*" will generate the complete list for the PDHOST domain. This can be very a very useful strategy to identify and debug low power failures, particularly in the early stages of design.

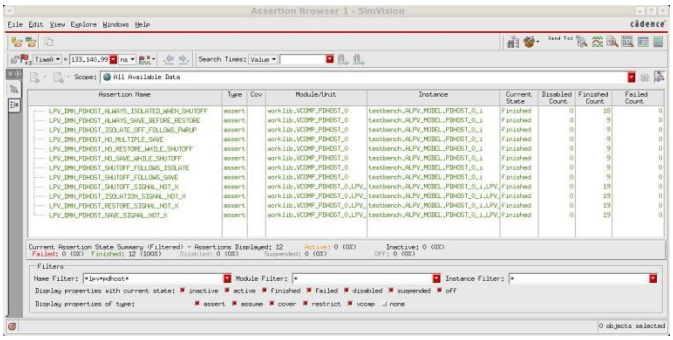

Figure 5. CPF file assertions displayed in an assertion browser window

.

5. CREATING LOW POWER COVERAGE AND VERIFICATION CODE

By using a utility program available from our tool vendor, we were able to generate verification code to create coverage on low power state behavior defined by the CPF file. We then manually extended this code to create coverage on low power states that are not a default part of the CPF file such as "SLEEP' mode. Finally, we used the code as part of a UVM compatible verification environment to verify the low power behavior of a design.

5.1 Generate Low Power Coverage Code

A script program (create_code.sh) was used to generate 'e' code for dynamic simulations. The generated 'e' code contains constructs such as an agent, monitor and structure definitions used to monitor and capture feature coverage power domain behavior as defined in the CPF file. The 'e' code testbench is mated to the design hierarchy by using signal references embedded within the CPF file.

5.2 Extending power Mode Coverage

Currently, the CPF v2.0 standard has definitions for the "ON" and "OFF" power states for a given power domain and not much else. Power states such as "SLEEP" where the power remains on but the clock has been gated are not included nor covered in the automatically generated verification code. However, by manually creating extensions to the verification code, power states such as "SLEEP" can be accounted for and verified in simulation.

For example, here is a piece of code using the 'e' verification language which extends the pre-defined power state definitions to include some custom power states. In this case, the concept of a "SLEEP" power state has been added along with several others.

```
extend uvm_lp_state_t : [SLEEP, STOP, DEEP_PD, ARRAY_RET,
                         ARRAY_RET_LOW, ARRAY_SHUT, PERIPH_SHUT];
```

Next, it is necessary to create a mechanism which will detect these newly defined low power states in the design. To accomplish this, we created event ports in the testbench and connected them to signals in the RTL which indicated the power state of a particular power domain. A method was created which was always running for the length of simulation which was responsible for detecting changes in the power state of a domain and indicating new states to the scoreboard via an event. As discussed in section 5.3.3, the low power scoreboard then records the power transition and ultimately uses it to provide checking that the power transition was indeed expected to happen. Below is some code for a domain which has many unique power states defined.

```
// Definition of ports connected to RTL state indicator signals
port_lp_control_e : list of in event_port is instance;
keep port_lp_control_e.size() == 6;
keep soft port_lp_control_e.hdl_path() == "lp_control";

// This method watches for changes in the RTL which indicate
// a change in low power state and emits a corresponding event
// used by the low power scoreboard
state_watcher_lp_control() @clock is {
        var int_lp_control: uint;
        var valid_state: bool;

        while TRUE {
           message(HIGH,"Waiting for any change on the lp control
               bus.");
           wait (@port_lp_control_e);
           message(HIGH,"Detected a change on the lp control bus");

           // wait for all control signals to settle to their steady state
           wait [550]*cycle;
           message(HIGH,"Sampling lp control bus.");

           int_lp_control = port_lp_control$;

           message(HIGH,"The new lp control bus state is ",
               int_lp_control);

           valid_state = TRUE; // default value
           case int_lp_control {
              0x04: { power_state_lp = ON; };
              0x14: { power_state_lp = ARRAY_RET; };
              0x16: { power_state_lp = ARRAY_RET_LOW; };
              0x10: { power_state_lp = ARRAY_SHUT; };
              0x18: { power_state_lp = PERIPH_SHUT; };
              default: {
                 valid_state = FALSE;
                 dut_error("Unidentified power state [", power_state_lp,
                           "].");
              };
           };
           if valid_state { emit lp_state_change;};
        };
     };
};
```

