Those Who Left Us with Much, also Left Much Too Soon.

By Tom Fitzpatrick, Editor and Verification Technologist

As many of you who have been long-time readers of Verification Horizons know, I usually like to spend a little time in these Editor’s Notes relating an anecdote about my family and tying it back to a trend or theme in our industry related to the articles in that issue. But this time around, I feel moved to write about two members of our EDA family who were taken from us too soon this year.

Stu Sutherland passed away this summer, shortly after DAC. I first met Stu over twenty years ago when we were both working on what was then Verilog-1995. Since that time, we continued to collaborate on standards, including IEEE 1364 and IEEE 1800. In addition to being a key technical contributor to these standards, I’m not sure that many people outside of that process know just how much work Stu actually did as technical editor in merging 1364 and SystemVerilog 3.0 from Accellera into IEEE 1800. The word “Herculean” might come close to describing it. His efforts continued as editor through the most recent release of IEEE 1800. Stu was one of the premier independent Verilog educators in our industry as well as a prolific author of books and technical papers. I had the honor of co-authoring a few papers with Stu over the years. To all of these accomplishments, Stu always brought an unfailing sense of decency, humility and serenity that one could not help but treasure. To be able to have called Stu a friend as well as a colleague is truly one of the highlights of my career.

Joe Daniels passed away suddenly last week. Joe and I worked together for the past several years on the Accellera Portable Stimulus Working Group, where he served as technical editor. As Vice Chair of the PSWG, I had many interactions with Joe both in group meetings and one-on-ones as we worked through the details of turning the sometimes chaotic contributions from multiple WG members into a coherent standard. Joe was extraordinarily good at what he did and was always ready to offer valuable advice on many aspects of the standardization process—from meeting management to document organization—and was always encouraging and direct. Whenever he and I spoke, either in person, on the phone or by video chat, he would never let the conversation end without asking how I was doing or how my family was. Joe’s ability to combine this personal touch with his technical depth and clarity made him unique in my estimation.

Stu and Joe will be greatly missed, and so this issue is dedicated to them.
We have four articles in this issue, but what we lack in quantity we more than make up for in quality. We start with “FPGA Verification Challenges and Opportunities” from my friend and colleague Harry Foster. As you may know, Harry has been the brains behind our biennial Wilson Research Group survey of the EDA industry to discern trends in a variety of design and verification areas. This article will trace some of the key findings for the FPGA segment of the industry over the past several years. I always love it when we get objective data that show that many of the verification techniques we’ve been advocating for years actually serve to reduce the number of bugs that escape into production.

Next, we continue our Portable Stimulus series by another friend and colleague of mine, Matthew Ballance, with “Building a Better Virtual Sequence with Portable Stimulus.” Virtual sequences in UVM can be some of the most challenging pieces of your verification environment to create such that they are both useful and, importantly, reusable. This article will show how you can use Portable Stimulus to create scenarios for your block-level verification environment that are more robust and flexible than writing a virtual sequence. It’ll also show you how to make the virtual sequence customizable and reusable as you move up to the subsystem level – both of which are very hard to do with a virtual sequence in UVM.

In “A New Approach to Low-Power Verification: Power Aware Apps,” you’ll learn how to take advantage of some new information-model package functions in UPF 3.0, together with some Tcl code using the information model API, to create some powerful (pardon the pun) apps to help with various aspects of verifying low-power designs. By following the examples and case studies presented, you should be able to replicate the work on your low-power design project. A version of this article was published at DVCon-Europe 2018, but we wanted to share it with you.

We complete this issue with “Simplifying Mixed-Signal Verification” from my Analog Mixed-Signal colleagues at Mentor. If you’re facing a mixed-signal design for IoT, automotive, communication, industrial or any other application, you’ll find this discussion of AMS methodologies to be very helpful. The article wraps up by showing how Mentor’s new Symphony Mixed-Signal Platform provides the flexibility, accuracy, and performance you’ll need to verify the analog portions of your design alongside your digital design. You’ll also see some advanced debugging capabilities of Symphony that will really make your verification productivity sing.

With the loss of Stu and Joe, our industry has lost two of its finest, not to mention a tremendous amount of institutional knowledge. The rest of us will go on, and as I have learned in the past few months, life is too short not to appreciate the outstanding people we are blessed to work with in this industry, regardless of the companies we work for. Take a moment to hug your family members, and the next time you’re talking to a colleague about work, be sure to ask how things are going outside of work. May God bless you and your families, and Godspeed, Stu and Joe.

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons
CONTENTS

Page 4: FPGA Verification Challenges and Opportunities
by Harry D. Foster – Mentor, A Siemens Business

Page 8: Building a Better Virtual Sequence with Portable Stimulus
by Matthew Ballance – Mentor, A Siemens Business

Page 14: A New Approach to Low-Power Verification: Power Aware Apps
by Madhur Bhargava and Awashesh Kumar – Mentor, A Siemens Business

Page 24: Simplifying Mixed-Signal Verification
by Sumit Vishwakarma – Mentor, A Siemens Business
FPGA Verification Challenges and Opportunities

by Harry D. Foster – Mentor, A Siemens Business

INTRODUCTION

There have been multiple studies on IC/ASIC functional verification trends published over the years. [1][2][3][4] However, there are no published studies specifically focused on Field-Programmable Gate Array (FPGA) verification trends. To address this dearth of information, this article highlights a few key FPGA findings from the 2018 Wilson Research Group Functional Verification Study. The findings from this study provide invaluable insight into the state of today’s FPGA market in terms of functional verification.

The Global Semiconductor Market

The global semiconductor market was valued at $444.70 billion in 2017, of which, $4.7 billion is accounted for by FPGAs. [5][6] The FPGA market is expected to reach a value of $8.8 billion by 2027, growing at a compounded annual growth rate (CAGR) of 6.4% during this forecast period. The growth in this market is being driven by new and expanding end-user applications related to automotive, IoT, telecommunication, industrial, mil/aero, consumer, and emerging AI applications within the data center requiring acceleration.

Historically, FPGAs have offered two primary advantages over ASICs. First, due to their low NRE [7], FPGAs are generally more cost effective than IC/ASICs for low-volume production. Second, FPGAs’ rapid prototyping capabilities and flexibility can reduce the development schedule since a majority of the verification and validation cycles have traditionally been performed in the lab. More recently, FPGAs offer advantages related to performance for certain accelerated applications by exploiting hardware parallelism (e.g., AI Neural Networks).

Growing Design Complexity

The IC/ASIC market in the mid- to late-2000 timeframe underwent growing pains to address increased verification complexity. Similarly, we find today’s FPGA market is being forced to address growing verification complexity. With the increased capacity and capability of today’s complex FPGAs, and the emergence of high-performance SoC programmable FPGAs (e.g., Xilinx® Zynq®, Intel® Arria®, Cyclone®, and Stratix®, along with Microsemi® SmartFusion®), traditional lab-based approaches to FPGA verification and validation are becoming less effective. In this article, we quantify the ineffectiveness of today’s FPGA verification processes in terms of non-trivial bug escapes into production.

FPGA VERIFICATION EFFECTIVENESS

IC/ASIC projects have often used the metric “number of required spins before production” as a benchmark to assess a project’s verification effectiveness. Historically, about 30% of IC/ASIC projects are able to achieve first silicon success, and most successful designs are productized on the second silicon spin. Unfortunately, FPGA projects have no equivalent metric. As an alternative to IC/ASIC spins, our study asked the FPGA participants “how many non-trivial bugs escaped into production?” The results shown in Fig. 1 are somewhat disturbing. In 2018, only 16% of all FPGA projects were able to achieve no bug escapes into production, which is worse than IC/ASIC in terms of first silicon success, and for some market segments, the cost of field repair can be significant.

Figure 1: Non-trivial FPGA bug escapes into production
For example, in the mil-aero market, once a cover has been removed on a system to upgrade the FPGA, the entire system needs to be revalidated.

**FPGA VERIFICATION EFFORT**

In this section, we discuss trends in terms of FPGA project time and resources.

**Percentage of Project Time Spent in Verification**

Fig. 2 shows the percentage of total FPGA project time spent in verification. You can see two extremes in this graph. In general, projects that spend very little time in verification are typically working on designs with a good deal of existing pre-verified design IP, which is integrated to create a new product. On the other extreme, projects that spend a significant amount of time in verification often have a high percentage of newly developed design IP that must be verified.

