Successive Refinement: A Methodology for Incremental Specification of Power Intent

Adnan Khan
ARM Ltd.
110 Fulbourn Road
Cambridge CB1 9NJ
UK
adnan.khan@arm.com

John Biggs
ARM Ltd.
110 Fulbourn Road
Cambridge CB1 9NJ
UK
john.biggs@arm.com

Eamonn Quigley
ARM Ltd.
110 Fulbourn Road
Cambridge CB1 9NJ
UK
eamonn.quigley@arm.com

Erich Marschner
Mentor Graphics Corporation
3919 River Walk
Ellicott City, MD 21042
USA
erich_marschner@mentor.com

Keywords—low power design; low power verification

Abstract—IEEE 1801 UPF [1] enables early specification of "power intent", or the power management architecture of a design, so that power management can be taken into account during design verification and verified along with design functionality. The verified power intent then serves as a golden reference for the implementation flow. To fully realize the advantages of this capability, a methodology called Successive Refinement was conceived during development of IEEE 1801-2009 UPF. However, this methodology is still not well understood in the industry.

In this paper, we present the UPF Successive Refinement methodology in detail. We explain how power management constraints can be specified for IP blocks to ensure correct usage in a power-managed system. We explain how a system's power management architecture can be specified in a technology-independent manner and verified abstractly, before implementation. We also explain how implementation information can be added later. This incremental flow accelerates design and verification of the power management architecture. Partitioning power intent into constraints, configuration, and implementation also simplifies debugging power management issues. We illustrate these advantages by applying the methodology to an IP-based system design.

I. Introduction

Managing power consumption is now one of the key drivers of design and implementation of Systems on Chip, whether it is for extending battery life in mobile devices or constraining power envelopes to manage thermal dissipation. As a result, many more IC design houses and IC IP suppliers are now starting to become much more serious about defining complex power control strategies for their products. This is resulting in widespread adoption of IEEE 1801 UPF as a means for describing both these strategies from high level power intent right down to details of how the power control strategies are implemented.

UPF was originally developed as an Accellera standard [1]. That version of UPF was then updated and was released as IEEE Std 1801^{TM}-2009 UPF [2] and has continued to evolve further as an IEEE standard [3,4]. The initial version of the UPF standard focused more on implementation detail, but IEEE 1801 UPF and its more recent updates have widened the focus to cover more strategic or abstract aspects of power control strategies.

Although IEEE 1801 UPF provides these more abstract capabilities, many users are still employing the older implementation-oriented methodology. In this paper, we describe the new abstract concepts in IEEE 1801 and present a methodology called "Successive Refinement" that takes maximum advantage of these new concepts. Successive Refinement provides a consistent approach for using the entire scope of UPF in order to most efficiently convey the specific power control information required at each different stage of the design flow for a complex system on chip.

Traditional UPF Usage

UPF is used to specify 'power intent' - the power management structures and behavior for a design - separate from and before the design and its power management logic is implemented. This has enabled design flows in which UPF specifications drive both verification and implementation steps. In verification, the design is augmented with structures and behavior from the UPF so that the design behavior reflects power management effects, to ensure that the implementation will work correctly when it is completed. In implementation, the UPF power intent directs the implementation tool to insert the necessary logic to support the power management capabilities.

The UPF used for this kind of flow has tended to be implementation-oriented, since the primary goal is to drive the implementation process and verify that the implementation will work correctly. However, UPF also provides capabilities for more abstract power intent specifications. These capabilities target the development of power intent over time, by different individuals and teams, as design components are aggregated into larger blocks and ultimately into a complete system. The "Successive Refinement" methodology addresses the needs of various participants in this process, from the IP developers who create reusable IP, to the system designers who configure IP blocks and integrate them together with power management logic, to the chip implementers who map the logical design onto a technology-specific physical implementation.

Challenges of using Implementation-Oriented UPF

What exactly are the problems with staying with the less-abstract descriptions of power intent? It is certainly true that system-on-chip designers have managed to make this work and have successfully taped out systems with complex power control strategies. The main issue is the efficiency with which the power intent information is managed across different stages of the design flow and also across different implementations of a particular system.

In the non-abstract approach, the system designer is required to define the power control strategy in terms of implementation-specific details. This means specifying things like power nets, power ports and switches and other implementation-specific details. There are several problems with this approach.

- At early stages in the design flow, it is quite likely that the target process technology has yet to be selected for this system. In this case, the designer may have to make an arbitrary selection of a particular style of implementation in order to describe the high-level power intent. There is a strong possibility that this will prove to be incompatible with the final process technology selection and therefore will need to be re-written when the process technology decision is finally made.
- Having captured this description of the power intent in the UPF file, the system designer will use this file with UPF-aware verification flows in order to verify that the UPF description is itself correct and that the overall power control strategy specified in the UPF file achieves the intended goals. But having invested a lot of verification effort to prove that the strategy and UPF file are correct, if the UPF file has to be modified or re-created after selection of the target process technology, the verification equity built up is lost and the verification process has to be repeated with associated delays and extra resource costs.
- Often large blocks of IP are re-used either in different systems on chip or several different generations of a particular system or even for porting a proven system to a different target technology. Again, if the power intent has been captured in an implementation-specific way, it would potentially need to be re-generated for a new target technology even if the design itself has not changed.
- This is a particular problem for IP suppliers who need to be able to supply descriptions of power intent for products to their customers without having any information about what implementation-specific decisions might be taken by the customer.