Next, we added logic which tracks the power state of each domain and creates coverage to record how many times each power state was reached. Transition coverage can also be added to record all transitions from one power state to the next. This code will ultimately be used to measure which power modes and power transitions have been exercised in simulation. Below is an example of a coverage statement used on one power domain:

```
cover lp_power_domain is {
    // Covering power modes reached
    item power_state : uvm_lp_state_t = power_state
    using ignore = (power_state not in [ON, OFF, SLEEP]);

    // Covering power transitions made
    transition power_state using ignore = (
      // only transitions to and from the state of ON are allowed
      (prev_power_state == ON and power_state == ON) or
      (prev_power_state != ON and power_state != ON)
    );
  };
```

5.3 Extending Low Power Checking

The CPF file does drive some verification directly via the PSL assertions that are used in the course of dynamic simulations. These assertions primarily verify proper control signal behavior in that the signals are verified to be asserted and de-asserted by the SoC design in the proper order and never take on a value of unknown. While this is helpful, it is definitely not the entire verification needed to insure correct behavior of the low power feature. For example, the assertions do not answer questions like: When the system was required to transition into a low power mode, did it correctly do so? If the system did make a power transition, was it expected?

However, once again, we extended the verification code that was automatically generated in section 5.1 to add this functionality. In order to get this done, we needed to build a UVM verification environment with a few key components:

- A way to drive device firmware and generate/detect stimulus that will change the power state of the device.
- A way to generate/detect power state changes in the RTL. In our case, we used hardware IRQ signals to force power state changes.
- A mechanism to monitor the low power state of the device
- A checking mechanism that will verify each expected power state transition eventually happens and that each transition that does happen was expected.

A simplified diagram showing the UVM compatible low power verification environment is shown in Figure 4.

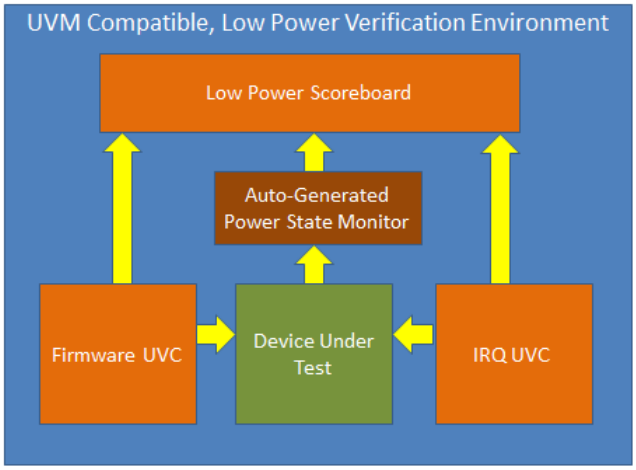

Figure 6. Diagram of the UVM low power verification environment

5.3.1 Creating a Firmware UVC

In order to extend low-power checking, it was important to drive the verification from firmware. With this capability, we were able to execute firmware instructions which drive the device into low power modes using the same mechanisms that the "real" firmware would use. Using software extensions available from some EDA tool vendors, it is possible to create firmware routines that can be "driven" onto a microprocessor in a SoC much in the same way a testbench might drive a signal pin on the SoC in hardware. We created a firmware UVC capable of driving and monitoring firmware low power instructions. A Sequence generator was created which injects power related instructions into a processor at random intervals. The constrained random nature of the stimulus allowed us to simulate unforeseen, but valid, combinations of firmware and hardware state that the design might reach after manufacturing in the real world. A monitor mechanism was created to detect and communicate the power expectations via a method port to a scoreboard checking mechanism.

