Low Power Specification V2

| Revision # | Author | Date | Notes |
| --- | --- | --- | --- |
| 0.1 | Jim Bauman, Rimu Kaushal | 28 Jan 2015 | Initial Strawman |
| 0.11 | Jim Bauman | 30 Jan 2015 | Updated Power Domain FSM, added WakeReq FSM & section about signal aggregation |
| 0.12 | Jim Bauman | 3 Feb 2015 | Updated Power Domain FSM to tighten up async handshake, misc other updates |
| 0.13 | Jim Bauman | 9 Feb 2015 | Change PMU interface to AMBA AXI/ACE low power interface (sec A-9 AXI spec) |
| 0.14 | Jim Bauman | 10 Feb 2015 | Describe power sequencing steps, discussion of regbus |
| 0.15 | Joji Philip | 17 Feb 2015 | Added micro-arch sections describing master bridge fence & drain |
| 0.16 | Jim Bauman | 26 Feb 2015 | Describe power friendly modifications to regbus, misc other updates |
| 0.17 | Jim Bauman | 27 Feb 2015 | Power Domain FSM updates – remove generation of RstPowerDomain |
| 0.18 | Jim Bauman | 4 Mar 2015 | Added section describing power domain interdependency tracking |
| 0.19 | Jim Bauman | 5 Mar 2015 | Updated Power Domain FSM to add signals for interdependency tracking |
| 0.20 | Jim Bauman | 16 Mar 2015 | Formatting update: add revision tracking, ToC, etc. |
| 0.30 | Jim Bauman | 27 Mar 2015 | Changed from AXI to AMBA Q-Channel Low Power Interface -> updated PMU interface, FSM, etc.  Changed signal polarities and names so that all clamped values are 0. |
| 0.31 | Jim Bauman | 31 Mar 2015 | Updated section 1.2 to describe requirements for multi-voltage an DVFS support |
| 0.32 | Jim Bauman | 7 Apr 2015 | Updated section 1.12 to clarify decisions regarding regbus optimization for low power and add some detail. |
| 0.35 | Jim Bauman | 19 Apr 2015 | Updates  section 1.2.2.2.2.1: updated with refined description of handling voltage boundaries at host interfaces.  section 1.13.1: added to describe details of tune\_power enhancements.  section 1.13.4.1: added to enumerate genrtl HFNet requirements.  section 1.17: added to described handling of compound bridges. |
| 0.36 | Jim Bauman | 23 Apr 2015 | Added *fence\_done\_<PD>\_<master>\_n* to handle powered down masters in aggregation for *fence\_ack\_deasserted\_<PD>\_n*. |
| 0.37 | Jim Bauman | 27 Apr 2015 | Added section 1.8.1 describing reset==idle==clamp==1’b0 rule.  Updated chapter 1.12 to add detail about voltage/power/clock domain boundaries in regbus, plus info about regbus tunnel. |
| 0.38 | Jim Bauman | 4 May 2015 | section 1.5: added rules for signaling robustness.  Added descriptions of power related NocStudio properties and updated related doc sections…  section 1.6.1: overhauled description of *QACTIVE\_<PD>*  section 1.13.8: added to enumerate low power properties.  section 1.14.1: added to cover fencing decision between decerr and autowake.  section 1.14.2: added to describe atomicity of fencing.  section 1.18: started (lacking content) section summarizing optional/phased features. |
| 0.39 | Jim Bauman | 7 May 2015 | Bug 2986: sections 1.4.3 and 1.9.1 update Power Domain FSM to clean up power up sequence by adding REMOVE\_FENCE state to assure clean cessation of fencing before deasserting *QACCEPTn\_<PD>*. |