Since implementation detail is required for verification, power aware verification is often postponed until late in the flow. This tends to limit both its value and the amount of power aware verification that can be performed within the schedule.

How are these problems solved by more abstract descriptions of power intent?

- Being able to define the power intent at an abstract level without needing to use detail which would be implementation-specific removes any dependence on process technology decisions and implementation decisions which could affect power control.

The power intent can be verified using the same approach as before, and we end up with a verified power strategy and UPF file. But how do we now add the implementation-specific details needed to allow us to implement the design through the back-end flows? We could create a new file but this would again cause us to lose any verification equity built up in the abstract UPF file.

- This is where the "Successive Refinement" concept comes into play. This concept and methodology relies on the idea that we can add further more refined or detailed descriptions of the power control in separate files which are included along with the abstract power intent file such that the tools interpreting these files ensure that any additional detail in subsequent files honours any abstract power intent specified in earlier files.

The next sections go on to describe in more detail how “Successive Refinement” works and some recommendations for how it should be used.

II. What Is Successive Refinement?

Successive refinement in UPF allows an IP provider to capture the low power constraints inherent in an IP block without predicating a particular configuration. Then any licensee of this IP can configure the IP, within these constraints, for their particular application without predicating a particular technology specific implementation. The result is a simulatable but technology-independent "golden reference" against which all subsequent technology-specific implementations can be verified. In this way the "verification equity" invested in proving the integrity of the golden reference is preserved and need not be repeated when the implementation details change.

Constraint UPF:
- *Describe the power intent inherent in the IP - power domains/states/isolation/retention etc.*
- *Constraints are part of the source IP and as such belong with the RTL*

Configuration UPF:
- *Describes application-specific configuration of the UPF Constraints*
    - *supply sets, power states, logic expressions, etc.*
- *Required for simulation - created by end user*

Implementation UPF:
- *Describes technology-specific implementation of the UPF configuration*
    - *supply nets/ports, switches, etc.*
- *Required for implementation - created by end user*

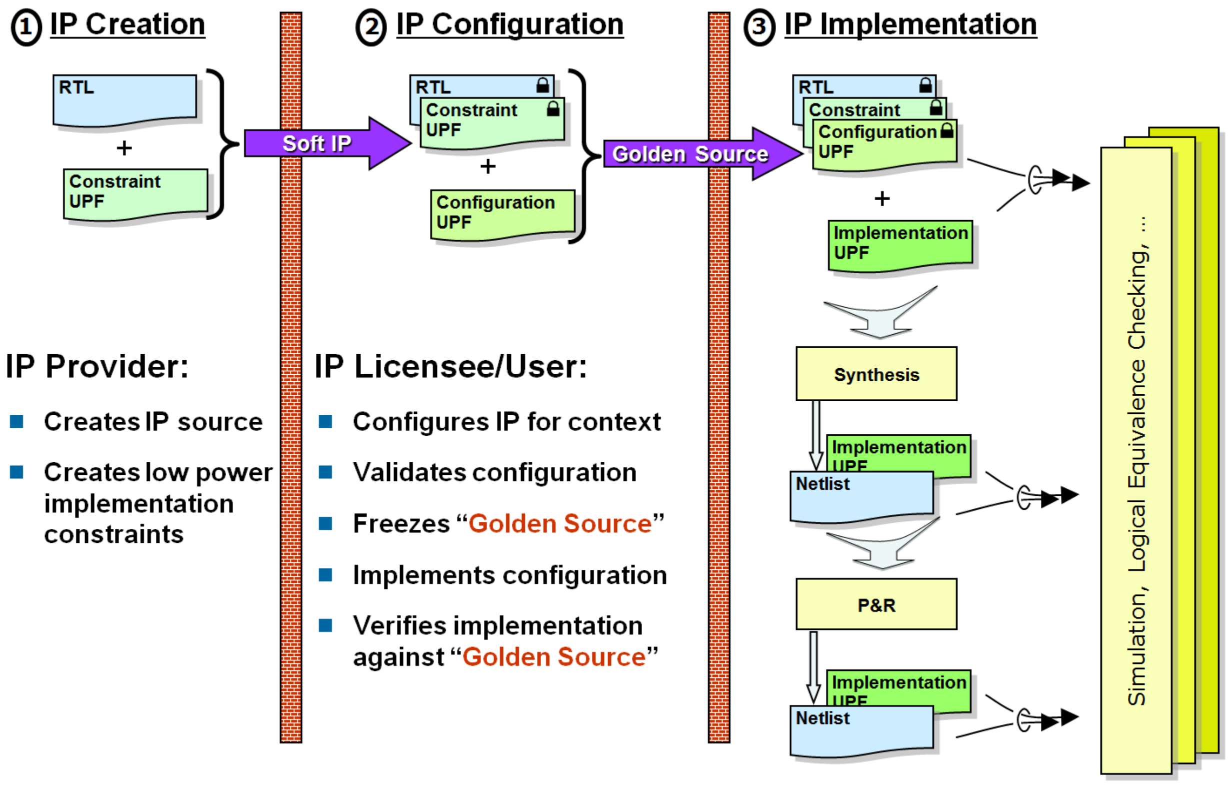

Figure 1. The Successive Refinement Flow

Successive refinement is illustrated in Fig. 1. The IP developer creates Constraint UPF that accompanies the HDL source for a soft IP component. The IP consumer adds Configuration UPF describing his system design including how all the IP blocks in the system are configured and power managed. The Configuration UPF loads in the constraint UPF for each IP so that tools can check that the constraints are met in the configuration. Once the configuration specified by the golden source has been validated, the implementer adds Implementation UPF to specify implementation details and technology mapping. The complete UPF specification then drives the implementation process.