![Figure 2: Percentage of FPGA project time spent in verification](image)

Overall, we found an increase in average percentage of FPGA project time spent in verification during the period 2014 through 2018. This is an indication of growing design and verification complexity.

**Mean Peak Number of Engineers**

Perhaps one of the biggest challenges today is to control cost and engineering headcount, which means identifying FPGA design and verification solutions that increase productivity. To illustrate the trend in terms of increasing engineering headcount.

Fig. 3 shows the mean peak number of FPGA engineers working on a project.

While, on average, the demand for design engineers is growing at about a 4% CAGR (which is similar growth for IC/ASIC), the demand for verification engineers is growing at about a 10% CAGR. It is worth noting that during the period 2007 through 2014, the IC/ASIC market went through similar growth demands related to verification engineers to address growing verification complexity.[3]

![Figure 3: Mean peak number of FPGA engineers on project](image)

**FPGA VERIFICATION ADOPTION TRENDS**

To address growing verification complexity, we find that many FPGA projects have been forced to mature their verification processes. In this section, we present FPGA trends related to the adoption of various verification techniques, which are fairly standard practice today on most IC/ASIC projects.

The adoption trends for formal property checking (e.g., model checking) and automatic formal applications are shown in Fig. 4. We found that the adoption of formal property checking on FPGA projects is growing at an impressive 21% CAGR, and the adoption of automatic formal applications is growing at a 29% CAGR. Historically, the formal property checking process has required specialized skills and expertise. However, the recent emergence of automatic formal applications provides narrowly focused solutions and does not require specialized
skills for adoption. In general, formal solutions (i.e., formal property checking combined with automatic formal applications) are one of the fastest growing segments in functional verification.

CONCLUSION AND DISCUSSION

In this article, we presented FPGA design and verification trends based on a recent, large industry study. FPGAs have grown in complexity equal to many of today’s IC/ASIC designs. We quantified the impact of this growing complexity in terms of verification effectiveness and effort.

Perhaps the most disturbing finding from this year’s study relates to the number of FPGA projects with non-trivial bug escapes into production. We did find an interesting correlation between the improvement of reduced functional flaws contributing to non-trivial bug escapes, as shown in Fig. 1, and the maturing of FPGA projects’ functional verification processes. The data suggest that projects that are more mature in their functional verification processes will likely experience fewer bug escapes. To test this claim, we partitioned the study participants into two groups: FPGA projects with no bug escapes and FPGA projects that experienced a bug escape. We then examined the percentage adoption of various verification techniques and the results are shown in Fig. 6. These findings are statistically significant in that the group with no bug escapes tended to have higher adoption of various verification techniques, which suggest they are more mature in their verification process. However, what we are unable to measure from our study is how effective a project was in adopting any of these processes. For example, a project that experienced a bug escape could claim that they have adopted functional coverage, yet the fidelity of their functional coverage model might be poor due to their inexperience. From our study data, we are unable to assess successful or effective adoption for any particular verification technique.

Fig. 5 shows the FPGA project adoption trends for various simulation-based techniques from 2012 through 2018, which include code coverage, functional coverage, assertions, and constrained-random simulation.

One observation from these adoption trends is that the FPGA market is maturing its verification processes. This maturity is likely due to the growing complexity of designs.
REFERENCES


Building a Better Virtual Sequence with Portable Stimulus

by Matthew Ballance – Mentor, A Siemens Business

When using the Universal Verification Methodology (UVM), sequences are the primary mechanism by which stimulus is generated in the testbench. Sequences come in two flavors: simple sequences for driving a single interface, and virtual sequences that control more complex behavior. Simple sequences tend to work with a single sequence item, while virtual sequences often spawn off multiple sub-sequences to accomplish their intended task. Good virtual sequences are challenging to create, and even more challenging to reuse in a way not explicitly intended by the original author. Portable stimulus can make creating virtual sequences easier, increase the verification value achieved by running these virtual sequences, and enable more reuse of the description used to create the virtual sequence. This article will walk through an example showing how portable stimulus applies to creating scenarios for a DMA engine.

DMA ENGINE OVERVIEW

The DMA engine used in this article is a relatively simple 8-channel DMA engine. It has a register interface for programming DMA transfers, and two master interfaces for the DMA engine to use for transferring data. Each DMA channel can either perform direct memory-to-memory transfers, or can use peripheral handshake signals to transfer data from memory to a peripheral device, or from a peripheral device to memory.

SIMPLE SEQUENCES

A simple sequence for the DMA engine might look as simple as what is shown in Figure 2.

```vhdl
/**
 * Class: wb_dma_rand_single_transfer_seq
 *
 * Simple random sequence for driving DMA transfers
 *
 * class wb_dma_rand_single_transfer_seq extends
 *    wb_dma_transfer_seq;
 *    `uvm_object_utils(wb_dma_rand_single_transfer_seq)
 *
 * Task: body
 *
 * Override from class
 *
 * virtual task body();
 *    wb_dma_single_transfer_descriptor desc;
 *
 *    repeat(200) begin
 *        desc = wb_dma_single_transfer_descriptor::
 *           type_id::create("desc");
 *
 *        start_item(desc);
 *        if (!desc.randomize()) begin
 *            `uvm_fatal(get_name(), "Failed to randomize sequence item");
 *        end
 *        finish_item(desc);
 *    end
 *
 * endclass
```

Figure 2: Simple UVM Sequence
This simple sequence simply randomizes a sequence item that represents a DMA transfer, then sends that sequence item to the driver to program registers, etc. A simple sequence like this will use a sequence item, like that shown in Figure 3, containing random fields and constraints appropriate to describe a DMA transfer on a given channel.

```plaintext
class wb_dma_descriptor extends uvm_sequence_item;
    `uvm_object_utils(wb_dma_descriptor)
    rand bit[5:0]    channel;
    rand bit        mode;
    rand bit        inc_src;
    rand bit        inc_dst;
    rand bit        src_sel;
    rand bit        dst_sel;
    bit[31:0]        src_addr;
    bit[31:0]        dst_addr;
    rand bit[11:0]   tot_sz;
    rand bit[8:0]    trn_sz;
    rand bit[31:0]   transfer_sz;
    constraint trn_sz_c {
        trn_sz inside [1, 2, 4];
    }

endclass
```

Figure 3: DMA Sequence Item

However, our simple sequence starts to hit limitations when we want to test sequences of operation or operations in parallel. For example, what if we wanted to launch two transfers in parallel to exercise 2 DMA channels at the same time? Without some guidance, we would likely have a mess since more than one of our simple sequences would target the same DMA channel. We could, for example, use a queue to randomly assign channels to the transfers as we set them up, as shown in Figure 4.

```plaintext
virtual task body();
    wb_dma_single_transfer_descriptor d1, d2;
    int channels[] = {0, 1, 2, 3, 4, 5, 6, 7};

    repeat(10) begin
        channels.shuffle();
        d1 = wb_dma_single_transfer_descriptor::type_id::create("d1");
        d2 = wb_dma_single_transfer_descriptor::type_id::create("d2");

        d1.channel = channels[0]
        d2.channel = channels[1]

        fork
            begin
                start_item(d1);
                finish_item(d1);
            end
            begin
                start_item(d2);
                finish_item(d2);
            end
        join
    end

endtask
```

Figure 4: Directed-random Virtual Sequence

This approach is very typical of the directed-random style of virtual sequences. We use procedural methods to avoid conflicts between lower-level transactions that carry out our high-level test intent. This allows us to get some randomness in our scenarios, while following the rules of the system – in this case, the fact that parallel DMA channels must use different channels. While this directed-random approach solves our immediate problem, this code isn’t very reusable or scalable.
For example, we cannot create a new sequence that extends from this sequence but disables the use of certain channels. The declarative nature of portable stimulus allows us to do exactly this type of reuse.

**CRAFTING A PORTABLE STIMULUS VIRTUAL SEQUENCE**

Let’s take a look at how we can model a virtual sequence with Portable Stimulus for our DMA engine. Portable stimulus encapsulates behavior in actions. Our first task is to take a step back and consider which actions we need to create. Our DMA engine can perform three logical functions:

- Transferring data between two regions of memory
- Transferring data from memory to a peripheral device
- Transferring data from a peripheral device to memory

We will represent each of these operations as a PSS action that we can then use in a scenario.

