An Approach to Create Scalable Power Management Verification Environment

Pranav Mopkar, Intel Corporation, Santa Clara, CA - USA (pranav.mopkar@intel.com)

Abstract - Creating a power management verification environment for complex designs with multiple power domains is a daunting task and making it scalable is even more challenging. This paper presents an approach to simplify the verification view of a complex comprising a heterogeneous mix of diverse functionalities and multiple power domains to create a scalable power management verification environment.

Keywords— scalability, scalable, pre-silicon validation, power management validation, multi power domain, multiple power domains.

I. Introduction

Complex designs with multiple power domains (**PD**s) as the Device Under Test (**DUT**) have become more common in design verification due to the increasing importance of minimizing power consumption, driven by the proliferation of handheld battery-powered devices. Architects have been attempting to add the ability to power gate digital logic at a much finer granularity than before to meet aggressive power targets [1] [2][3]. This has resulted in multiple PDs, even at the subsystem level.

A. Motivation

Figures 1 and 2 below are examples of logic restructuring by architects. Consider a design 'DUT 1' that has a memory block and a DMA engine to transfer data in and out of the memory. An IP customer might think that there is complete dependency between these two blocks and decide that they need to be grouped together in a single PD. Another customer might need a design 'DUT 2' to have the memory split into two memories and then decide that the two memories go into their own PDs. In this case, the DMA engine might be required for each of the memories and hence might need to be put into its own independent PD. This is how functional logic can move into different PDs purely based on structural changes or customizations that occur in the DUT.

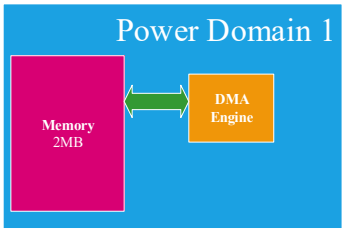

Figure 1: Example 'DUT 1'

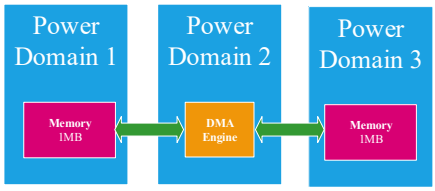

Figure 2: Example 'DUT 2'

Power-management verification engineers often view the DUT as a collection of multiple PDs. Each PD is either assumed to have a fixed set of functionalities contained in it or not considered at all. Verification environments (**VE**s) developed for this view of the DUT reflect this hardcoded association between functionality and PDs. However, architects often decide to move functional design blocks from one PD to another to achieve their power targets. Environmental changes required for these changes can result in an overhaul when such assumptions are made by verification engineers.

B. *Current Prevalent Verification Approach*

Despite diverse approaches, such as HW/SW co-design for complex PM designs, as described in [4], there is a significant reliance on pre-silicon verification to verify PM flows and sequences, as concluded in [4]. The pre-silicon PM VEs that view each PD as a monolithic block of functionality tightly coupled with power management (**PM**) logic create VEs in the image of this view.

C. *Limitations of Existing Methods*

Most PM VEs tend to be constructed with functional logic dependencies, such as functional clock gating, idle indicators, and wake conditions, which are deeply integrated with the PDs and their dependent PM resources, such as trunk clock gating, reset isolations, aggregation of functional wakes, power gating, and wake flows. When the logic needs to be reorganized by moving the functional logic between PDs, new functional logic is added to existing PDs or new PDs are added, and significant effort is required to reconfigure the VE to move the functionality-dependent code out of the code that relates to the PDs. We believe that our framework, which involves reusable collaterals and a simple configuration setup, can significantly reduce the effort required for reconfigurations.

II. METHODOLOGY

D. *Framework Overview*

The basic premise of the proposed solution is to separate the PDs from the functional logic or functional blocks (**FBLK**s). This enables easier VE reconfiguration when FBLKs move between PDs. We also propose simplifying the verification view of PDs and FBLKs to create templates for modeling their PM implications. These two concepts together enable us to create a power-management VE that is scalable with respect to the number of PDs and FBLKs as well as to changes in the PD of any of the FBLKs. We will use SystemVerilog (**SV**) constructs, such as parameters, modules, interfaces, assertions, sequences, and some basic UVM constructs to explain the methodology. This demonstrates that the framework works even with the most basic verification constructs. Advanced methodologies such as Portable Stimulus should just as easily work with our methodology by extension.