III. Using Successive Refinement

In this paper we show how Successive Refinement can be employed to define the power intent of a processor-based design. The power intent is defined in the following UPF files:

1. Constraint UPF file
2. Configuration UPF file
3. Implementation UPF file

A. Constraint UPF

The constraint UPF file is the most abstract view of power intent. This file is used to describe constraints on the power intent of the design, as it applies to a particular design component. The constraint UPF file is provided by soft IP vendors for use in both verification and implementation of a system using a corresponding IP component. The constraint UPF file should not be replaced or changed by the IP consumer.

The constraint UPF file defines all the power intent that is recommended and verified by the IP provider for the IP it accompanies. At the same time, it does not require the IP consumer to adopt any particular power management implementation approach. All implementation choices can be made by the system designer/integrator.

A constraint UPF file contains the following:

1. A description of the "atomic" power domains
2. The retention requirements
3. The isolation requirements
4. Power states that are legal

A constraint UPF file need not contain information about

1. Any additional design ports that a design may need to control power management logic
2. Any isolation or retention strategies and how they will be controlled
3. Logical references to any particular switch design
4. Technology references such as voltage values or library cell mappings

The above information will typically be provided later by the IP consumer as system design and implementation proceed.

There is also one more important consideration before looking at an example. Almost all IP delivered by an IP provider must be configured before it can be used in a customer-specific implementation. The same configuration is usually applied to the constraint UPF file to match the configured RTL. In that case, it makes sense for configuration of power intent to be a part of the same process that is used to provide a configured RTL.

An Example System

To illustrate the successive refinement methodology, we present an example SoC design together with the constraint UPF, configuration UPF, and implementation UPF that might be developed to define the power intent for this design. The example system, shown in Figure 2, contains an instance of an MPCore processor component called CORTEX®, along with other elements that are not shown. The MPCore processor can have one, two, three, or four CPUs along with L1 and L2 cache. Each CPU has Advanced SIMD (Single Instruction Multiple Data) capability as well.

The MPCore processor component is partitioned such that each individual CPU can be power-managed individually. The functionality of each CPU is divided further into two possible power domains, PDCPU and PDSIMD. The level two (L2) cache and snoop filters are in a separate power domain, PDL2. The rest of the functionality such as the L2 cache controllers and governor is in another power domain, PDCORTEX. Each power domain is shown in a different color in the diagram in Figure 2.

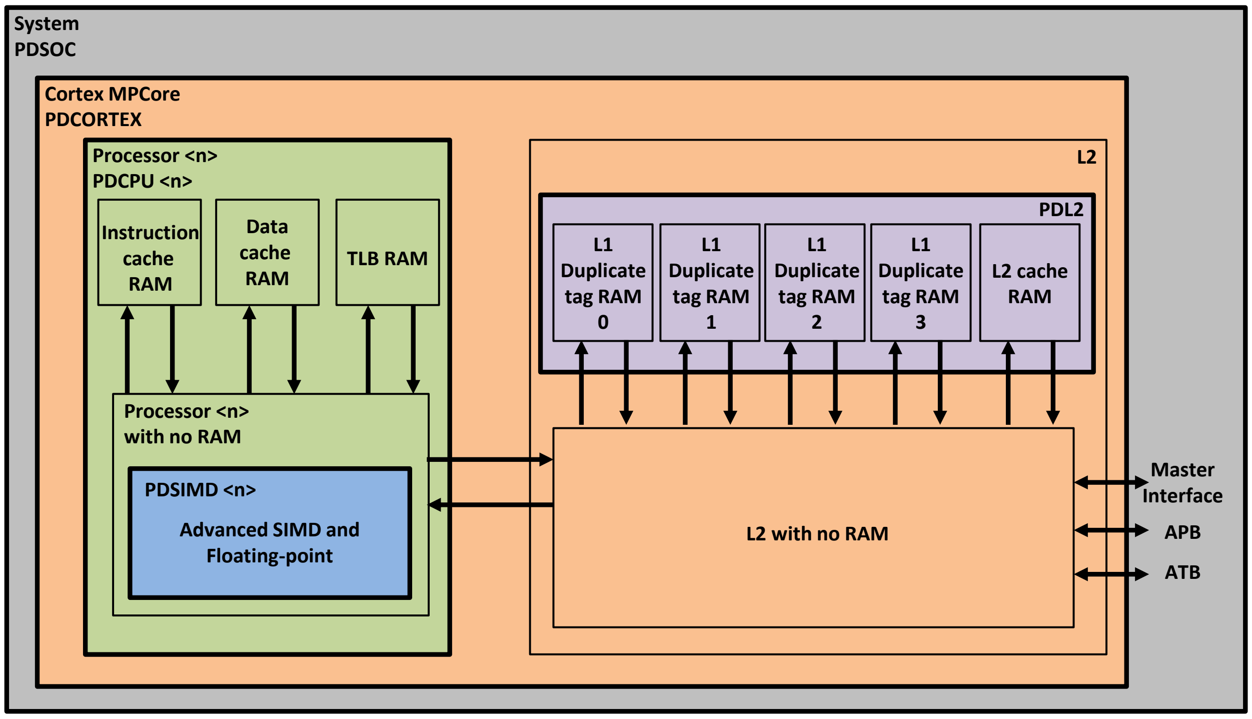

Figure 2. An Example System

In the following, we show how constraint UPF can be written for the MPCore processor component, how configuration UPF can be written for the system as a whole, focusing on power intent related to the MPCore instance, and how implementation UPF can be written for the system, again focusing on aspects related to the MPCore instance. In this illustration, we show the UPF commands[1] only for a single CPU within the MPCore processor component; the UPF for additional CPUs would be similar.