The PSS language allows actions to declare input and output ports, which allow the action to specify what data must be present when it executes, and what data it provides for the use of other actions. One of the first things we need to consider when modeling behavior as PSS actions is what inputs the behavior requires and what outputs it provides.

The memory-to-memory action reads data from a source location in memory and writes it to a destination location. This indicates that we should have an input to represent the source location and an output to represent the destination location. The mem2dev and dev2mem actions will, respectively, read from a region of memory and write to a region of memory. This indicates that they should, respectively, have an input and an output. But, how should we represent the device that these actions interact with? We could represent the device address using an input and output as well, but in this case we will not because we know that the device address is closely linked with the DMA channel used by the mem2dev and dev2mem actions. The device address will be a function of the system address map and the DMA channel in use, not something the test-scenario writer will manipulate directly.

Figure 5 shows our DMA action primitives in diagram form.

```
/**
 * Memory-to-memory transfer
 */
action mem2mem_a : dma_dev_a {
  input data_ref_mem_b  dat_i;
  output data_ref_mem_b  dat_o;

  constraint {
    dat_i.sz <= 4096;
    (dat_i.sz % trn_sz) == 0; // Size must be a multiple of the transfer size
    (dat_i.addr % trn_sz) == 0; // Source and dest addresses must be aligned
    (dat_o.addr % trn_sz) == 0;
    dat_i.sz == dat_o.sz;
    dat_i.ref == dat_o.ref;
  }
}
```

Figure 6 shows the PSS description of our memory-to-memory action. The device-to-memory and memory-to-device actions look very similar, and will not be shown.

Figure 6: Memory-to-memory transfer action

In addition to declaring the inputs and outputs of the action, we must specify the constraints that govern its operation. Specifically, we must specify that the size
of data transferred must be 4k or less—a constraint imposed by the DMA engine. The address of both the source and destination address must be aligned to the DMA’s transfer size as well. Note that the action declaration doesn’t specify anything about how the DMA will be programmed. The action only contains the high-level rules on a DMA transfer. We will add in the mapping to our UVM environment later.

CREATING THE SCENARIO

Now that we have low-level actions that represent the core operations the DMA engine can perform, we can assemble those actions into a compound action that carries out a scenario. The scenario we started with was running two DMA transfers in parallel on different channels. An equivalent scenario is shown in Figure 7 below.

```plaintext
component parallel_xfer_c {
  action parallel_xfer_a {
    wb_dma_c::mem2mem_a m2m_1, m2m_2;
    wb_dma_c::dev2mem_a d2m_1, d2m_2;
    wb_dma_c::mem2dev_a m2d_1, m2d_2;
    pvm::alloc4_a alloc;
    // Ensure parallel operations have unique channels
    constraint unique {
      m2m_1.channel, m2m_2.channel,
      d2m_1.channel, d2m_2.channel,
      m2d_1.channel, m2d_2.channel
    };
    activity {
      bind m2m_1.dat_i alloc.a0;
      bind m2m_2.dat_i alloc.a1;
      bind m2d_1.dat_i alloc.a2;
      bind m2d_2.dat_i alloc.a3;
      repeat (10) {} parallel {
        select {
          m2m_1;
          d2m_1;
          m2d_1;
        }
      }
    }
    }
  }
}
```

**Figure 7: PSS Parallel-Transfer Scenario**

The first portion of the description creates instances of our low-level actions for performing DMA transfers. After creating action instances, we create a constraint to ensure that the channel selected for the DMA operations that eventually run will be different. Next, the activity block composes a scenario from the action instances. The parallel construct in PSS makes it just as easy to describe parallel behavior as in SystemVerilog, and the declarative nature of a PSS description enables us to write constraints that apply across the procedure of our test scenario.

Our PSS test scenario weighs in at 40 lines of code, certainly on-par with the roughly 50 lines required for our SystemVerilog sequence, showing that PSS provides a succinct way to capture scenarios.

COVERAGE

One challenge with directed-random sequences is that it’s difficult to tell whether we’ve generated the stimulus combinations we really care about. Portable stimulus provides a covergroup construct, just like SystemVerilog does. We can add a covergroup to our portable stimulus model to ensure, for example, that we generate all possible scenarios across our parallel scenarios in our virtual sequence (Figure 8 on the following page).
PSS currently supports data-centric coverage, so we still need to represent the scenarios we need to cover as a data relationship. However, here again, the declarative nature of PSS makes it much simpler to capture the coverage relationships we care about. We add in two enumerated-type variables (scen1 and scen2) to represent the scenario being executed, and add constraints to the activity to force these variables to the appropriate value given the scenario being executed. This adds a few lines to our scenario, but will allow us to ensure that we exercise all the scenarios.

**CONNECTING TO THE TESTBENCH**

Thus far, we have focused on test intent – the high-level view of what we want to test. In order to actually run traffic in our testbench environment, we need to run sequences or call APIs in SystemVerilog. PSS provides the exec construct to connect the high-level test intent described in PSS to the low-level test realization described in SystemVerilog, C, or any number of other implementation languages.

The exec block shown in Figure 9 provides a mapping between the fields of the mem2mem_a action and a task in our SystemVerilog virtual sequence named wb_dma_dev_mem2mem. This task is responsible for programming the DMA engine to carry out a transfer on the selected channel.

**CUSTOMIZING OUR VIRTUAL SEQUENCE**

Now, let’s look at the original challenge we faced with our hand-coded virtual sequence: disabling the use of one of the channels. PSS provides us several ways to customize a scenario. The simplest might be to just create a new top-level scenario that inherits from our existing action, and add constraints to force the base action to not use a specific channel (Figure 10). This allows us to reuse our existing scenario, while customizing its behavior.

---

```verilog
enum scenario_e { m2m, d2m, m2d }

action parallel_xfer_a {
    // ...
    action scenario_e scen1, scen2;

    covergroup {
        scen1_cp : coverpoint scen1;
        scen2_cp : coverpoint scen2;
        scen_1x2 : cross scen1_cp, scen2_cp;
    }
    // ...
    activity {
        // ...
        repeat(10) {} parallel {
            sequence {
                select {
                    { m2m_1; constraint scen1==m2m; }
                    { d2m_1; constraint scen1==d2m; }
                    { m2d_1; constraint scen1==m2d; }
                }
                scen1;
            }
            // ...
        }
    }
}

extend action wb_dma_c::mem2mem_a {
    exec body SV = ""wb_dma_dev_mem2mem({devid}), {{channel}}, {{dat_i.addr}},
    {{dat_o.addr}}, {{dat_i.sz}}, {{trn_sz}});"
}

action parallel_xfer_no_ch0_a : parallel_xfer_a {
    constraint {
        m2m_1.channel != 0; m2m_2.channel != 0;
        d2m_1.channel != 0; d2m_2.channel != 0;
        m2d_1.channel != 0; m2d_2.channel != 0;
    }
}
```

---

Figure 8: Adding Coverage

Figure 9: Connecting PSS to SystemVerilog

Figure 10: Customizing the Scenario with Inheritance
The approach we just showed for customizing the scenario targets a specific instance of the scenario. What if we needed to ensure that all instances of the dev2mem action, wherever they appeared in the scenario, never used channel 5? PSS provides a type-extension mechanism that allows us to layer in constraints that will apply to all instances of a given type, as shown in Figure 11.

```plaintext
extend action wb_dma_c::dev2mem_a {
  constraint channel != 5;
}
```

**Figure 11: Customizing the Scenario with Extension**

The advantage (and the disadvantage) of this approach is that the new constraint will apply to all instances of the specified type. In some cases, this is exactly what we want. In other cases, we want to be more targeted. PSS also provides a factory-like mechanism that allows us to override types in very specific contexts.