E. *What is new in this work?*

This paper suggests treating PDs and FBLKs as separate entities and creating separate verification components for them. This differs from the traditional approach of combining them into a single entity for verification. Traditional approaches, that create monolithic verification components that combine PD and FBLK code, do not allow us to move the functionality freely between the PDs of the DUT.

Custom verification components are usually created for each PD based on their functionality in most traditional PM verification approaches. The traditional approach hampers the scalability of FBLKs because of this customization of verification components. We must view the design blocks as PDs that house FBLKs as a separate component from the FBLKs themselves. We propose creating generic verification templates, one for PDs and one for FBLKs. Templates with their configuration knobs can then be used to create template instances tailored to each PD and FBLK. This allows for greater scalability and flexibility in reconfiguring the verification environment when architectural restructuring is performed. We have used UVM components to create these templates and implemented configuration knobs as UVM configuration knobs for the templates.

F. *Finding Similarities*

PM architecture typically partitions the DUT into multiple PDs. These PD can request the central Power Management Unit (**PMU**) to switch on or gate their power. They can also request the PMU to ungate the clocks. Resets for each of these domains are released by the PMU at an opportune time. This interaction between the PDs and PMU is typically standardized. PD components can be created based on these similarities among the PDs. Figure 3 shows a moderately complex subsystem with multiple PDs, each of which houses one or more FBLKs. Figure 4 shows how the PDs from Figure 3 can be represented as instances of PD components named PD1, PD2, etc.

Every major functional feature of DUT, which is equivalent to an FBLK mentioned earlier, is typically contained within a top-level SV module instance for that feature. These top-level instances interact with the PD and PMU logics through inputs such as functional clocks and resets and through outputs such as idle-indicator and wake-requests. Because of these shared characteristics, it's possible to create a template for FBLKs in form of a UVM component for FBLKs.

Figure 4 shows how the FBLKs DSP, Memory, DMA engine, and Accelerator IPs, etc., in Figure 3 can be represented as different instances of a single FBLK component in the VE. Note that we can also create two-dimensional instances of FBLKs, such as FBLK_DSP[0] and FBLK_DSP[1] that are multiple instances of the same FBLK in the design. This will allow us to scale the verification collaterals for the same type of FBLK (DSP type in this case) when the number of instances of this FBLK increases in the future. We performed this type of two-dimensional structuring for Context Detection and Wake IPs as well as shown in Figure 4.