Defining Atomic Power Domains

The constraint UPF file should define each power domain that is identified in the engineering specification for the IP component. For a parameterized IP component such as the MPCore processor, the constraint UPF file for a particular configuration[2] of that IP block would include all of the power domains required for that configuration. For example, the constraint UPF file for a four-CPU version of the MPCore processor would include power domains for all four CPUs.

All that is required to define these power domains is a specification of the components in the RTL design hierarchy that will be included in the power domain. Power domains can be defined as shown below:

[1] The example is presented using UPF 2.1 syntax, which is slightly different from that of UPF 2.0, but the methodology described here can still be used with UPF 2.0.

[2] This refers to configuration of soft IP RTL code by specifying parameter values, not to configuration of system power management by writing configuration UPF.

```
#-------------------------------------------------------------------------
# Create the atomic power domains
#-------------------------------------------------------------------------

# Create the cluster power domain
create_power_domain PDCORTEX –elements {.} -atomic

# Create the CPU0 power domain
create_power_domain PDCPU0 –elements "u_ca_cpu0" -atomic


# Create the SIMD0 power domain
create_power_domain PDSIMD0 –elements "u_ca_advsimd0" -atomic

# Create power domains for CPU1-3 and SIMD1-3 in a similar manner
…

# Create the L2 Cache domain
create_power_domain PDL2 –elements \
  "u_ca_l2/u_ca_l2_datarams \
   u_ca_l2/u_ca_l2_tagrams \
   u_ca_l2/u_cascu_l1d_tagrams" \
  -atomic
```

A domain can contain more than one element. Multiple elements sharing the same power domain can be combined together in a single logical block (for example, a Verilog module). The use of –elements {.} indicates that the instance corresponding to the current scope, and all of its contents, are included in the power domain.[3] The option –atomic indicates that this power domain cannot be further partitioned by the IP consumer.[4]

Retention Constraints

Retention is not actually specified in the constraint UPF file for an IP component. Implementation of retention is an implementation choice and is usually left to IP licensees to decide whether they would like to include it in their design. However, it is necessary to specify which state elements must be retained if the user decides to make use of retention in his power management scheme. The 'set_retention_elements' command specifies these requirements. In the example below, set_retention_elements indicates that all state elements in instance u_ca_cpu0 must be retained (i.e., full retention). Partial retention could also be specified by providing a more detailed –elements list.

```
#-------------------------------------------------------------------------
# Retention permitted in this design
#-------------------------------------------------------------------------

# Define retention requirements
set_retention_elements PDCPU0_RETN –elements "u_ca_cpu0"
```

Isolation Constraints

Like retention, isolation also is not actually specified in the constraint UPF file for an IP component. The need for isolation is driven by system level power management decisions. However, the constraint UPF file should specify the isolation clamp values that must be used if the user decides to shutdown portions of the system as part of his power management scheme.

An isolation cell is used on any signal that crosses a domain boundary from a powered-down domain to one that is powered-up. The signal must be clamped to avoid floating signals and 'x' propagation in simulation. Isolation cells are essentially an AND gate (to clamp to 0) or an OR gate (to clamp to 1) with one input tied to the isolation enable

[3] Use of –elements {.} is a UPF 2.1 feature; the equivalent UPF 2.0 feature is –include_scope.

[4] The –atomic option is also a UPF 2.1 feature. There is no equivalent UPF 2.0 feature. When UPF 2.0 is used, the atomic nature of these power domains can be documented separately.

signal. Clamps are normally tied low, for minimal leakage impact, but in some cases clamps are tied high to avoid functional issues for the powered on parts of the design.

The command ‘set_port_attributes’ is used to define the clamp value requirements:

```
#-------------------------------------------------------------------------
# Isolation semantics - clamp values required during powerdown
#-------------------------------------------------------------------------

set CPU_CLAMP1 [list u_ca_hierarchy/output_port_a ]

# default is isolate low
set_port_attributes -elements "u_ca_hierarchy" \
  -applies_to outputs \
  -clamp_value 0

# CPU_CLAMP1 is a list of exceptions which should be clamped high
set_port_attributes \
  -ports "$CPU_CLAMP1" \
  -clamp_value 1
```

If a default clamp value low rule is assumed, then the constraints should include a list of ports for which a clamp high isolation rule is applied.

The command ‘set_isolation’ should not be specified in the constraint UPF as it would presume that isolation will be required.