| 0.40 | Jim Bauman | 12 May 2015 | Sections 1.6/1.6.1/1.6.2, 1.9.1.3-1.9.1.5, and 1.13.8.1: updates to handling of *QDENY\_<PD>* & *QACTIVE\_<PD>*. Updated PD FSM to add REMOVE\_FENCE\_NACK state to cleanly handle *QDENY\_<PD>*. Related, added descriptions of NocStudio PD properties for controlling implementation of *QDENY\_<PD>*, and refined properties controlling *QACTIVE\_<PD>*.  Chapter 1.12: cleaned up to reflect final implementation choices. |
| 0.41 | Jim Bauman | 27 May 2015 | Updates related to cold reset, including updating PMU interface to drive *reset\_n\_<PD>* per power domain. Several other updates.  See the following sections for most of the changes: 1.4.4, 1.6, 1.6.3, 1.7, 1.8, 1.13.8.1, 1.18 and its subsections. |
| 0.42 | Jim Bauman | 9 Jun 2015 | Section 1.7 – added CDC info and requirements.  Section 1.13.8.3 – new require\_autowake\_pd\_for\_routers property.  Section 1.14.2.1 – fencing atomicity execptions.  Section 1.8 – added to lists of phased development items. |
| 0.43 | Jim Bauman | 18 Jun 2015 | Clock Gating - sections 1.10, 1.10.1: updated to describe changes coming with LP V2.  Regbus – sections 1.12.1 and sub-sections: documented limit of 7 clock/power domain combos per node, requirement to use compacted address space.  RTL/CPF and HFN Structure – new section 1.18: describe RTL grouping w/LP and describe how HFNs will be implemented. |
| 0.44 | Jim Bauman | 26 Jun 2015 | Minor update to section 1.10.1 to capture final decision on how to implement cg\_or signals for clock gating. |
| 0.45 | Jim Bauman | 2 Sep 2015 | Multi-voltage Microarchitecture: moved content from 1.2 to newly inserted section 1.18 and added to it detailed description of host interface VD crossing.  clk\_nsps: officially defined this.  Power sequence waveforms: updated to latest, consistent with 15.08 LP User Manual. |
| 0.46 | Jim Bauman | 21 Oct 2015 | Section 1.9.1.1 Handling of reset\_pd\_<PD>\_n: added paragraphs documenting that reset must only be asserted during Q\_STOP, but it may be deasserted during Q\_STOP or Q\_EXIT.  Misc other cosmetic fixes. |
| 0.47 | Jim Bauman | 22 Oct 2015 | New Section: 1.10 – Fundamental Architecture of Fencing and Draining.  1.13.2 – minor updates for regbus master fencing and draining.  1.13.5 – major update for regbus tunnel  New Section: 1.19 – Other Bridges and Agents: specifically added basic requirements for IMGBus4 and IMGBus2. |
| 0.48 | Jim Bauman | 26 Oct 2015 | Filled out 1.19.4 and 1.19.5 about Metabridge and Reorder Bridge respectively. |
| 0.49 | Jim Bauman | 27 Oct 2015 | Section 1.18.1: overhauled spec for AHB-lite master bridge. |
| 0.50 | Jim Bauman | 3 Nov 2015 | Section 1.9: updated PD FSM diagram, added round\_trip\_timer to allow fencing handshake to be aborted to enable QDENY to resolve autowake deadlock.  Section 1.6.3: reset\_<PD>\_n and reset\_pd\_<PD>\_n – moved some content here that was elsewhere (nothing new). |
| 0.51 | Jim Bauman | 24 Nov 2015 | Section 1.7: updated to document drain\_ack\_n.  Section 1.9: updated PD FSM diagram and subsections to document addition of drain\_ack\_n and handling of sleeping/waking masters.  Section 1.8.2: added to discuss interrupts.  Section 1.13.1: updated to reflect removal of requirement that ring master live in same power domain as regbus router.  Section 1.13.4: Explicitly describe voltage/power/clock domain relationships among regbus tunnel elements. |
| 0.52 | Jim Bauman | 23 Aug 2016 | Section 1.13.5: (added) user regbus  Section 1.18.1: Minor updates for ahblm  Section 1.19.5: Minor updates for reorder bridge |