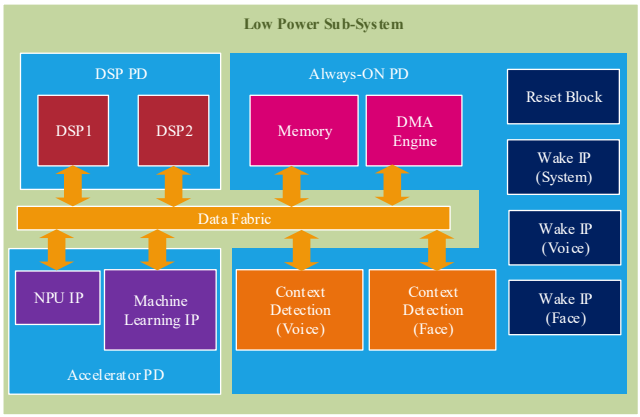

Figure 3: Architectural view

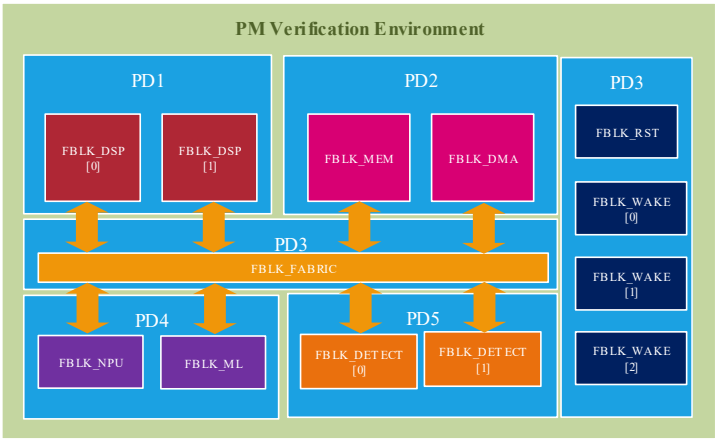

Figure 4: Verification view

G. Managing Differences

PDs can differ from one another in several ways. Some PDs, such as DSPs, may have interrupts as power wake events, whereas others, such as fabrics, may be power gated only in the deepest device power states. When we create power-domain components we should be able to use them for all power-domains. Power-domain component instances have a configuration object associated with them, which has knobs to configure the differences. A similar approach is adopted for the functional-block components. The functional block component has its own configuration object, where all the differences among the FBLKs can be configured.

H. Creating Templates

Power Domain Template

The PD template, implemented as a UVM component, has a configuration object and virtual signal interface. The configuration captures the differences between the PDs and needs to be assigned values based on the configuration of the DUT. The signal interface is hooked up to the corresponding power, clock, and reset signals for a particular PD. It also has checkers and coverage that check the signals. Interfaces are parameterized based on the configuration settings so that the checkers and coverage can be scaled according to the interface. The signal interface can also be used by the

stimulus to make it self-checking in a scalable manner. Figure 5 shows a representation of a PD template and the type of items that can be part of this template.

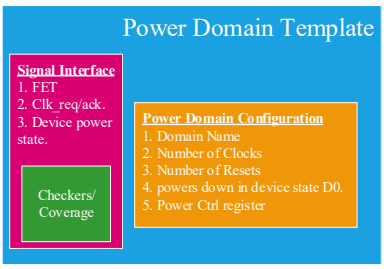

Figure 5: Power Domain Template

FBLK Template

The FBLK template, implemented as a UVM component, has a configuration object and a virtual signal interface as well. The configuration object contains the knobs that differentiate one FBLK from the other. Configuration helps us to deal with the differences between the FBLKs in a scalable manner. It needs to be updated for every new configuration of the DUT. It also has the array index of the PD in which the FBLK is contained. Signal interfaces are connected to the FBLK signals, such as functional clocks, block enables, functional-traffic-idle, and interrupts. It also has checkers and coverage that verify the functional correctness of the connected signals. The interface can be parameterized based on the configuration values. The parameterized interfaces help us to insert or remove checkers/coverage based on the differences between the FBLKs. Figure 6 shows a representation of an FBLK template and the type of items that can be part of this template. In this example, interrupt-related checkers are inserted only if the FBLK is a DSP with an Interrupt input.

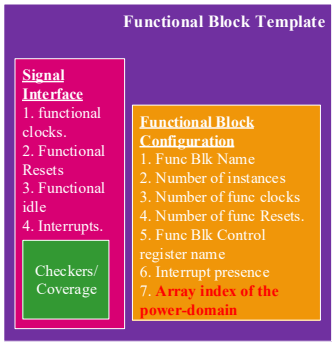

Figure 6: Functional Block Template

I. Environment assembly

The PD and FBLK components were used as building blocks to create the environment. This involves creating instances of PD component for every PD and of FBLK component for every FBLK. FBLKs depend on PDs to reach an operational state. The power of domain must be switched on before any of the resources in FBLK become available for configuration and operation. This requires FBLKs to have information about the PD with which they are associated. This information will be used by the stimulus to bring up the FBLK in and out of operation. This information will also be used by the checkers to check for PD and FBLK dependencies, such as power, clocks, and resets.