Clamp value attributes may also be specified for the ports of a component without reference to any specific instances in the design. In the example below, the clamp value constraint for all output ports of model CORTEX is specified as 0 except for those named in the variable $CPU_CLAMP1, for which the clamp value constraint is specified as 1.

```
#-------------------------------------------------------------------------
# Isolation semantics - clamp values required during powerdown
#-------------------------------------------------------------------------

# default is isolate low
set_port_attributes -model CORTEX \
  -applies_to outputs \
  -clamp_value 0

# CPU_CLAMP1 is a list of exceptions which should be clamped high
set_port_attributes -model CORTEX \
  -ports "$CPU_CLAMP1" \
  -clamp_value 1
```

Power States

Accellera UPF provided commands for defining ‘power state tables’, or PSTs, which defined the legal combinations of supply sources providing power to a chip. Although the PST commands are still present in IEEE1801 UPF, the IEEE standard also provides the command ‘add_power_state’ for defining abstract power states. Such abstract power state definitions are more applicable and valuable when using successive refinement.

The add_power_state command supports power state definition based on both logical and electrical characteristics of an object. A ‘logic expression’ defines the technology-independent logical conditions under which a given power state occurs; a ‘supply expression’ (for supply sets) defines the supply states (and optionally the technology-dependent supply voltages) that exist when the power state occurs.

For constraint UPF, add_power_state should be used to define the fundamental power states of an IP block and its component domains in a technology-independent manner, since technology mapping will not occur until the implementation stage. This implies that power states should be defined without reference to voltage levels. Similarly, constraint UPF should not impose any particular power management approach on the IP consumer, so it should define power states without dictating how power will be controlled.

```
#--------------------------------------------------------------------------
# Power states for PDSIMD of CPU0
#--------------------------------------------------------------------------

add_power_state PDSIMD0 -domain \
  -state {RUN -logic_expr {primary == ON}} \
  -state {SHD -logic_expr {primary == OFF}}

add_power_state PDSIMD0.primary –supply \
  -state {ON  -simstate NORMAL} \
  -state {OFF -simstate CORRUPT}
```

In this example[5], two basic power states have been defined for power domain PDSIMD0: state RUN (running), and state SHD (shutdown). Each is defined with a logic expression that specifies when the domain is in this power state. These domain power states are defined respectively in terms of power states ON and OFF of the domain's primary supply set. The supply set power states are also defined with add_power_state.

In the latter case, a 'simstate' is also defined, which specifies how logic powered with that supply set will simulate when the supply set is in that state. Simstate NORMAL indicates that the logic will operate normally since the power is on. Simstate CORRUPT indicates that the logic will not operate normally, since the power is off; in this case the output of each logic element is corrupted.

Power states defined for supply sets can also include a supply expression that specifies the states and voltages of the supply set functions (power, ground, etc.). This information can be included in the initial add_power_state definition or it can be added later via –update, as shown below. In most cases the voltage information will be technology-specific and deferred to implementation UPF.

```
#--------------------------------------------------------------------------
# Power state updates for PDSIMD0.primary
#--------------------------------------------------------------------------

add_power_state PDSIMD0.primary –supply –update \
  -state {ON  -supply_expr {power == FULL_ON && ground == FULL_ON}} \
  -state {OFF -supply_expr {power == OFF}}
```

Although the use of state retention is up to the IP consumer, as part of the overall power management architecture for the target system, it may be appropriate to define abstract retention power states in the constraint UPF so that power state constraints that would come into play if retention is used can be expressed. In the configuration UPF for the system, for a given instance of this IP, such an abstract retention power state could be used as is, or marked as an illegal state if retention will not be used for that instance, or even refined to create multiple power states representing different levels of state retention.

The following UPF updates the power states of PDSIMD0 to add such an abstract retention state, based on the domain primary supply and its default_retention supply:

```
#--------------------------------------------------------------------------
# Abstract Retention Power State for PDSIMD of CPU0
#--------------------------------------------------------------------------

add_power_state PDSIMD0 -domain –update \
  -state {RET -logic_expr {primary == OFF && default_retention == ON}}
```

Power states would also be defined for the other power domains (PDCORTEX, PDCPU, and PDL2) in the MPCore processor. These would be similar to the definitions given above for PDSIMD0. Here again the constraint UPF file typically defines only the basic power states that will apply to all instances; the set of power states for a given domain may be refined later when configuration decisions are made.

[5] This example includes the –supply and –domain options, which are new in UPF 2.1; they should be removed when using UPF 2.0. Also, in UPF 2.0, the state name is placed immediately before the opening curly brace rather than immediately after it.

```
#----------------------------------------------------------------------------
# Power states for PDCORTEX
#----------------------------------------------------------------------------

add_power_state PDCORTEX -domain \
  -state {RUN -logic_expr {primary == ON  && PDL2 == RUN && PDCPU == RUN}} \
  -state {DMT -logic_expr {primary == OFF && PDL2 == RUN && PDCPU == SHD}} \
  -state {SHD -logic_expr {primary == OFF && PDL2 == SHD && PDCPU == SHD}}
```

Power states of one power domain may be dependent upon power states of another power domain. In this example, the PDCORTEX domain is in the RUN state only if the PDL2 and PDCPU domains are also in their respective RUN states. Similarly, PDCORTEX is in its SHD state only if PDL2 and PDCPU are each also in their respective SHD states. This dependency is captured in the logic expression for the power states of PDCORTEX above, as well as a dormant power state DMT in which the L2 cache is still powered up while the PDCORTEX and PDCPU domains are shut down.

```
#----------------------------------------------------------------------------
# Abstract Retention Power states for PDCORTEX
#----------------------------------------------------------------------------

add_power_state PDCORTEX -domain -update \
  -state {CPU0_RET -logic_expr {primary == ON  && PDL2 != SHD && PDCPU == RET}} \
  -state {L2_RET   -logic_expr {primary == ON  && PDL2 == RET && PDCPU != SHD}}
```

Some power state dependencies may exist in the event that state retention is used. These potential dependencies can be captured based on abstract retention power states in the constraint UPF. In the example system, the PDCORTEX domain may be in either of two retention states, depending upon whether the PDCPU domain or the PDL2 domain is in retention.