Table of Contents

[1 Low Power Hardware Requirements (high level) 11](#_Toc436132837)

[1.1 Customer Requirements 11](#_Toc436132838)

[1.2 Power Domains and Relationships to Clock and Voltage Domains 12](#_Toc436132839)

[1.2.1 Clock Domains 12](#_Toc436132840)

[1.2.2 Multiple Voltage Domains 13](#_Toc436132841)

[1.2.3 External (customer specified) Power Domains 14](#_Toc436132842)

[1.2.4 Internal (NS Derived) Power Domains 14](#_Toc436132843)

[1.2.5 Resolving Power Domain Interdependency 15](#_Toc436132844)

[1.3 NetSpeed Power Management Architecture 15](#_Toc436132845)

[1.3.1 PMU (aka Customer Power Management Logic): 16](#_Toc436132846)

[1.3.2 NetSpeed Power Supervisor: 16](#_Toc436132847)

[1.3.3 NoC Element Power Management Logic: 16](#_Toc436132848)

[1.4 Basic Power Sequencing Procedure 17](#_Toc436132849)

[1.4.1 PMU Interface 17](#_Toc436132850)

[1.4.2 Power Down Sequence Details 17](#_Toc436132851)

[1.4.3 Power Up Sequence Details 19](#_Toc436132852)

[1.4.4 Power on Reset 20](#_Toc436132853)

[1.5 Signaling 21](#_Toc436132854)

[1.6 NetSpeed Power Supervisor Interface to PMU 21](#_Toc436132855)

[1.6.1 QDENY\_<PD> 23](#_Toc436132856)

[1.6.2 QACTIVE\_<PD> 23](#_Toc436132857)

[1.6.3 reset\_<PD>\_n and reset\_pd\_<PD>\_n 24](#_Toc436132858)

[1.7 NetSpeed Power Supervisor Interface to NoC Elements 25](#_Toc436132859)

[1.7.1 Signal Aggregation 28](#_Toc436132860)

[1.8 NoC Elements 30](#_Toc436132861)

[1.8.1 Reset == Idle == Clamp == 1’b0 31](#_Toc436132862)

[1.8.2 Interrupts 31](#_Toc436132863)

[1.9 Power Supervisor State Machine 32](#_Toc436132864)

[1.9.1 Power Domain FSM 32](#_Toc436132865)

[1.9.2 Power Domain Interdependencies – TBD Interlock 38](#_Toc436132866)

[1.10 Fundamental Architecture of Fencing and Draining 38](#_Toc436132867)

[1.10.1 Two Phases: Fencing/Draining and Sleep\_req/Sleep\_ack 38](#_Toc436132868)

[1.10.2 Fencing and Draining Across Series of NoC Elements that Maintain Transaction State 39](#_Toc436132869)

[1.11 Clock Gating 40](#_Toc436132870)

[1.11.1 Changes to Coarse Grained Clock Gating Implementation 40](#_Toc436132871)

[1.12 Preserving/Restoring State 40](#_Toc436132872)

[1.12.1 Retention Registers and Reset 41](#_Toc436132873)

[1.13 Regbus 41](#_Toc436132874)

[1.13.1 Low Power Friendly Regbus Optimization 41](#_Toc436132875)

[1.13.2 Regbus Master Fencing and Draining 43](#_Toc436132876)

[1.13.3 Ring Master and Regbus Routers 44](#_Toc436132877)

[1.13.4 Regbus Tunnel 44](#_Toc436132878)

[1.14 NocStudio Requirements 45](#_Toc436132879)

[1.14.1 Tune\_power Enhancements 45](#_Toc436132880)

[1.14.2 Generate and Communicate Power Domain Information to Master Bridges 47](#_Toc436132881)

[1.14.3 Generation of NetSpeed Power Supervisor Module 47](#_Toc436132882)