The association between the PDs and FBLKs is created by assigning each FBLK the array index of the PD that houses the FBLK in the DUT. As the FBLKs can move around between PDs from one DUT configuration to another, this step must be performed for every configuration change so that the environment structure matches the DUT configuration. When an FBLK is moved from one PD to the other, the array of the PD instance that is passed to this FBLK must be changed to the new PD instance. By maintaining this association based solely on the array index of the power-domain passed to the FBLK instance we significantly simplify the changes required for the environment for such routine changes in the DUT configuration.

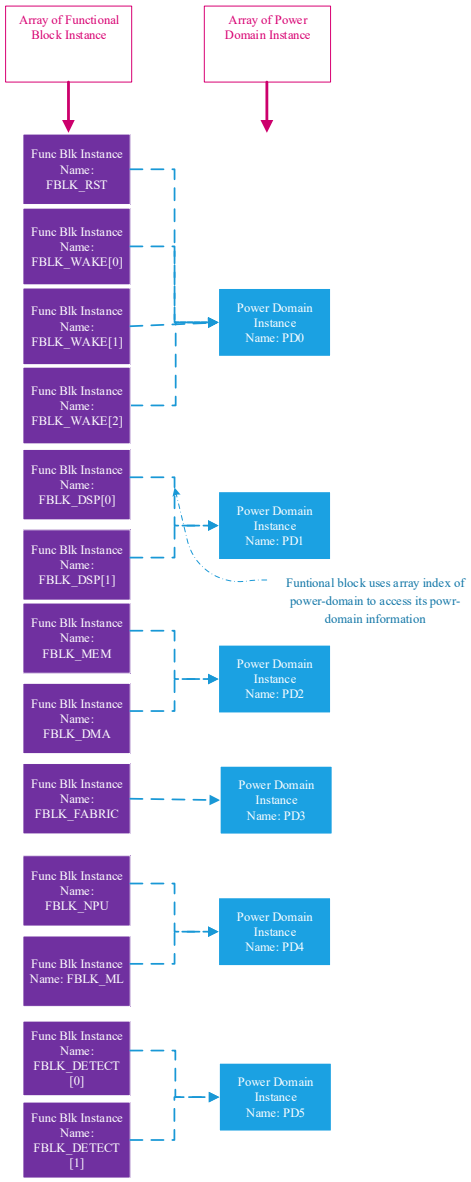

Figure 7: FBLK to Power Domain association

Figure 7 shows how FBLKs and PDs are instantiated in the VE in the form of arrays. This allows us to access these components using the index of these arrays. This, in turn, makes the stimulus scalable, as explained in the stimulus section. Figure 8 shows how the FBLK-PD association can be coded with SV and UVM constructs using the FBLK and PD templates discussed previously.

```
//========================================================================
// UVM Power Management Environment
//========================================================================

class pm_top_env extends uvm_env;

  // Enums for type-safe indexing
  typedef enum {PD_AON=0, PD_DSP=1, PD_MEM=2, PD_FABRIC=3, PD_NPU=4, PD_CONTEXT_DT=5} e_pwr_domain_type;
  typedef enum {DSP_CORE, NPU, ML, MEM, DMA, CONTEXT_DT, RST, WAKE} e_func_blk_type;

  // Enum-derived array sizes
  localparam int NUM_PWR_DOMAINS = e_pwr_domain_type'PD_CONTEXT_DT + 1;
  localparam int NUM_FUNC_BLKS = e_func_blk_type'WAKE + 1;
  localparam int FUNC_BLK_NUM_INST[NUM_FUNC_BLKS] = '{2/*DSP_CORE*/,1/*NPU*/,1/*ML*/,2/*MEM*/,1/*DMA*/,2/*CONTEXT_DT*/,1/*RST*/,3/*WAKE*/};

  // Per-block power domain mappings (sized by instance count)
  localparam int FBLK_TO_PD_CORES[FUNC_BLK_NUM_INST[DSP_CORE]] = '{PD_DSP/*DSP0*/, PD_DSP/*DSP1*/};
  localparam int FBLK_TO_PD_NPU[FUNC_BLK_NUM_INST[NPU]]        = '{PD_NPU/*NPU0*/};
  localparam int FBLK_TO_PD_ML[FUNC_BLK_NUM_INST[ML]]          = '{PD_NPU/*ML0*/};
  localparam int FBLK_TO_PD_MEM[FUNC_BLK_NUM_INST[MEM]]        = '{PD_MEM/*MEM0*/, PD_MEM/*MEM1*/};
  localparam int FBLK_TO_PD_DMA[FUNC_BLK_NUM_INST[DMA]]        = '{PD_MEM/*DMA0*/};
  localparam int FBLK_TO_PD_DT[FUNC_BLK_NUM_INST[CONTEXT_DT]]  = '{PD_CONTEXT_DT/*DT0*/, PD_CONTEXT_DT/*DT1*/};
  localparam int FBLK_TO_PD_RST[FUNC_BLK_NUM_INST[RST]]        = '{PD_AON/*RST0*/};