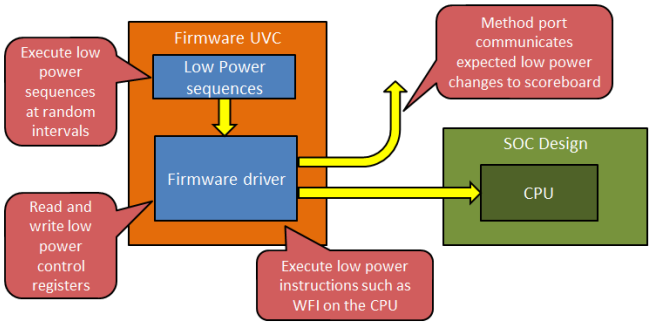

Figure 7. The Firmware UVC

Below is one example of a low power sequence, LP_PDHOST_WFI_GEN, which is executed at constrained random intervals by the verification environment. This sequence drives the processor firmware to enable a transition to a low power state. This sequence randomly generates a Boolean value that will be used to select a child sequence which then sets a value written into a system control register by firmware. When set, the processor will be configured to enter a 'Deep Sleep' or power off mode after the next interrupt has been serviced. When this variable is cleared, the processor will be configured to enter 'Sleep' mode, where power remains on but the clock to the domain is gated.

```
extend mdt_ngmcu_sw_sequence_kind_t : [LP_PDHOST_WFI_GEN];
extend LP_PDHOST_WFI_GEN mdt_ngmcu_sw_sequence {
   !poweroff_wfi_seq: LP_PDHOST_POWER_OFF_WFI
mdt_ngmcu_sw_sequence;
   !clockoff_wfi_seq: LP_PDHOST_CLOCK_OFF_WFI
mdt_ngmcu_sw_sequence;
   deep_sleep: bool;

   body() @driver.clock is only {
      // execute the WFI instruction
      if deep_sleep {
         message(LOW, "ISX: LP: LP_PDHOST_WFI_GEN: About to
execute a poweroff_wfi sequence.");
         do poweroff_wfi_seq; // WFI will not complete until the next IRQ
because CPU is asleep
         message(LOW, "ISX: LP: LP_PDHOST_WFI_GEN: poweroff_wfi
sequence complete.");
      } else {
```

```
        message(LOW, "ISX: LP: LP_PDHOST_WFI_GEN: About to
execute a clockoff_wfi sequence.");
        do clockoff_wfi_seq; // WFI will not complete until the next IRQ
because CPU is asleep
        message(LOW, "ISX: LP: LP_PDHOST_WFI_GEN: clockoff_wfi
sequence complete.");
      };
   };
};
```

The child sequence, LP_PDHOST_POWER_OFF_WFI or LP_PDHOST_CLOCK_OFF_WFI, sets the system control register as desired and then executes a WFI (Wait For Interrupt) firmware command on the processor. An example of one of these sequences is shown below:

```
extend mdt_ngmcu_sw_sequence_kind_t :
[LP_PDHOST_POWER_OFF_WFI];
extend LP_PDHOST_POWER_OFF_WFI mdt_ngmcu_sw_sequence {
   !sys_control_reg_seq: LP_PDHOST_WRITE_SYS_CONTROL_REG
mdt_ngmcu_sw_sequence;
   !wfi_seq: WFI mdt_ngmcu_sw_sequence;
   lp_change: mdt_lp_change_s;
   keep lp_change.domain == PDHOST;
   keep lp_change.state == OFF;
   keep lp_change.trigger == SOFTWARE;
   keep lp_change.required == TRUE;
   keep lp_change.time_stamp == 0;

   body() @driver.clock is only {
      message(LOW,"ISX: LP: LP_PDHOST_POWER_OFF_WFI: Turning
CPU power off.") {};
      do sys_control_reg_seq keeping {
         it.sleepdeep == TRUE;
         it.sleeponexit == FALSE;
      };
      driver.lp_change_out$(lp_change);
      do wfi_seq;
      message(LOW,"ISX: LP: LP_PDHOST_POWER_OFF_WFI: Sequence
complete.") {};
   };
};
```

The sequence also creates a structure, lp_change, and places it on a scoreboard through a method port. The structure is used by the low power scoreboard to identify an expected power change in the PDHOST domain. Later in this document I will show how the scoreboard uses these items to provide low power verification by comparing these expected power transitions with actual power transitions observed in the hardware model.

The example sequences in this section show a portion of the low power sequences for the PDHOST domain. It is necessary to create similar sequences for all of the power domains in the design.

5.3.2 Creating an IRQ UVC

Another component required by the low power verification environment was a UVC which will drive IRQ lines that trigger low power state transitions. In the real world, IRQ events are not coordinated with the current activity of the processor or firmware. In order to model this type of interaction in simulation, the UVC needed to be capable of driving interrupts at constrained-random intervals completely independent from the firmware UVC. The UVC included a monitor that would detect IRQs and communicate the events to a scoreboard mechanism via a method port. The scoreboard would use these events to record the expected power state transitions for the SoC design.