**REUSE BEYOND THE BLOCK LEVEL**

The atomic actions that we created, such as mem2mem and dev2mem, are necessary to support our block-level scenarios. However, there’s nothing tethering them to our block-level UVM environment except the UVM-specific implementation. PSS makes that simple enough to change with another exec block that maps our test intent to a test realization implemented in C (Figure 12).

```plaintext
extend action wb_dma_c::mem2mem_a {
  exec body C = ```
   wb_dma_dev_mem2mem({devid}), {channel}, {dat_i.addr},
   {dat_o.addr}, {dat_i.sz}, {trn_sz});
```
}
```

**Figure 12: Mapping to Test Realization in C**

The declarative nature of PSS, coupled with appropriate language-specific mapping, makes it very easy to reuse test intent from block level when creating SoC-level scenarios. It doesn’t make sense to reuse all the PSS content we create at block level, since many of the scenarios will test functionality that is specific to block level. However, there are certainly opportunities for reuse at SoC level – infinitely more than with pure SystemVerilog UVM sequences!

**CONCLUSION**

Portable stimulus is often seen as being specific to SoC-level testing. But, as this article has shown, portable stimulus can be very helpful in block-level testing to make scenario creation with UVM virtual sequences simpler, more effective, and makes the scenarios we create more reusable. And, we can get these benefits with roughly the same number of lines of code as are required to capture much simpler UVM sequences.
INTRODUCTION
The effective verification of low-power designs has been a challenge for many years now. The IEEE Std 1801-2015 Unified Power Format (UPF) standard for modeling low-power objects and concepts is continuously evolving to address the low-power challenges of today’s complex designs. One of the main challenges for low-power verification engineers has been the fact that there is a disconnect between the traditional RTL and low-power objects. Users cannot access and manipulate the low-power objects in the same way as they do the RTL objects. Low-power concepts are abstract and complexities arise because of the number of sources like UPF, HDL and Liberty all provide power intent in a low-power design. It has also been seen that the majority of verification time is spent debugging complex low-power issues. There are not too many ways in which users can do self-checking of their designs. As the low-power architecture is complex and the number of power-domains used in designs is high, selective reporting of a part of a design is needed. The lack of an industry standard in this regard has resulted in inconsistencies in the different ad-hoc approaches adopted by different tool vendors.

To keep pace with the increasing complexity of low-power architectures, the IEEE 1801 standard is expanding its gamut of constructs and commands to include more low-power verification and implementation scenarios. In this article we will present some innovative ways of writing Power Aware Apps using the UPF 3.0 information model HDL package functions and Tcl query functions. The article also demonstrates how these Power Aware Apps can help in reporting, debugging and self-checking low-power designs. We will also highlight how these apps will help offer an efficient way to significantly save verification effort and time.

POWER INTENT SPECIFICATION AND BASIC CONCEPTS OF UPF
The IEEE Std 1801-2015 UPF allows designers to specify the power intent of the design. It is based on Tcl and provides concepts and commands which are necessary to describe the power management requirements for IPs or complete SoCs. A power intent specification in UPF is used throughout the design flow; however it may be refined at various steps in the design cycle. Some of the important concepts and terminology used in power intent specification are the following:

- **Power Domain**: A collection of HDL module instances and/or library cells that are treated as a group for power management purposes. The instances of a power domain typically, but do not always, share a primary supply set and typically are all in the same power state at a given time. This group of instances is referred to as the extent of a power domain.

- **Power State**: The state of a supply net, supply port, supply set, or power domain. The power state is an abstract representation of the voltage and current characteristics of a power supply and the operating mode of the elements of a power domain or module instance (e.g., on, off, sleep).

- **Isolation Cell**: An instance that passes logic values during normal mode operation and clamps its output to some specified logic value when a control signal is asserted. It is required when the driving logic supply is switched off while the receiving logic supply is still on.

- **Level Shifter**: An instance that translates signal values from an input voltage swing to a different output voltage swing.
• Hard Macro: A block that has been completely implemented and can be used as-is in other blocks. This can be modeled by a hardware description language (HDL) module for verification or as a library cell for implementation.

UPF 3.0 INFORMATION MODEL
UPF 3.0 has come up with the concept of an information model to represent low-power objects and concepts in a structured and consistent manner. The information model captures the low-power management information taken from the application of low-power UPF commands on a design. The information model consists of a set of objects and various information-bearing properties defined for those objects. It also defines the relationship between the HDL and UPF.

The information model provides a set of well-defined APIs to query the low-power information in either Tcl or HDL. The Tcl APIs can be used to query the static information of a low-power object; e.g. the file/line detail of a UPF object or a list of isolation strategies of a power domain. To get the dynamic information, we can rely on the Tcl APIs provided by verification tools (simulators) to access the dynamic values of the UPF and RTL objects. Together with the static and dynamic information, innovative applications can be written to help with the checking and debugging of a design.

UPF 3.0 also presents the HDL package functions and native HDL object definitions for the UPF object, which has some dynamic information; e.g. power domain, power states, etc. Native object definition and usage is described in the example in the following section. Using these HDL package functions, the user can access the static and dynamic information of low-power object in HDL. This capability can be leveraged to help verification engineers create random verification scenarios.

KEY COMPONENTS OF THE UPF 3.0 INFORMATION MODEL
There are two main components of the information model.

Objects: These are the primary holders of information, accessed by a handle ID. They represent UPF, HDL and the relationship between them.

There are three main classes of objects, namely:
• UPF Objects: Model objects created by UPF.
• HDL Objects: Model objects representing the HDL design.
• Relationship Objects: Objects that model the relationship of UPF and HDL objects; e.g. upfExtentT, upfCellInfoT.
Properties: These are the basic pieces of information accessed by a property ID; such as UPF_NAME, UPF_ISOLATION_STRATEGIES.

**UPF 3.0 HDL PACKAGE FUNCTIONS**

**Native HDL Representation**

UPF 3.0 defines the native HDL representation for objects that have dynamic properties. The native HDL representation is the struct/record type in HDL that contains two fields:

1. A value field corresponding to the dynamic property of the object.
2. A handle or reference to the UPF object to allow access of other properties of the object.

The following HDL types are supported with a native HDL representation:

<table>
<thead>
<tr>
<th>Type Name</th>
<th>SV Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>upfPdSsObjT</td>
<td>struct { upfHandleT handle; upfPowerStateObjT current_state; } upfPdSsObjT</td>
</tr>
<tr>
<td>upfPowerStateObjT</td>
<td>struct { upfHandleT handle; upfBooleanT is_active; } upfPowerStateObjT</td>
</tr>
<tr>
<td>upfBooleanObjT</td>
<td>struct { upfHandleT handle; upfBooleanT current_value; } upfBooleanObjT</td>
</tr>
<tr>
<td>upfSupplyObjT</td>
<td>struct { upfHandleT handle; upfSupplyTypeT current_value; } upfSupplyObjT</td>
</tr>
</tbody>
</table>

*Table 1*

In Table 1, the field representing the dynamic property of the object has been highlighted in bold. For example, for a power domain or supply set, the associated dynamic property is the current power state of the power domain which is represented by the current_state field of the struct in the SV native representation of the upfPdSsObjT type. The other field is a handle to the low-power object, which has all the static information about the object; e.g., object name, its creation scope, and file/line information.

Table 2 summarizes the UPF 3.0 information model objects with native HDL information. The HDL types defined in Table 1 are used to represent the dynamic properties of these objects.

**HDL Package Functions**

UPF 3.0 provides a number of HDL package functions that are used to access low-power objects and their properties. These are broadly classified in the following five classes of functions:

1. **HDL access functions**: These are the basic functions to access the low-power objects and properties. For example, the following access function can be used to get the handle of an object.

   ```
   upfHandleT pd = upf_get_handle_by_name("/top/dut_i/pd")
   ```

   - returns the handle of power domain ‘pd’

   One of the key HDL access functions is the "upf_query_object_properties".

   ```
   upfHandleT scope = upf_query_object_properties(pd, UPF_CREATION_SCOPE)
   ```

   This function returns the handle to a property corresponding to an enumerated value passed as property, e.g.,

   ```
   upfHandleT scope = upf_query_object_properties(pd, UPF_CREATION_SCOPE)
   ```

   - returns the creation scope of power domain with handle ‘pd’.

2. **Immediate read access HDL functions**: All the objects in the UPF 3.0 information model allow read access to its properties. In the case of dynamic properties, these functions return the current dynamic value/state of that property when this function is called; for example:
### 3. Immediate write access HDL functions:

Some objects of the information model allow the immediate write access only if they don’t have an existing driver. This allows the manipulation of low-power objects from a testbench or simulation model. For example: `supply_on("supply_net_name", value)`. The following objects allow immediate write access:

- **upfPowerStateT**
- **upfLogicNetT**
- **upfLogicPortT**
- **upfSupplyNetT**
- **upfSupplyPortT**

These functions are a powerful tool for users to manipulate low-power objects during simulation from a testbench.

### 4. Continuous access HDL functions:

These functions enable continuous monitoring of dynamic values of an object in the information model. It enables the user to trigger an always block or process statement using the dynamic values of the low-power objects.