B. Configuration UPF

The configuration UPF file defines the power intent needed for the IP consumer's system design. The IP consumer must ensure that his usage of the IP satisfies the constraints specified in the constraint UPF delivered with that IP. These constraints are applied to the system design by loading the constraint UPF file for each instance of the IP to which it applies. The rest of the configuration UPF file specifies the details of the power management scheme for the system. Verification tools can check that these details are consistent with the constraints applied to each IP instance.

A typical configuration UPF file contains the following:

1. Add design ports that a design may use to control power management logic.
   - `create_logic_port`
2. Create isolation strategies on power domains and define how isolation is controlled.
   - `set_isolation`
3. Create retention strategies on power domains and define how retention is controlled.
   - `set_retention`
4. Update power domain states with logic expressions to reflect control inputs.
   - `add_power_state –update`

A configuration UPF file typically does not contain

1. Logical references to any switch designs
2. Technology references such as voltage values, cell references

This information is usually provided later, during the implementation phase.

A configuration UPF file typically does not contain any implementation and technology specific details. The configuration UPF file should be written in an abstract manner such that it can be used for RTL verification without the unnecessary details of implementation, but it should be reused by the backend engineers when they add implementation detail.

Soft IP such as a processor typically can be configured in quite a few different ways. For example, a processor IP block might be configurable anywhere from a single core to a multi-core version. A customer can configure such cores to match their design needs. For the configuration UPF file, one of the logical configurations of the design is chosen, and the customers can use that file as an example to match their design.

Retention Strategies

The configuration UPF file specifies the retention strategies to be used for each power domain. A design can either retain all its flops or can decide to use partial retention in which only certain flops are retained.

The 'set_retention' command is used to specify a retention strategy. In configuration UPF, the retention strategy specifies the control signals used to control retention. These control signals can be defined in the configuration UPF file using the UPF 'create_logic_port' command and then referenced in retention strategies.

```
#-----------------------------------------------------------------------
# Retention Strategy
#-----------------------------------------------------------------------
create_logic_port -direction in nRETNCPU0
create_logic_net nRETNCPU0
connect_logic_net nRETNCPU0 -ports nRETNCPU0

# -------- cpu0 ret ---------
set_retention ret_cpu0 -domain PDCPU \
  -retention_supply_set PDCPU0.default_retention \
  -save_signal "nRETNCPU0 negedge" -restore_signal "nRETNCPU0 posedge"
```

A retention supply is also needed when writing a retention strategy. In the above example a default retention supply set 'default_retention' was used but equivalently a designer can specify their own supply set by using '-supply' option in 'create_power_domain' command.

Isolation Strategies

An isolation strategy is specified for any power domain where the signal crosses a domain boundary. In the constraint UPF file the IP provider specifies the clamp values that are safe to use if the design will have isolation cells between different power domains. In the configuration file, any isolation strategy must specify clamp values consistent with the specifications in the constraint UPF.

The command 'set_isolation' is used to define isolation strategy. As with retention strategies, isolation control signals required for retention can also be defined in the configuration UPF.

```
#-----------------------------------------------------------------------
# Isolation Strategy
#-----------------------------------------------------------------------
# -------- cpu clamp 0 ---------
set_isolation iso_cpu_0 -domain PDCPU0 \
  -isolation_supply_set PDCORTEX.primary \
  -clamp_value 0 \
  -applies_to outputs \
  -isolation_signal nISOLATECPU0 \
  -isolation_sense low

# -------- cpu clamp 1 ---------
set_isolation iso_cpu_1 -domain PDCPU0 \
  -isolation_supply_set PDCORTEX.primary \
  -clamp_value 1 \
  -elements "$CPU_CLAMP1" \
  -isolation_signal nISOLATECPU0 \
  -isolation_sense low
```

In this example, the majority of the signals crossing the power domain boundary are constrained to be clamped low if power is turned off. However, a few signals in the design are constrained to be clamped high because clamping them low in the event of power shutoff would cause a functional issue.

In the above example, this is done by leveraging UPF precedence rules: the first set_isolation command applies to all outputs of the domain PDCPU0 without explicitly naming them, while the second set_isolation command applies to a list of specific port names given in the –elements list via the variable $CPU_CLAMP1. Since UPF gives precedence to a command that explicitly names an object over one that references an object obliquely in some manner, the second command takes precedence over the first for those ports that must be clamped high if power is turned off.

Another approach would be to specify an explicit –elements list in each case, without depending upon the precedence rules. One technique or the other may be more appropriate in a given situation, depending upon how the UPF is written.

Power States

In the configuration UPF file, the power states defined in constraint UPF files are further refined to specify the logical control signals used by the power management controller to enable that state. These logical signals may include control signals for isolation and retention cells.