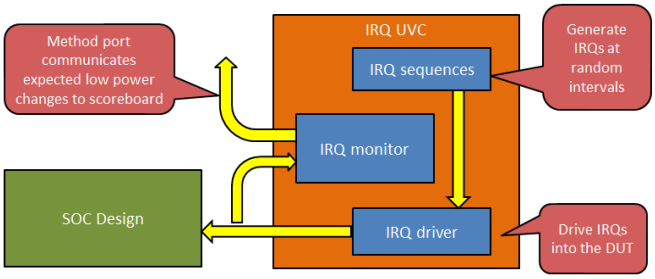

Figure 8. The IRQ UVC

Below is an example of a sequence used by the testbench to drive the IRQ line at constrained-random intervals in our design. The IRQ line is asserted for a random time interval between 400us and 10us.

```
extend mdt_ngmcu_sw_sequence_kind_t : [LP_ASSERT_GPIO0_IRQ];
extend LP_ASSERT_GPIO0_IRQ mdt_ngmcu_sw_sequence {
   body() @driver.clock is only {
      var random_delay : time;

      message(HIGH,"ISX: LP: LP_ASSERT_GPIO0_IRQ: Asserting gpio0
IRQ.") {};
      // generate a pulse which is at least one clock in duration.
      gen random_delay keeping { it > 400 ns; it < 10 us; };
      driver.smp.gpio0_simple$ = 1'b1;
      wait delay(random_delay);
      driver.smp.gpio0_simple$ = 1'b0;
      message(HIGH,"ISX: LP: LP_ASSERT_GPIO0_IRQ: Assert gpio0 IRQ
complete.") {};
   };
};
```

A method port was added to the IRQ UVC driver to pass IRQ events to the scoreboard which would then expect a power transition back to the "ON" power state. Below is an example of this code.

```
lp_irq : mdt_lp_irq_s;
   keep lp_irq.domain == PDHOST;

   event lp_irq_e is @irq_e;
   on lp_irq_e {
      lp_irq.host_irq = p_env.isx_exception$.as_a(mdt_ngmcu_interrupt_t);
      lp_irq_out$(lp_irq);
   };
```

5.3.3 Development of a Low Power Scoreboard

The scoreboard is used to provide the needed checking mechanism for the low power control feature of the device. When the firmware or IRQ generates stimulus resulting in an expected power state transition, this is communicated to the scoreboard via the method ports described previously and an item is placed on a list structure for expected power transitions. Actual power state transitions are detected by the power state monitor and also communicated to the scoreboard via method ports and placed on a list of actual power state transitions. Each new entry on the list of actual power transitions is compared to the list of expected power transitions to verify that the SoC design did indeed execute the expected power transitions. In this way the low power "intent" of the low power feature was verified for

a given domain. Separate list structures were used to provide a checking mechanism for each power domain.

The code to monitor the low power state of the device has already been created automatically from the supplied utility program and can be used by the scoreboard to recognize actual power state transitions.

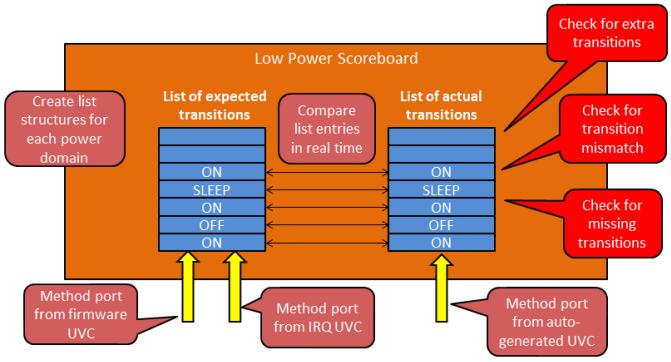

Figure 9. The low power scoreboard

There are 3 checks that we needed to apply in the scoreboard:

- Check for unexpected power transitions. This would be an item on the list of actual transitions without a corresponding item on the list of expected transitions.
- Check for missing power transitions. This would be an item on the list of expected transitions that is never paired with a corresponding item on the list of actual transitions.
- Check for incorrect power transitions. This would be an item that does not match on both the expected and actual power transition lists.