[1.14.4 Wiring between NSPS and NoC Elements 47](#_Toc436132883)

[1.14.5 Regbus Optimization 49](#_Toc436132884)

[1.14.6 Power Aware Routing 49](#_Toc436132885)

[1.14.7 Power Domain Interdependency Tracking 49](#_Toc436132886)

[1.14.8 Low Power Related Properties 51](#_Toc436132887)

[1.15 Master Bridge request Fencing 52](#_Toc436132888)

[1.15.1 Controlling Fence Response: Decerr vs. Autowake 53](#_Toc436132889)

[1.15.2 Fencing Atomicity 54](#_Toc436132890)

[1.16 Master Bridge Response draining 55](#_Toc436132891)

[1.17 Streaming Bridge 56](#_Toc436132892)

[1.17.1 Streaming Bridge Fencing 56](#_Toc436132893)

[1.17.2 Streaming Bridge Draining 56](#_Toc436132894)

[1.18 Compound Bridges 56](#_Toc436132895)

[1.18.1 AHB-lite Master Bridge 57](#_Toc436132896)

[1.18.2 Compound Slave Bridges 59](#_Toc436132897)

[1.19 Other Bridges and Agents 60](#_Toc436132898)

[1.19.1 IMGBus4 Master 60](#_Toc436132899)

[1.19.2 IMGBus2 Slave 60](#_Toc436132900)

[1.19.3 Regbus Tunnel 61](#_Toc436132901)

[1.19.4 Metabridge 61](#_Toc436132902)

[1.19.5 Reorder Bridge 62](#_Toc436132903)

[1.20 Multi-Voltage Microarchitecture 62](#_Toc436132904)

[1.20.1 Voltage Domain Crossing FIFO 62](#_Toc436132905)

[1.20.2 Voltage Boundary Locations 63](#_Toc436132906)

[1.20.3 DVFS – Dynamic Voltage Frequency Scaling 65](#_Toc436132907)

[1.21 RTL & CPF Structure and Related Back-End Considerations 67](#_Toc436132908)

[1.21.1 RTL Grouping by Power Domain 67](#_Toc436132909)

[1.21.2 Pipeline Registers 67](#_Toc436132910)

[1.21.3 High Fan-out Nets 67](#_Toc436132911)

[1.22 Phased Development & Optional Features 70](#_Toc436132912)

[1.23 References 70](#_Toc436132913)

Table of Figures

[Figure 1 NoC with Power Management 11](#_Toc436132914)

[Figure 2 NetSpeed Power Management Architecture 16](#_Toc436132915)

[Figure 3 Power Down Sequence Waveforms 19](#_Toc436132916)

[Figure 4 Power Up Sequence Waveforms 20](#_Toc436132917)

[Figure 5 Power Domain FSM 33](#_Toc436132918)

[Figure 6 Waking Master Bridge Fencing and Draining Hazard Waveforms 34](#_Toc436132919)

[Figure 7 Master Bridge low power interfaces 51](#_Toc436132920)

[Figure 8 Master bridge request fencing 52](#_Toc436132921)

[Figure 9 Master bridge response draining 54](#_Toc436132922)

[Figure 10 VD Crossing FIFO Implementation for AXI Master Bridge 63](#_Toc436132923)

[Figure 11 HFN Structure When PD FSMs Live in System Power Domain 67](#_Toc436132924)

[Figure 12 HFNs When PD FSMs Mapped to Native Power Domains 68](#_Toc436132925)

# Low Power Hardware Requirements (high level)

**Power Management Unit (SoC)**

**Master**

**Power Domain A**

**Power Domain B**

**Power Domain C**

**Power Domain D**

**PDS-A**

**External Master/Slave Interface**

**Master**

**Slave**

**PDS-B**

**Slave**

**Master**

**Master**

**PDS-C**

**PDS-D**

**Slave**

**Slave**

**R**

**R**

**R**

**R**

**NoC Power Supervisor (AO)**

**Power Domain power Supervisor to NoC Element I/F**

**NPS to PDS I/F**

**NetSpeed Power Supervisor to PMU I/F**

**PDS-AO**

Figure 1 NoC with Power Management

## Customer Requirements

The following assumptions about customer desires and requirements drive choices made in this specification.