```tcl
upfSupplyObjT vdd_monitor;
up_create_object_mirror("/top/dut_i/vdd", "vdd_monitor");
```

### 5. Utility functions:

These functions are general utility functions to assist users. For example:

```tcl
upfClassIdE upf_query_object_type(upfHandleT handle)
```

- returns the type of a handle. Using this, the user can find out if the object is a power domain, supply set, or some other low-power object.

### UPF 3.0 TCL API

The UPF 3.0 information model defines a number of Tcl query commands to access low-power objects and properties. UPF 3.0 introduced a Tcl-based Information Model Application Programmable Interface. This API can be used to access power-aware information:

```tcl
upfHandleT ps = upf_get_handle_by_name("/top/dut_i/pd.power_state_on")
upfHandleT ps_active_hndl = upf_query_object_properties(ps, UPF_IS_ACTIVE)
integer ps_on_value = upf_get_value_real(ps_active_hndl)
```
**Basic Tcl API**

## To get various attributes on a given object

```tcl
upf_query_object_attributes obj -attribute <attr_name> -detailed
```

## To get the type of the object

```tcl
upf_query_object_type obj
```

## To check if an object belongs to a particular group

```tcl
upf_object_in_group obj -group <group_id>
```

## To get the full hier path of an object relative to given scope

```tcl
upf_query_object_pathname obj -relative_to <object_handle>
```

**Example:**

```tcl
upf_query_object_properties /tb/top1/pd.iso_strategy_property
upf_clamp_value
```

An object handle is used to access any power aware information. A handle can be a pathname; e.g., `/tb/top1/PD1.ret1` or some tool assigned ID; e.g., `#UPFEXTENT1234`.

**Building Tcl Based Power Aware Apps Using Tcl APIs**

Tcl based apps are nothing more than Tcl procedures users can write for special requirements, such as reporting, debugging or checking the design. Building blocks of Tcl procedures (Tcl Power Aware Apps) include:

- Four basic UPF 3.0 APIs which can be used to access any UPF information.
- Tcl APIs provided by verification tools (simulators) to access the dynamic data.

Once an app is built using the above APIs, it can be run either in a verification tool environment, at their static time, to get static information or in post-simulation to get both static and dynamic waveform data. The following is an example of how the user can build an app to find the source of corruption/retention of a signal and see the values of these signals.

**UPF:**

```tcl
set_scope /tb/chip_top
create_power_domain PD_CAMERA -include_scope
create_supply_net pd_pwr -domain PD_CAMERA
create_supply_set ss -function [power pd_pwr]
associate_supply_set ss -handle PD_CAMERA.primary
```

Here the signal in question is `/tb/chip_top/c`, which is corrupted at some time instance in simulation. The goal is to find the source of corruption of this signal: #aliasing upf_query_object_properties to simple name such as alias alias query upf_query_object_properties.

### # Step 1: Get the properties of the signal

```tcl
examine tb/chip_top/c
# 1'bx
query tb/chip_top/c
#{ [upf_name c] [upf_parent /tb/chip_top] [upf_cell_info #UPFCELL0_71653#] [upf_port_dir UPF_DIR_OUT] }
```

### # Step 2: Get the properties of cell applied on that signal

```tcl
query #UPFCELL0_71653#
#{ [upf_cell_kind upf_cell_corrupt] [upf_hdl_cell_kind upf_hdlcell_comb] [upf_cell_origin upf_origin_inferred] [upf_source_extents #UPFEXTENT2130711#] }
```

### # Step 3: Get the properties on source extent (extent of power domain, retention strategy, etc.) of the cell

```tcl
query #UPFEXTENT2130711#
#{ [upf_hdl_element tb/chip_top] [upf_object tb/chip_top/PD_CAMERA/*power domain*/] }
```

### # Step 4: Get the supplies of the upf_object (power domain, retention strategy etc.)

```tcl
query /tb/chip_top/PD_CAMERA -property upf_supply_set_handles
#/ tb/chip_top/PD_CAMERA.primary /tb/chip_top/PD_CAMERA.default_retention /tb/chip_top/PD_CAMERA.default_isolation
```
EXAMPLES AND CASE STUDIES

When using Tcl APIs and HDL package functions a number of novel objectives can be achieved. This section captures some of the innovative Power Aware Apps based on information model APIs to solve practical low-power verification problems, which otherwise are relatively difficult to solve and users have to rely on tool vendors for those specific features. The article captures a few useful applications. However, along similar lines, users can write their own application for various needs.

There are two main approaches to access and manipulate the low-power objects and properties. First we will describe Power Aware Apps using HDL package functions and then Power Aware Apps using Tcl APIs.

Power Aware Apps Based on HDL Package Functions

Low-Power App 1:
(Coverage App) Coverage of a Low-Power Design Using HDL Package Functions

In a low-power design, it is of utmost importance for a verification engineer to ensure that all IP in the design behaves properly in OFF/ON mode. They also need to ensure that transitions from ON->OFF and OFF->ON have been verified. This requirement can be achieved by creating a coverage infrastructure to ensure the full coverage of the simstate property of the primary supply set of all power domains.

The aim of this application is to do simstate coverage (Normal/Corrupt) of all the power domains in the design. The application will cover the NORMAL->CORRUPT and CORRUPT->NORMAL transitions for each power domain in the design. The following shows how UPF 3.0 HDL package functions can be used to achieve this.

Step 1: Mirror UPF objects to HDL objects

```haskell
query /tb/chip_top/PD_CAMERA.primary.power
#: { {upf_name power} {upf_creation_scope /tb/chip_top/ PD_CAMERA} {upf_parent /tb/chip_top/PD_CAMERA} {upf_ref_kind upf_ref_power} {upf_ref_object /tb/chip_top/ pd_pwr} }
```

```haskell
// Native HDL representation for power domains
typedef struct {
    upfHandleT handle;
    upfSimstateT simstates;
} upfPdObjT;
```

Use the mirror function to continuously monitor the simstate of all the power domain in the design:

```haskell
pd_iter = upf_get_all_power_domains();
pd_hdl = upf_iter_get_next(pd_iter);
while (pd_hdl) begin
    pd_obj = "power_domain_objs["
    pd_cnt_str.itoa(pd_cnt);
    pd_obj = {pd_obj, pd_cnt_str};
    pd_obj = {pd_obj, "]"};
    upf_create_object_mirror (upf_query_object_pathname (pd_hdl), pd_obj);
    pd_cnt++;
    pd_hdl = upf_iter_get_next(pd_iter);
end
```

Step 2: Covergroup definition for state and transition coverage

```haskell
covergroup PD_STATE_COVERAGE (string pd_name, ref upfSimstateE simstate) @( simstate);
CORRUPT: coverpoint simstate
    { bins ACTIVE = [CORRUPT]; }
NORMAL: coverpoint simstate
    { bins ACTIVE = [NORMAL]; }
COA: coverpoint simstate
    { bins ACTIVE = [CORRUPT_ON_ACTIVITY]; }
option.per_instance = 1;
type_option.merge_instances = 0;
type_option.merge_addresses = 0;
type_option.merge_interface = 0;
option_comment = pd_name;
endgroup

continued on the following page
```
Step 3: Instantiation of coverage module:

```verilog
covergroup PD_TRANS_COVERAGE (string pd_name, ref upfSimstateE simstate) @ (simstate);
TRANSITION_COVERAGE coverpoint simstate {
    bins OFF_to_ON = (CORRUPT => NORMAL);
    bins ON_to_OFF = (NORMAL => CORRUPT);
    bins ON_COA_OFF = (NORMAL => CORRUPT_ON_ACTIVITY => CORRUPT);
} option.per_instance = 1;M
    type_option.merge_instances = 0;
    option.comment = pd_name;
endgroup

PD_STATE_COVERAGE pd_state_cov[$];
PD_TRANS_COVERAGE pd_trans_cov[$];
initial begin
    for (int i = 0; i < pd_cnt; i++) begin
        pd_state_cov[i] = new
            (upf_query_object_pathname(power_domain_objs[i].handle), power_domain_objs[i].simstate);
        pd_trans_cov[i] = new
            (upf_query_object_pathname(power_domain_objs[i].handle), power_domain_objs[i].simstate);
    end
end
```

Monitor the simstates of a power domain: User can also monitor the simstates of one or more power domains of interest.