  localparam int FBLK_TO_PD_WAKE[FUNC_BLK_NUM_INST[WAKE]]      = '{PD_AON/*WAKE0*/,PD_AON/*WAKE1*/,PD_AON/*WAKE1*/};

  pwr_domain_template  pwr_domains[NUM_PWR_DOMAINS];
  func_blk_template    func_blks[NUM_FUNC_BLKS];

  function new(string name = "pm_top_env");
    super.new(name);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Create power domains
    foreach(pwr_domains[i]) begin
      pwr_domains[i] = pwr_domain_template::type_id::create($sformatf("pwr_domains[%0d]", i), this);
      pwr_domains[i].cfg.pd_name = e_pwr_domain_type'(i);
    end

    // Create function blocks
    foreach(func_blks[i]) begin
      func_blks[i] = func_blk_template#(.NUM_INST(FUNC_BLK_NUM_INST[i]))::type_id::create($sformatf("func_blks[%0d]", i),this);
      func_blks[i].num_inst = FUNC_BLK_NUM_INST[i];
      func_blks[i].cfg.fblk_name = e_func_blk_type'(i);
    end

    custom_cfg();
  endfunction

  // Power domain & instance-specific fblk configuration
  function void custom_cfg();
    pwr_domains[PD_AON].cfg.always_on = 1;
    pwr_domains[PD_FABRIC].cfg.deep_pd_only = 1;

    // DSP_CORE wake sources + power mapping
    for(int i=0; i<FUNC_BLK_NUM_INST[DSP_CORE]; i++) begin
      func_blks[DSP_CORE].inst[i].cfg.has_wake_src = 1;
      func_blks[DSP_CORE].inst[i].cfg.power_domain = FBLK_TO_PD_CORES[i];
    end

    // All blocks: enum-driven power domain assignment
    foreach(func_blks[i]) begin
      e_func_blk_type blk_type = func_blks[i].cfg.fblk_name;
      for(int j=0; j<FUNC_BLK_NUM_INST[i]; j++) begin
        case(blk_type)
          DSP_CORE:    func_blks[i].inst[j].cfg.power_domain = FBLK_TO_PD_CORES[j];
          NPU:         func_blks[i].inst[j].cfg.power_domain = FBLK_TO_PD_NPU[j];
          ML:          func_blks[i].inst[j].cfg.power_domain = FBLK_TO_PD_ML[j];
          MEM:         func_blks[i].inst[j].cfg.power_domain = FBLK_TO_PD_MEM[j];
          DMA:         func_blks[i].inst[j].cfg.power_domain = FBLK_TO_PD_DMA[j];
          CONTEXT_DT:  func_blks[i].inst[j].cfg.power_domain = FBLK_TO_PD_DT[j];
          RST:         func_blks[i].inst[j].cfg.power_domain = FBLK_TO_PD_RST[j];
          WAKE:        func_blks[i].inst[j].cfg.power_domain = FBLK_TO_PD_WAKE[j];
        endcase
      end
    end
  endfunction
endclass
```

Figure 8: Environment snippet for the DUT of (figure 4)

J. Stimulus

The stimulus is created using basic building-block sequences that leverage the environment that we have created thus far. Example types of these building-block sequences are as follows:

1. Sequence for changing the power state of any given PD.
2. Sequence for changing the operating state of any given FBLK.
3. Sequence for running functional traffic for any given functional domain.