Logical control signals may also include signals that will eventually control power switches when they are defined as part of the implementation. If the implementation strategy is already known when the design is configured, the system designer may choose to create and refer to switch control signals in order to define power states representing various operational modes. If the implementation strategy is not already known at that time, or if the system designer wants to enable maximum reuse of the configuration UPF, he can define mode control signals that could be used later to control switches in a given implementation. Either way, the power switches themselves are typically not defined in the configuration UPF file, and it is up to the backend engineer to choose the correct formation of power switches depending on the implementation.

```
#---------------------------------------------------------------------------
# Power state controls for PDSIMD of CPU0 and its supply sets
#---------------------------------------------------------------------------

add_power_state PDSIMD0 –domain -update \
  -state {RUN -logic_expr {nPWRUP_SIMD0 == 0 && nPWRUPRetn_SIMD0 == 0}} \
  -state {RET -logic_expr {nPWRUP_SIMD0 == 1 && nPWRUPRetn_SIMD0 == 0 && \
                           nRETN_SIMD0  == 0 && nISOLATE_SIMD0   == 0}} \
  -state {SHD -logic_expr {nPWRUP_SIMD0 == 1 && nPWRUPRetn_SIMD0 == 1}}

add_power_state PDSIMD0.primary –supply -update \
  -state {ON  -supply_expr {power == FULL_ON && ground == FULL_ON} \
              -logic_expr {nPWRUP_SIMD0 == 0}} \
  -state {OFF -supply_expr {power == OFF     || ground == OFF} \
              -logic_expr {nPWRUP_SIMD0 == 1}}

add_power_state PDSIMD0.default_retention –supply \
  -state {ON  -supply_expr {power == FULL_ON && ground == FULL_ON} \
              -logic_expr {nPWRUPRetn_SIMD0 == 0}} \
  -state {OFF -supply_expr {power == OFF     || ground == OFF} \
              -logic_expr {nPWRUPRetn_SIMD0 == 1}}
```

C. Implementation UPF

The implementation UPF file used in successive refinement defines the implementation details and technology-specific information that is needed for the implementation of the design. The constraint UPF, configuration UPF, and implementation UPF files taken together define the entire power intent of the design.

The implementation UPF file contains the information that prescribes the low level details of power switches and voltage rails (supply nets). It defines which supply nets specified by the implementation engineer are connected to the supply sets defined for each power domain. This file also defines the formation of any power switches that have been chosen for this implementation.

An implementation UPF file contains the following information:

1. Creation of supply network elements that a implementation design will need
   - `create_supply_port`
   - `create_supply_net`
   - `create_supply_set`
2. Logical references of any switch design
   - `create_power_switch`
3. Connecting supply nets with the supply sets
   - `connect_supply_net`
   - `associate_supply_set`
4. Technology references such as voltage values, cell references.

It is useful to keep the implementation details separate from the constraint and configuration UPF files so that those power intent files can be used for different implementations that differ in technology details.

Creating the Supply Network

In implementation UPF, the first thing that needs to be done is definition of the supply nets and creation of the supply network. This can be done by using the commands ‘create_supply_port’ and ‘create_supply_net’ as shown below.

```
#------------------------------------------------------------------------
# Supply Network
#------------------------------------------------------------------------
create_supply_port VDD
create_supply_net  VDD -domain PDCORTEX
create_supply_net  VDDCORTEX -domain PDCORTEX
```

Defining Power Switches

The implementation UPF file describes the switch design. A design team does not have to make any decisions prior to implementation UPF about the formation of the switch design it needs for the implementation. This also helps to keep the constraint UPF and configuration UPF files in an abstract form that is used for RTL verification purposes.

The ‘create_power_switch’ command is used to define a power switch.

```
#------------------------------------------------------------------------
# Power switch
#------------------------------------------------------------------------
create_power_switch ps_CORTEX_primary -domain PDCORTEX \
  -input_supply_port { VDD VDD } \
  -output_supply_port { VDDCORTEX VDDCORTEX } \
  -control_port { nPWRUPCORTEX nPWRUPCORTEX } \
  -on_state { on_state VDD {!nPWRUPCORTEX} } \
  -off_state { off_state {nPWRUPCORTEX} }
```

One point to note is that the signals referenced in the control expressions of the power switch definition must correspond in some way to the control signals that were defined in the configuration UPF file and were used in ‘add_power_states’ to control the states of the domains.

Connecting Supply Nets to Supply Sets

Supply sets that were defined for each power domain need to be connected to supply nets provided by the implementation. This can be done by using the –function option of the ‘create_supply_set’ command, as shown below. This option is used here with –update to add the names of the supply nets (VDDCPU, VDDRCPU, VSS) to be connected to functions power, ground of the respective supply sets. In this case, the supply sets themselves were created as part of creating the power domain PDCPU.

```
#---------------------------------------------------------------------------
# Supply Set PDCPU0.primary
#---------------------------------------------------------------------------
create_supply_set PDCPU0.primary -update \
  -function {power VDDCPU0} -function {ground VSS}

create_supply_set PDCPU0.default_retention -update \
  -function {power VDDRCPU0} -function {ground VSS}
```