```verilog
always @(power_domain_objs[0].simstate) begin
    $display ($time, "Power Domain '%s' simstate changed to '%s'", identstr,
        upf_get_value_str(power_domain_objs[0].simstate));
end
```

Low-Power App 2:
Write Function to Print Current Simstates of a Power Domain Using HDL Package Functions

Users can write the following set of functions to print the simstates of all the power domains of the design at any instance of time in simulation.

```verilog
function string get_simstate_str(power_state_simstate simState);
    if(simState == NORMAL) get_simstate_str = "NORMAL";
    if(simState == CORRUPT) get_simstate_str = "CORRUPT";
    else if(simState == CORRUPT_ON_ACTIVITY) get_simstate_str = "CORRUPT_ON_ACTIVITY";
    else if(simState == CORRUPT_STATE_ON_ACTIVITY) get_simstate_str = "CORRUPT_STATE_ON_ACTIVITY";
    else if(simState == CORRUPT_STATE_ON_CHANGE) get_simstate_str = "CORRUPT_STATE_ON_CHANGE";
    else if(simState == CORRUPT_ON_CHANGE) get_simstate_str = "CORRUPT_ON_CHANGE";
endfunction
```

```verilog
function reg print_current_state_of_hndl(upfHandleT hndl);
    upfHandleT state_hndl, simstates_hndl, pd_nm_hndl,
        state_nm_hndl;
    upfHandleT line_no_hndl, file_nm_hndl, iter_hndl;
    int simstate;
    state_hndl = upf_query_object_properties(hndl, UPF_CURRENT_STATE);
    pd_nm_hndl = upf_query_object_properties(hndl, UPF_NAME);
    file_nm_hndl = upf_query_object_properties(hndl, UPF_FILE);
    line_no_hndl = upf_query_object_properties(hndl, UPF_LINE);
    state_nm_hndl = upf_query_object_properties(state_hndl, UPF_NAME);
    simstate_hndl = upf_query_object_properties(state_hndl, UPF_SIMSTATE);
    simstate = upf_get_value_int(simstate_hndl);
    $display ($time, "Power domain: %s (%s:%0d), Current simstate: %s", identstr,
        upf_get_value_str(pd_nm_hndl), upf_get_value_str(file_nm_hndl), upf_get_value_int(line_no_hndl), get_simstate_str(simstate_hndl));
    return 1;
endfunction
```

```verilog
function reg print_pd_simstates();
    upfHandleT pd_iter;
    upfHandleT pd_hndl;
    int pd_cnt;
    int pd_iter;
    pd_iter = upf_get_all_power_domains();
    pd_hndl = upf_iter_get_next(pd_iter);
    while (pd_hndl) begin
        print_current_state_of_hndl(pd_hndl);
        power_domains[pd_cnt++] = pd_hndl;
        pd_hndl = upf_iter_get_next(pd_iter);
    end
    return 1;
endfunction
```
**Power Aware Apps Based on Tcl APIs**

**Low-Power App 3:** (Reporting App) UPF query_* commands

Reporting is an essential part of the low-power verification process. Once the power intent is captured in a UPF file, it is important for the verification and design engineers to know that it has been captured as the original intention. This requirement can be fulfilled by query_* procedures. These query commands can query the UPF data as interpreted by the verification tools and stored in the information model. The output of query commands can be used to do selective reporting.

```tcl
interp alias {} query {} upf_query_object_properties;
interp alias {} type {} upf_query_object_type;
interp alias {} group {} upf_object_in_class;
interp alias {} name {} upf_query_object_pathname

proc query_port_direction {{port_name ""} args} {
    set direction [query $port_name -property upf_port_dir]
    switch $direction {
        UPF_DIR_IN {set result "in"}
        UPF_DIR_OUT {set result "out"}
        UPF_DIR_INOUT {set result "inout"}
        default { set result ""}
    }
    return $result
}
Usage:
query_port_direction /tb/t/a/vdd
```

**Result:** "in"

```tcl
proc query_power_domain {{domain_name} args} {
    if {[type $domain_name] == "upfPowerDomainT"] {
        set property [query $object -property upf_model_name]
        set element [lindex $property 0]
        if {[type $element] == "upfHdlScopeT"] {
            lappend result "model [lindex $property 0]"
            lappend result "file [lindex $property 1]"
            lappend result "line [lindex $property 2]"
        } elseif {[type $element] == "upfHdlPortBitT"] {
            set parent [query $element -property upf_parent]
            lappend result "parent_model [lindex $parent"
            lappend result ""domains"
            lappend result "elements [lindex $element]"
            return $result
        } else {
            return "ERROR : Invalid object. Expecting 'HdlPort'
                or 'Instance'"
        }
        set attr [query $object -property upf_hdl_attributes]
        #return $result
    } else {
        return "ERROR : Invalid arguments. arg
            '$domain_name' not a
```

**Low-Power App 4:** (Debug/Reporting App)

Get All Attribute Information

In a low-power design, along with the UPF file, some of the power intent can be present in a Liberty file. The Liberty information is annotated on RTL objects using attributes which can then be further updated using the UPF command set_port_attributes. In a low-power design containing hard macros, attribute information plays a vital role when debugging or reporting. These low-power attributes can be present on an instance or port of an instance. This low-power app can be used on any signal or instance in the design to get the attribute information and the respective signal values if wave data is available.

```tcl
proc pa_query_attributes {{object} args} {
    set result ""
    if {[type $object] == "upfHdlScopeT"] {
        lappend result "model [lindex [query $object -property upf_model_name] 0]"
        lappend result "file [lindex [query $object -property upf_model_name] 1]"
        lappend result "line [lindex [query $object -property upf_model_name] 2]"
    } elseif {[type $object] == "upfHdlPortBitT"] {
        set parent [query $object -property upf_parent]
        lappend result "parent_model [lindex [query $parent -property upf_hdl_attributes] 0]"
    } else {
        return "ERROR : Invalid object. Expecting 'HdlPort'
                or 'Instance'"
    }
    set attr [query $object -property upf_hdl_attributes]
    #return $result
}
```

continued on the following page
Low-Power App 5:  
(Debugging App) Trace Drivers of UPF Objects  
For a low-power design consisting of RTL along with UPF, all the supply network including creation of ports, nets, and their connections is written inside the UPF file. Debugging of the supply network is a major problem that many verification engineers come across. This low-power app is useful as it can trace the driver of any UPF objects along with printing the values of all the ports and nets in the path. The input of this app can be either a UPF created supply, Liberty created supply pin, or a supply defined in HDL.

```tcl
if {$attr != ""} {
    lappend result "attributes $attr"
} else {
    lappend result "attributes NO_ATTRIBUTE_SET"
}