1. Simple interface driven by a handful of signals: we have feedback that our initial low power architecture was too complex. Customers were not happy with requirement to sequence regbus transactions in order to safely change power state of NoC elements.
   1. Stable interface well-defined early in project: customers will generally want to lock down requirements of power control logic early in the project, perhaps prior to freezing NoC implementation. It is desirable to abstract away dependencies upon NoC micro architecture.
2. Industry standard to the extent possible: our customers will find it easier to accept our implementation if it looks similar to what they have worked with in the past.
   1. One customer explicitly asked us to support ARM AMBA low-power interface. We have modeled our interface to be aligned with this specification.
   2. The industry has broadly adopted the concept of a disconnect protocol that implements the concepts of fencing and draining, where in response to power-down requests, network initiators hold off injection of new traffic targeting the domain to be powered down (fencing) and withhold acceptance of power-down request until any pending transactions have completed (draining). We implement this.
   3. Wake-on-Demand: related to fencing, the industry offers optional support for signaling the power control circuitry when new transactions arrive at a fenced initiator interface to request power be restored to the target. Alternatively, the initiator can reject the new requests, e.g., by signaling a decode error. We support either option.
3. Power Domains are Physically Expensive: in many implementations, though probably not all, there is a fair amount of area overhead and physical design complexity associated with each power domain. Therefore:
   1. In many SoC scenarios, the number of power domains will be limited. *It would be interesting to come up with a metric that says something like “80% of designs are expected to have fewer than X customer defined power domains.”*
      * As of 3/26/15, feedback from customers suggests this may not be very broadly true. Mobile platforms may have dozens of power domains.
   2. Customers will be resistant to defining new power domains specifically for use within the NoC, particularly if these new domains would have to span a large and/or sparse area.
      * As of 3/26/15, customer feedback also suggests this is not likely generally true. Some customers appear to favor dedicating power domains to the NoC.

## Power Domains and Relationships to Clock and Voltage Domains

Power domains are the logical construct through which power control is communicated between SoC power control (aka PMU) and the NoC. They nominally describe the boundaries of regions that share common status regarding clock gating (above NoC element level) and power gating. The sections below describe the interactions between power domains, clock domains and voltage domains.

### Clock Domains

Clock domains may span power domain boundaries (and possibly even voltage domain boundaries, though that is likely more rare). However, where clock gating is intended at any level above an individual NoC element (e.g., routers and bridges), a power domain must be defined that matches the boundary of the gated clock domain. A distinct clock input pin must be provided at the power domain boundary for each clock contained within the power domain. It is recommended that these distinct clock pins be brought to the NoC top-level interface where they can be connected together at SoC level where clock domains span power domain boundaries.

The SoC PMU must request power state change from the NoC as described below before gating clocks at this level.

In cases where course grained clock gating is intended above the NoC element level but at finer granularity than would be power gated (i.e., a power gating domain spans multiple clock gating domains), power domains must be defined for each gated clock domain, and when power gating the aggregate domain, the SoC PMU must request power state change for all the gated clock domains that are part of the aggregate power gating domain. The converse also applies.

### Multiple Voltage Domains

Somewhat distinct from low power functionality but related, we must support multiple voltage domains within the NoC. This means we must implement structures that allow for safe passage of signals between voltage domains.

Clock domains may span voltage domains, though this is not expected to be very common in practice, and distinct clock pins brought to the NoC top-level interface should be provided for each voltage domain.

Power domains may not span voltage domains, but a voltage domain may contain multiple power domains.