Defining Supply Voltages for Power States

In the implementation file the voltage values for each supply sets are defined by using the '-supply_expr' option on add_power_state.

```
#---------------------------------------------------------------------------
# Supply Update with supply expressions
#---------------------------------------------------------------------------

#------CPU0-------
add_power_state PDCPU0.primary –supply -update \
  -state {ON  -supply_expr {power == {FULL_ON 0.81} && ground == {FULL_ON 0.00}}} \
  -state {OFF -supply_expr {power == OFF            || ground == OFF}}

add_power_state PDCPU0.default_retention –supply -update \
  -state {ON  -supply_expr {power == {FULL_ON 0.81} && ground == {FULL_ON 0.00}}} \
  -state {OFF -supply_expr {power == OFF            || ground == OFF}}
```

IV. Practical Application of Successive Refinement

To ensure that design IP is used correctly in a power-managed context, it is imperative that the constraint UPF for a given IP is used as provided, without modification by the user. For parameterized IP, which needs to be configured before it is used, the IP provider must be able to deliver constraint UPF that is consistent with any given legal configuration of the IP, so the user does not have to edit the constraint UPF in any way. One approach to this involves delivering software or scripts that take in a set of parameter values and generate both the configured RTL code and the corresponding constraint UPF for that IP configuration.

A given system design may involve multiple instances of a given IP component, and/or instances of multiple IP components. Each IP component will have its own constraint UPF file, the scope of which is local to the IP block. In contrast, the configuration UPF file for a system is written with a global view of the entire system. The configuration file should start with loading the constraint UPF file for each instance, using the load_upf command. Subsequent commands in the configuration UPF file then define how each IP instance is configured for power management in the context of this system.

If a constraint UPF file needs to be loaded for multiple instances of a given IP, the find_objects command can be used to obtain a list of all instances of that IP within the design, and then load_upf can be invoked with the list of instances. Another option would be to use a Tcl loop to apply a load_upf command to each instance name in the list.

If the configuration UPF needs to configure multiple instances of the same IP in the same way, there are similar choices. A Tcl loop could be used to apply the same set of UPF commands for specifying strategies, power state updates, etc. to each instance. It may also be convenient to put configuration commands for a given IP into a subordinate configuration UPF file specifically for that IP, which then can be loaded once for each IP instance.

In the latter case, it may be tempting to make the subordinate configuration UPF file for that IP first load the constraint UPF file for that IP. Such a UPF description could then be loaded once for each instance of the IP to take care of both constraint application and configuration. This may work well in some cases, such as for a hard IP in which the power management configuration is the same in every instance. However, for soft IP, it is usually better to keep separate the loading of constraint UPF files and the specification of power management configurations, since different instances of the same IP component may be configured differently for power management. For example, in a system containing two instances of the same processor, one may be always on while the other can be shut down.

An IP provider may choose to provide an example configuration UPF file along with the constraint UPF file for a given IP. This example configuration UPF file usually describes the power intent of a particular logical configuration example, which licensees can use as a basis for creating a configuration UPF file for their own system design.

When completed, the configuration UPF file together with the constraint UPF files for any IP components and the RTL for the system and its components can be verified in simulation. Once verified, these files can then be considered the 'Golden source' and can be fed into any tool farther along in the project cycle.

V. Challenges of using Successive Refinement

The challenge for any other soft IP vendor is first of all to separate and isolate the constraints themselves. Secondly to provide a description of the constraints that does not have to be altered and is compatible or interoperable with the design tools that will be used first for verification and then for implementation.

In order to achieve a harmonious solution it is necessary for the soft IP vendor to work closely with the EDA tool vendor, and manage the reality that any customer or partner for the IP will most likely require the support of several vendors and multiple tools. It should be clear that a soft IP vendor is well placed to describe and specify requirements in order to achieve the interoperable solutions that their partners require.

VI. Benefits of using Successive Refinement

The true benefit of stating these goals and working towards them is easily realised when we consider the power intent required to build a complex SOC for a mobile application such as a tablet computer or high end smart phone. Such devices will require the fast and efficient integration of IP from multiple sources. The system rapidly becomes complex and difficult to manage. Markets create competitive scenarios where rapid turnaround is required. Short design cycles and early tape outs are possible only when the verification equity of the design components is preserved and relied upon through the implementation stages. UPF constraints ensure that the power intent as constructed is consistent with original intent provided with the soft IP used in the system.

VII. Current State and Future Work

Successive Refinement requires essentially full support for IEEE 1801 UPF, including all of UPF 2.0 and some new features in UPF 2.1. The methodology can be used in part with a subset of UPF features, but the full value can only be realized when all the elements are available in the entire tool chain. At the time of this writing, not all tools support all the constructs required for Successive Refinement, but it is expected that this issue will go away with time.

The IEEE P1801 UPF working group is continuing to refine the definition of UPF in order to support Successive Refinement more effectively. In particular, improvements in power state definition and refinement are being developed for inclusion in the next release of the standard. These improvements should make adoption and use of Successive Refinement even more straightforward than has been described above.

References

[1] Accellera Unified Power Format (UPF) Standard, Version 1.0, February 22, 2007.
[2] IEEE Standard for Design and Verification of Low Power Integrated Circuits, IEEE Std 1801™-2009, 27 March 2009.
[3] IEEE Standard for Design and Verification of Low-Power Integrated Circuits, IEEE Std 1801™-2013, 6 March 2013.
[4] IEEE Standard for Design and Verification of Low-Power Integrated Circuits—Amendment 1, IEEE Std 1801a™-2014, August 2014.