# printing result
foreach i $result {
    if {[lindex $i 0] != "attributes"} {
        puts "$i"
    } else {
        puts "[(lindex $i 0)]"
        set i [replace $i 0 0]
        foreach j $i {
            puts "\$j"
        }
        puts "i"
    }
}
return $result
}
```

**Usage:**
```
pa_query_drivers tb/dut/ab5
```

**Result:**
```
{model analog}
{file analog.sv}[line 53]
{attributes
  > [mspa_cell_functionality pa [analog.lib][28]]
  > [level_shifter_type HL [analog.lib][28]]
  > [is_level_shifter true [analog.lib][28]]
}
```

proc pa_query_drivers {object} {
    set fanin $object
    set driver {}
    append driver $object
    while {[query $fanin -property upf_fanin_conn] != ""} {
        set driver [concat $driver [examine $fanin] <-]
        if {[llength [query $fanin -property upf_fanin_conn]] > 1} {
            set resolution [query $fanin -property upf_resolve_type]
            set fanin [query $fanin -property upf_fanin_conn]
            foreach index $fanin {
                set driver [concat $driver "$index
                    [examine $index]"
                ]
            }
            set driver [concat $driver "$resolution"]
        } else {
            break
        }
    }
    set driver [concat $driver "[query $fanin -property upf_
        fanin_conn]]"
    set fanin [query $fanin -property upf_fanin_conn]
    if {[llength $fanin] < 2} {
        set driver [concat $driver [examine $fanin]]
    }
    return $driver
}

**Usage:**
```
pa_query_drivers /tb/t1/m1/b1/vd_bot
```

**Result:**
```
/tb/t1/m1/b1/vd_bot (OFF 0) <- /tb/t1/m1/b1/vport1_bot (OFF 0)
<- /tb/t1/m1/vd_mid (OFF 0) <- /tb/t1/m1/vport1_mid (OFF 0)
<- /tb/t1/vd_top (OFF 0) <- /tb/t1/vport2_top (OFF 0)
/tb/t1/vport1_top (OFF 0) (PARALLEL)
```

POSSIBLE USAGE OF HDL PACKAGE FUNCTIONS AND TCL APPS

As observed in the above sections, there are two main approaches to access and manipulate the low-power objects and properties. One is HDL package functions and the other is to use the Tcl query commands. There are different scenarios in which one or the other approach is suited. Table 3 summarizes the various usage scenarios where HDL package functions or Tcl query commands can be used.
CONCLUSION
Verification engineers can use the proposed verification approach to achieve early low-power verification closure. The approach mentioned in this article using the UPF 3.0 information model provides a number of benefits. This approach is consistent across tool vendors as it is based on the UPF 3.0 standard. The learning curve for users is not steep. Also the user scripts created to use the proposed solution are easily scalable to bigger and more complex design scenarios.

Low power designs today are incredibly complex with intricate power architectures. Thorough low-power verification is a must for such designs, as any power bug left can cause a huge setback. In this article we have discussed the challenges of current low-power verification methods and how those challenges can be addressed better with UPF 3.0. We discussed the concepts of the UPF 3.0 information model and API to represent and access the low power information which is the result of the application of UPF on the design. We also presented examples and case studies showing how the UPF 3.0 information model concepts can be used to develop a more consistent, robust, and scalable low-power verification platform. In the end we discussed the benefits of using the proposed approach over conventional approaches.

Following the original publication of this article, there have been further developments that affect how SV-based testbenches that leverage the UPF SV API could be used in simulation. These additions are part of the forthcoming IEEE1801-2018 standard:

a) A new relationship singleton object has been added to the UPF information model. Using this relationship object, an SV testbench can retrieve all power domains in a design.

b) The native HDL type upfPdSsObjT (implemented as struct in SV) now has an additional field in upfSimstate current_simstate that reflects the current simstate of a supply_set or, in the case of a power domain, the current simstate of the domain’s primary supply set handle.

REFERENCES
[4] "Awashesh Kumar, Madhur Bhargava", Unleashing the Power of UPF 3.0: An innovative approach for faster and robust Low-power coverage, DVCon India 2017

<table>
<thead>
<tr>
<th>HDL Package functions</th>
<th>Tcl Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Useful for coverage of low-power objects</td>
<td>Useful in selective reporting</td>
</tr>
<tr>
<td>Useful for transition coverage of power states</td>
<td>Batch mode debug (live sim or post sim)</td>
</tr>
<tr>
<td>Directed assertions on low-power objects (e.g. simstates of power domain)</td>
<td>Power aware static checking</td>
</tr>
<tr>
<td>Dynamic checks involving lower-power objects</td>
<td></td>
</tr>
</tbody>
</table>

Table 3
INTRODUCTION
Mixed-signal design is the art of taking real world analog information, such as light, touch, sound, vibration, pressure, or temperature, and bringing it into the digital world for processing. The growth in mixed-signal design has been fueled by an increasing demand for AI, automotive, IoT, communication, and industrial hardware applications (Figure 1).

Designers are finding that these new markets for mixed-signal SoCs are revealing the shortcomings of legacy mixed-signal verification methodologies and tools. Some of these key issues are explored below.

THE “DIVIDE AND CONQUER” APPROACH IS RUNNING OUT OF STEAM
The push towards digitization & automation in real-world applications such as Industrial IoT and autonomous driving has exponentially increased the need for electronic components. These real-world use cases require the need to translate physical signals to digital domain and vice versa to achieve functional correctness in real-time. Autonomous driving is one example where we have LIDARS & active sensors that capture the physical status of surroundings. These physical signals are analog in nature and are fed in to the ADAS system in the vehicle to make real-time decisions using extensive digital computing. This feedback loop is very critical for making autonomous driving possible. These practical real-world applications require the use of Mixed-Signal ICs with Analog & Digital components.

While designers must simulate the analog and digital subsystems together, the simulation algorithms are fundamentally different.

High precision circuits, in many cases, require very accurate SPICE simulation to ensure proper operation, while digital circuits can rely on HDL simulators that run much faster. As a result, the analog simulation will typically dominate the overall system simulation time, as Table 1 below explains.

<table>
<thead>
<tr>
<th>Digital Simulator</th>
<th>Analog Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital devices and circuits operate in a discrete domain, where the device pins and circuit nodes have a binary state of either HIGH (1), LOW (0), X (unknown), or Z at any given instant of time.</td>
<td>Analog devices and circuits operate in a continuous domain, where node voltages and branch currents can take arbitrary (positive or negative) values. They vary continuously, as a function of time.</td>
</tr>
<tr>
<td>The algorithm solves logical expressions sequentially by triggering events. Simulation is event-driven and discrete-domain based.</td>
<td>Applying Kirchhoff’s laws, the algorithm solves the entire analog system matrix at every time step.</td>
</tr>
<tr>
<td>There is a defined signal flow from input to output.</td>
<td>There is no defined signal flow in any direction and circuit elements can instantaneously influence any other element in the matrix.</td>
</tr>
</tbody>
</table>

Table 1: Digital solver versus analog solver
Accurate verification of these Mixed-Signal ICs has become even more stringent to satisfy real-world applications. Mixed-signal ICs needs to be verified thoroughly from System to individual blocks in a cohesive and accurate manner. There are several challenges to meet the mixed-signal verification requirements.

Bridging the methodology gap between Analog & Digital verification is of paramount importance to meet accuracy requirements and time-to-market needs. Mixed-signal verification ideally needs verification engineers with expertise in both analog and digital verification methodologies which is very difficult to achieve. Digital verification engineers are used to a more automated and top-down regression type verification methodologies.

This is in contrast to analog verification, which typically involves bottom-up methodology.

Efficient mixed-signal verification methodologies needs a hybrid approach. For example in a high-speed IO design such as SerDes, the analog designers will need to verify the PLL jitter in the context of the top-level design. The top-level digital verification engineer will only need a functional PLL model to verify the system level requirements. An efficient mixed-signal platform should be flexible in terms of setup, ease-of-use, unified debugging environment and provide necessary abstraction of design data for SoC level & block-level verification requirements. Typical Mixed-Signal IC verification teams comprises of analog designers, digital designers, modeling experts to abstract analog circuit functions & system aggregators (Figure 2).

These 4 different function will need to work in cohesive manner to meet the verification requirements.
UNDERSTANDING MIXED-SIGNAL METHODOLOGIES

Mixed-signal design plays a critical role in ICs that have high-performance, low-noise analog interfaces connected to large digital signal processing blocks. This is the case in networking and wireless communication applications, where an analog/RF signal is converted to digital, processed, and converted back to analog. Mixed-signal design is now a key requirement for wearable, IoT, and automotive applications and the number of periphery sensors is growing in these applications. Sensors are proliferating to the periphery of individual blocks and they will eventually propagate to the periphery of the SoC itself. The various parts of the circuit must communicate more quickly in order to meet the increasing performance requirements for the overall system. In AI chips, for example, communication between processors and between memory and processors occurs at hundreds of gigabytes per second.

The combination of growing design size, dramatic changes in analog-digital integration and complexity, and the variability introduced at advanced process nodes threatens mixed-signal chip yield, performance, and longevity.

As a result, mixed-signal methodology has evolved continuously over the years to address these challenges. The two primary design methodologies are “analog driven” bottom-up and “digital driven” top-down.

ANALOG DRIVEN METHODOLOGY

The development of behavioral models (Figure 3) happens late in design cycle, when transistor-level simulations take too long to run. This begins to happen when designers integrate multiple SPICE blocks together. The amount of time and effort needed to create accurate models may push the project off schedule and model quality could be compromised.