These sequences are passed an index to the PD or FBLK array. These sequences create stimuli solely based on this index. When we want this stimulus to be executed for another PD or FBLK, we only must change this index that is passed to the sequence. These sequences can then be used to create more complex scenarios to carry out thorough verification of the PM functionality.

Figure 9 shows a flowchart of the building-block sequences of type #2 mentioned above. It also shows how the association between PDs and FBLKs becomes useful in the stimulus. This sequence also uses the interface signals, as depicted by the func_blk[X].if.* signals in the flowchart to create a self-checking stimulus. This sequence is one of the basic sequences for PM stimulus because it can be used to enable/disable any FBLK by merely changing the FBLK enumerated index 'X' that is passed to the sequence. We can create similar sequences for enabling/disabling PDs that take the PD enumerated index as input and create the required stimulus.

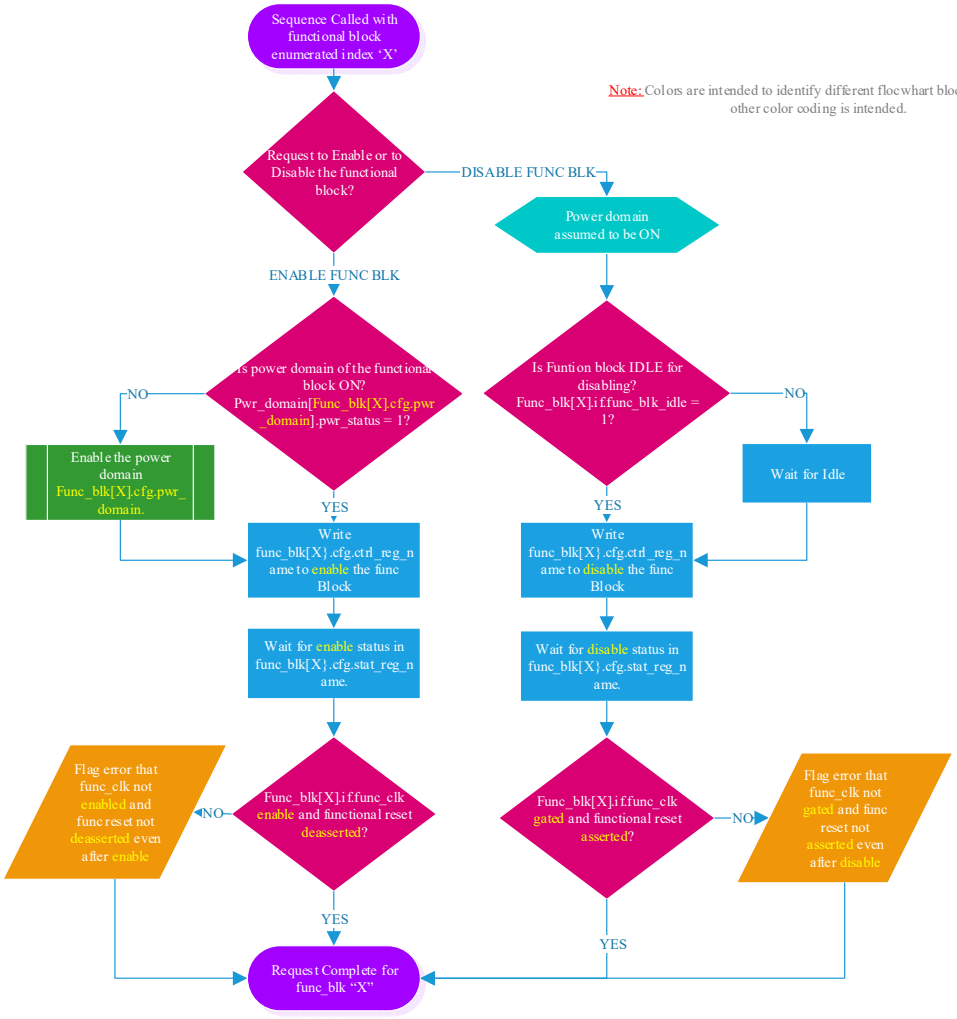

Figure 9: Flowchart for Function Block enable/disable sequence

The creation of complex PM scenarios is greatly simplified because of the scalable nature of these basic sequences. They can be combined with other sequences, such as device state change sequences and reset assertion/deassertion sequences, to execute the entire PM testplan. Power State Table (PST) verification and coverage can be achieved through these complex sequences [5].

K. Checker and Coverage

Checkers and coverage are categorized into three types for the purpose of this methodology:

1. PD checkers/coverage
2. FBLK checkers/coverage
3. Global checkers/coverage

PD checkers and coverage are the types that comprise typical PM items such as sequencing requirements for power-gating, trunk clock-gating, clock-switching, and reset.

FBLK checkers and coverage are related PM items local dynamic clock gating, generation on wake request based on quiescence of the functional logic and reset propagation.

Global checkers and coverage are the checks that are related to interaction between different FBLKs and PDs which cause them to have dependencies. These checks need access to all the PM components to be able to construct the required verification code.

PD and FBLK checkers/coverage were placed inside the respective signal interfaces for PD and FBLK components. Signal interfaces are parameterized so that all possible checkers and coverage can be inserted into them and then enabled based on the parameter passed to the interface at the time of instantiating the interface of the FBLK. Figure 10 shows a basic SV example of how the checkers for interrupts as wake events and memory retention checkers are placed inside the same FBLK interface and are selectively enabled for different FBLK components based on the value of an interface parameter, 'FUNC_BLK_TYPE', that is set by that component. Custom assertions can also be delivered to interfaces within PD components using this approach.