Rather than create unique checks for each power domain, macros were created for the three different checks. The macros were then applied to all of the power domains in the design. This reduced the probability of an error in the verification code and reduced the number of lines of code that needed to be written. Below is an example of one macro that was used to check for missing power transitions.

```
define <lp_missing_power_transition'action> "lp_missing_power_transition
<domain_name'name>" as computed {
   var statement: string;

   statement = append(statement,
            "check lp_missing_power_transition_", <domain_name'name>, "
that ");
   statement = append(statement,
            "    (expected_size - actual_size !> 1)  else ");
   statement = append(statement,
            "    dut_error(\"Missing power transition in the \", domain, \"
power domain.\") {");
   statement = append(statement,
            "       print_change_item(missing_item);");
   statement = append(statement,
            "    }");

   return statement;
};
```

6. GENERATING A LOW POWER VERIFICATION PLAN

It is possible to generate a low power verification plan from the CPF file either with or without automation. We used this to compare the low power verification goals with what actually happened in simulation and thus "close the loop" on low power verification and empirically determine when all verification and coverage goals have been met for the low power feature.

We set the main verification plan for the SoC design to reference the low power verification plan via Meta text. The Meta text acts as a link between the two verification plans, much in the same way that a link to a web page points to a separate html document. In this way, the auto-generated plan was included in the larger verification plan for the SoC.

Ideally, automation is used to read in the main verification plan and add the content from the low power verification plan as part of that process. When automation is used, the simulation results can be tied back to the verification plan and automatically fill in the low power section of the plan with the verification results created by the 'e' code during simulation. Figure 5 shows the power mode coverage obtained by a series of simulations for this project.

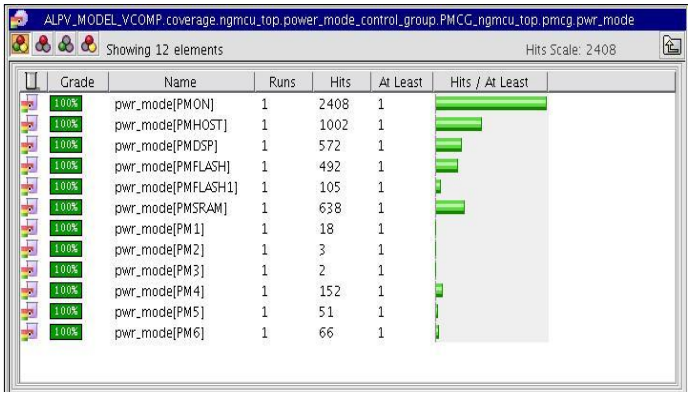

Figure 10. Power mode coverage

7. SUMMARY

In summary, a complete low power verification methodology:

Reduces time to market
Start with the Common Power Format file to capture the low power intent of a design and then use it to auto-generate significant portions of the low power verification effort thus reducing time to market while simultaneously increasing the quality of results.

Improves quality of results
The code auto-generated from the CPF file can be extended to create PSL assertions and a UVM compliant UVC which together form an exceptionally effective and thorough verification strategy

Provides a complete solution
The PSL assertions and UVM compliant verification environment coverage results can be pulled together and compared to verification plan goals to quantitatively analyze simulation results and form a complete low power verification picture.

We started with the Common Power Format file, to capture the low power intent of a design and then use it to drive the low power verification effort thus reducing time to market while simultaneously increasing the quality of results. The code generated from the CPF file was extended by the verification engineer to form a complete low power verification picture. We showed how the CPF file can be hierarchically structured to facilitate its use in dynamic verification, formal verification as well as place and route. It was shown how PSL assertions automatically derived from the CPF and then augmented with manually created assertions to verify power control signal behavior. The power state of defined power domains can be viewed in a waveform viewer as an aid to debug. Coverage and monitoring code can easily be generated to quickly get a UVM compatible verification environment started. This code can be extended to include other low power states such as “SLEEP” modes where the domain clock is gated and to include checking for low power behavior. Lastly, we showed how all of this can be included in a verification plan to close the loop on low power verification and empirically determine when verification is complete.

By taking full advantage of the CPF file and then extending automatically generated verification constructs manually, to create a complete low power verification effort; we can expect superior time to market and quality results.

8. ACKNOWLEDGMENTS

Thanks to Brent Carlson for his help in establishing this methodology with the support of the Cadence Incisive products and Joseph Hupcey III of Cadence for his editorial support.