After the model matures, it is easy to make minor tweaks to customize it for new specifications on the next version of the design. While turn-around time for project completion in an analog centric approach depends heavily on SPICE simulator performance, accurate modeling using Real Number Model (RNM) has recently seen good traction to expedite verification cycles.

Figure 3: Behavioral model validation using optimization
DIGITAL DRIVEN METHODOLOGY

Digital verification techniques have been at the forefront in terms of new methodologies. Traditional analog verification still uses the bottom-up methodology and user generated test benches for functional and performance verification. A good engineering methodology uses hybrid approach which is a balance between top-down and bottom-up. In an organization, design groups can pick one approach over another based on their technical expertise, project schedule, and the scope of the project.

A good engineering methodology uses a “meet-in-the-middle” approach which is a balance between top-down and bottom-up. In an organization, design groups can pick one approach over another based on their technical expertise, project schedule, and the scope of the project.

MODERN MIXED-SIGNAL VERIFICATION CHALLENGES

As design complexity multiplies, verification complexity explodes. Verification is now the art of applying many unique methodologies for each class of sub-design within an IC. In the pure digital verification space, the advent of new technologies, such as constrained-random data generation, assertion-based verification, coverage-driven verification, formal model checking, and intelligent testbench automation, have changed the way teams achieve functional verification productivity. However, most of these advances and new technologies have not been perfected for the analog domain and have not been extended to verify mixed-signal designs (Figure 4).

Mixed-signal verification is evolving at a slower pace and it faces some key challenges which are explored in the figure above.

PERFORMANCE

In mixed-signal simulations, the analog domain requires analog solvers and digital domain requires digital solvers. Inherently, digital solvers are event based and are exponentially faster than analog solvers that require very small timesteps. In order to overcome the analog simulation bottleneck, analog circuitry with transistors are replaced by behavior models based on real number models. However, behavior models cannot meet the high accuracy needed in today’s mixed-signal designs. Therefore, the need for high performance analog simulator is essential as well.
Teams face a key question: how much time should the team invest in creating models and is the return on investment worth the resources and time spent? For complex analog designs at smaller technology nodes, design teams still prefer to use a SPICE simulator which guarantees accuracy. This puts immense pressure on the EDA tool provider to offer a solution that can satisfy both performance and accuracy requirements that design teams demand.

**COMPLEX USE MODEL**

Traditional mixed-signal simulators available in the market today evolved more as an after-thought to combine analog and digital verification needs. There was little thought given to provide a simple use model. Most of the solutions available now have very complex set-up and do not provide a seamless interface to move between the digital centric flow and analog centric flow. Ideal mixed-signal solution should have a unified interface that is intuitive for analog and digital verification engineers.

**MIXED-SIGNAL DEBUG**

In mixed-signal design, errors most often occur at the interfaces between analog and digital blocks. Frequently, bugs in the interfaces are identifiable only at a higher level of hierarchy, sometimes at the I/O pins (Figure 5).

Mixed-signal debug gets even more complicated when the design employs advanced, low-power techniques. For example, data corruption in a digital block due to faulty power sequencing can pass to an analog block, resulting in erroneous voltage conversion. Scenarios like this are difficult to debug by analog designers who are unaware of digital low-power techniques.

**MULTIPLE ENGINE AND DESIGN ENVIRONMENT**

There is no single methodology for verifying a SoC. Teams employ multiple simulation engines in different contexts through varying verification flows. The design verification methodology changes from low-level design up to the system level. Migrating designs between engines can take months and a tremendous amount of effort. Additionally, the cost of migrating a SoC verification flow grows exponentially with design size.

Digital and analog teams working on a common SoC engage in multiple activities that involve different tools without any means of effective communication between them. It gets challenging when the time comes to integrate the IC for full mixed-signal verification because no designer has complete insight into the use models and configurations of all the tools in the flow. What is needed is a platform that can easily fit into any design environment and verification methodology for efficient verification closure.

---

**Figure 5: Mixed-Signal bugs**
INTRODUCING SYMPHONY
The next-generation Symphony Mixed-Signal Platform provides designers with unprecedented flexibility for choosing their own design methodology: bottom-up, top-down, or any combination of the two. Designers can make intelligent trade-offs by choosing detailed, continuous analog models or SPICE for high accuracy and discrete behavioral models for simulation speed performance.

Symphony Mixed-Signal Platform is the industry’s fastest and most configurable mixed-signal solution to accurately verify design functionality, connectivity, and performance across analog/digital (A/D) interfaces at all levels of the design hierarchy for all IC applications (Figure 6).

ACCURACY AND SPEED ADVANTAGE
Symphony’s modular architecture leverages Mentor’s Analog FastSPICE (AFS) circuit simulator to provide fast mixed-signal simulation performance with nanometer (nm) SPICE accuracy and capacity of 20M SPICE elements. With certified accuracy by the world’s leading foundries, AFS delivers 5-10x faster performance than traditional SPICE and 2-6x faster performance than parallel SPICE simulators. Symphony has been proven on a wide range of ICs and IC subsystems including ADCs, transceivers, PMICs, multi-GHz PLLs/DLLs, and sensors (Figure 7).

FULLY CONFIGURABLE ARCHITECTURE
Symphony’s unique, fully configurable fit-to-purpose architecture (Figure 8 on the following page) and design-aware technologies provide verification teams with the ability to integrate and optimize their mixed-signal flow for any application. Symphony works with all leading digital solvers, including Mentor’s Questa®, allowing designers to maximize reuse of their existing verification infrastructure including testbenches, stimuli, scripts, post processing, encrypted IP blocks, and A/D netlists.
BEST IN CLASS USABILITY
Symphony delivers the industry’s most intuitive use model with a simple configuration file format and command structure that allows full reuse of existing digital/analog solver command line arguments:

Symphony is also integrated into the leading schematic capture environments and works with both digital and analog centric flows. Symphony offers extensive A/D boundary element (BE) support covering all signal types and multiple power domains, including those with dynamic supplies.

POWERFUL MIXED SIGNAL DEBUG CAPABILITIES
Symphony’s state-of-the-art debugging capabilities improve efficiency of design error tracing across A/D interfaces. It offers a powerful debugging cockpit called the BE Browser to give designers the visual BE context needed to trace back design errors to their sources (Figure 9 on the opposite page).

ADVANCED FEATURES
Symphony offers a powerful set of features designed to increase the verification scope beyond the pure functional realm, into verifying performance aspects of the IC. For example, device noise is critical in nanometer-scale CMOS processes, where it often fundamentally limits circuit performance.

Symphony enables gauging the noise impact of analog blocks while preserving the A/D feedback of their parent subsystem, which increases the accuracy of the measurement. Symphony leverages AFS’s full-spectrum device noise analysis capability to accurately predict ~25 dB device noise impact correlating within < 1.5 dB of silicon measurement.

Symphony’s Hi-Z checking capability allows designers to detect when a mixed-signal net goes into a ‘Z’ state, enabling the testbench and the digital control logic to respond correctly to the ‘Z’ state. Symphony’s Save/Restart functionality increases designer productivity for specific applications by reducing full simulation restarts.
CONCLUSION
Verification of mixed-signal IP and SoCs is challenging. As complexity grows, verification engineers cannot rely on the “divide and conquer” approach of verifying digital and analog blocks individually and then stitching them together for full-chip verification. Verification teams need to run an increasing number of mixed-signal simulations at the top level as well as at the subsystem to make sure there are no functional errors due to interactions between analog and digital domains. Even functional errors caused by trivial bugs, such as wrong connectivity, inverted polarity, and incorrect bus order can result in costly silicon re-spins.

While digital verification techniques have evolved over the years, mixed-signal verification is still catching up. Modern analog modeling approaches are developed, but the need for accuracy still takes top priority when it comes to analog verification. Additionally, the analog and digital design environment and verification use models are different and it is challenging to integrate the IC for full mixed-signal verification. To address these challenges, mixed-signal simulation solutions need to be fast, accurate, easy to use, and seamlessly integrate into existing analog and digital verification flows.

Mentor’s Symphony Mixed-Signal Platform powered by Mentor’s Analog FastSPICE circuit simulator delivers the fastest mixed-signal simulation performance in the industry without sacrificing the analog accuracy needed for verification. Symphony is the industry’s most configurable mixed-signal solution that integrates with all the leading digital simulators to allow maximum re-use of verification infrastructure. In addition, Symphony’s advance debugging capabilities improve overall mixed-signal verification productivity.
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

33 Video Courses Available Covering

- DO-254
- UVM Framework
- UVM Debug
- Portable Stimulus Basics
- SystemVerilog OOP
- Formal Verification
- Metrics in SoC Verification
- Verification Planning
- Introductory, Basic, and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- PowerAware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 10,000 questions asked
Verification Patterns Library

www.verificationacademy.com