```
interface func_blk_if #(parameter e_func_blks FUNC_BLK_TYPE=NULL_FBLK);

   `include "func_blk_signal_list.svh"

   if(FUNC_BLK_TYPE == DSP_CORE)
   begin
      `include "interrupt_wake_assertions.svh";
   end

   if(FUNC_BLK_TYPE == MEM)
   begin
      `include "memory_retention_assertions.svh";
   end

   .
   .
   .
   .

endinterface
```

Figure 10: Parameterized FBLK interface used for scalability

Global checkers are inserted in the environment top class, where they have access to all PDs and FBLKs. The signals from the individual interfaces along with some additional global PM signal interfaces, can be used to create any complex checker required by the testplan. Various custom checkers and assertions, such as those mentioned in [5][6], can be made available to all PDs through this method in a scalable and reusable manner.

L. Environment Reconfiguration

The environment needs to be reconfigured every time there is a structural change in the DUT that causes a change in the association between PDs and FBLKs. This may involve adding new PD and/or FBLKs. All these changes can be easily incorporated into the environment by changing the parameters of the VE or by changing the association between PDs and FBLKs as described in figures 7 and 8. All these changes can be made in a highly scalable manner with minimum code changes. For structural changes alone, code changes are required for the environment configuration. Any PD and FBLK changes are made to the templates and assertions, which result in the changes being applied to all instances of PDs and FBLKs of the type.

For example, if the architect decides to separate the DSP cores in Figure 3 into separate PDs so that each of them can be powered off individually when they are idle, then we would split PD1 from Figure 4 into PD1_DSP0 and PD1_DSP1 and then change the association in Figure 8 for the two instances of DSP cores to the newly created PDs. The new PD interfaces must be appropriately connected to the signals of the new PD created in the design. If we had code for the two instances of DSP FBLK as part of PD code, the changes needed would be much more involved in separating the collaterals for each DSP instance and then insert this separated DSP code into a new PD code. These changes become exponentially difficult as the number of architectural changes increases.

III. RESULTS

All the components described in the previous section, that is, environment, stimulus, and checkers/coverage, work together to provide a highly scalable environment. This environment can be scaled at the primary level of PDs and FBLKs and at the secondary level of the individual configuration knobs of these templates. When a new configuration of the DUT was required, we edited the environment configuration to edit the PDs and FBLKs. We created an association between them to assemble the environment. The differentiation between various instances was coded into the respective configurations based on the required DUT configuration.

M. Solution Impact

We used this approach to create a PM environment for one of Intel's subsystem IPs. This IP is highly configurable IP and can have up to 15 different PDs with approximately 30 different FBLK instances. This IP is used across multiple SoCs with different power requirements. The configuration of the IP from one SoC generation to the next changed with respect to number of PDs, number of FBLKs, and change in the location of the FBLKs within the PDs. We were able to reconfigure the PM VE for these changes in a couple of days every time there has been such a change. This involved changes mostly to the environment configuration, but almost negligible simulation bring-up effort when the changes were restricted to moving FBLKs among different PDs. Without this framework, we need at least a week and a half worth of effort in the past to make the environment change and redo the simulation bring-up. Even more effort savings were achieved when changes to the PD and FBLK codes were required. These changes were much easier to make and scaled for all instances of the same type because there was clear separation between the two types of environmental components. We reduced the verification effort by two-thirds for every generation of SoCs. We have successfully used this framework for four generations of this IP. Other IPs in the IP organization with similar PM architectures have started moving their codes to this framework based on its proven value.

N. Conclusion

The approach presented in this paper can be applied to any moderately complex design with four or more PDs containing six or more FBLKs. The higher the complexity, the better the scalability returns achieved using this approach.

Although this approach was implemented at the subsystem level, it can easily be scaled at the SoC level. SoCs can reuse the IP configurations to create the SoC configuration and the environment, if the SoC environment is also based on this approach.

This approach presents only the basic framework creation and does not intend to provide a one-stop solution to all PM verification scalability problems. More complex verification features, such as those suggested in [5], can be built using this framework as the basis.

REFERENCES

[1] Ann Mutschler, "Power Domain Implementation Challenges Escalate," Semiconductor Engineering, 2022 (https://semiengineering.com/power-domain-implementation-challenges-escalate/)

[2] Sheetal Kaul(Intel), "Techniques for Optimizing Power, Performance, and Area," EAJournals, 2025 (https://eajournals.org/wp-content/uploads/sites/21/2025/06/Techniques-for-Optimizing-Power.pdf)

[3] M. Kondo, H. Kobyashi, R. Sakamoto, and M. Wada, "Design and Evaluation of Fine-Grained Power-Gating for Embedded Microprocessors," *Proc. Design, Automation & Test in Europe*, pp. 1–6(Entire Paper), 2014. (https://past.date-conference.com/proceedings-archive/2017/pyear/PAPERS/2014/DATE14/PDFFILES/06.4_1.PDF)

[4] Muralidhar R., Krishnakumar N., Morgan B., Karas R., Dennie B., and Rosenberg N. (2016). Pre-Silicon Power Management Verification of Complex SOCs: Experiences with Intel Moorefield. DVCon Proceedings. (https://dvcon-proceedings.org/wp-content/uploads/pre-silicon-power-management-verification-of-complex-socs-experiences-with-intel-moorefield.pdf)

[5] Deepmala Sachan, Thameem Syed S, Raghavendra Prakash, Venugopal Jennarapu (Intel) Low power Verification challenges and coverage recipe to sign-off Power aware Verification", DVCon 2016 Europe. (https://dvcon-proceedings.org/wp-content/uploads/low-power-verification-challenges-and-coverage-recipe-to-sign-off-power-aware-verification.pdf)

[6] H. Matsutani, M. Koibuchi, D. Ikebuchi, K. Usami, H. Nakamura, H. Amano, "Ultra Fine-Grained Run-Time Power Gating of On-Chip Routers for CMPs," in *Proc. IEEE/ACM Int. Symposium on Networks-on-Chip (NOCS)*, pp. 1–8(Entire Paper), 2010. (https://www.arc.ics.keio.ac.jp/~matutani/papers/matsutani_nocs2010.pdf)