Some basic requirements for multi-voltage domain support are:

1. NocStudio must provide a mechanism for defining voltage domains and describing their boundaries.
   1. We need to consider that physical constraints may drive placement of these boundaries between grid points.
2. RTL must be structured to be partitioned at voltage domain boundaries to facilitate physical design.
3. CPF/UPF must describe the voltage domains and their boundaries, including specifying the insertion of level shifters on the interface signals that cross them.
   1. Where a power domain boundary coincides with a voltage domain boundary, as will often be the case, it would be desirable to specify integrated level shifter isolation cells if available in the target technology.

A key challenge is identifying optimal locations within the NoC to implement voltage boundaries by inserting level shifters.

#### Special Case: Synchronous Voltage Domain Boundary

While this is not generally expected, it is possible that a design might be implemented such that synchronous clocks are provided on each side of a voltage boundary. In this case, all that needs to be done is to specify in CPF/UPF that level shifters must be inserted in all the interfaces that cross the boundary.

*At present, there are no plans to support this scenario.*

#### General Case: Asynchronous Voltage Domain Boundary

In general, voltage domain boundaries imply clock domain boundaries due to the challenge of balancing distribution of a synchronous clock across the voltage change. So we need to implement logic that safely passes signals across clock domain boundaries and is partitioned at the voltage boundary to support level shifter insertion and separate synthesis and physical design of each side of the voltage boundary.

### External (customer specified) Power Domains

NocStudio allows customers to specify power domains via the *add\_power\_domain* command, and elements are assigned membership to these power domain via power\_domain properties, either explicitly with prop\_set commands or automatically via *tune\_power.* One or more power profiles are defined that specify expected/allowed power states in the SoC: a description of which power domains may be turned off at together. These power profiles are used by *tune\_power* to optimize power domain assignment (as well as to drive power simulation, etc.).

The power domains specified with *add\_power\_domain* are “external” power domains, explicitly specified by the customer, general assumed to encompass host logic in addition to some NoC elements. We expect that some external power management logic exists that drives the decisions around when to power down each of these domains, and this logic is responsible for driving controls for power domain circuitry such as power gates and isolation clamps.

Choices for mapping NoC elements into these external power domains are driven by a least a couple considerations:

1. Is the element in question required to sustain communication for any flows terminating outside the target external power domain? I.e., are there any “through paths” passing through the element.
2. Physical proximity: is the element local to the physical boundaries of the target power domain?

For bridges, most of the time the answers will support aggregation with the host power domain, exceptions occurring perhaps if host logic communicating with different interfaces (e.g., streaming bridge, apb bridge) lives in different power domains (*do we support this? Should we?*). Caveat: physical design considerations may factor in where host clock domain is different than NoC clock domain for the bridge. *Do we support separate power domains for host vs. NoC side logic in our bridges?*

For routers, in the general case, these considerations will often prevent mapping them into external power domains. For example, it will generally be undesirable for traffic between to powered up domains to be interrupted because a router carrying traffic between these domains is mapped to a 3rd unrelated domain that is powered down.

### Internal (NS Derived) Power Domains

NocStudio has the ability to create and map routers into internal power domains that can be controlled so that they may be kept powered when some subset of through paths may still be active (i.e., some power domains with dependent flows remain active). NocStudio uses its knowledge of traffic profiles and power profiles to determine which elements have common sets of dependencies and can be grouped into the same internal power domain. However, customers may be resistant to creating many/any new power domains, particularly if the number of them and/or the constraints on how they are used vary with iterations of the NoC design.

### Resolving Power Domain Interdependency

How do we handle the cases where we can’t cleanly map routers (or other elements) into external power domains? Several alternatives must be considered:

1. Map routers into new “internal” power domains.
2. Map routers into a default NoC power domain: this could be one of the existing external power domains, or it could be a degenerate case of #1 where a single new internal power domain is created. This power domain would be constrained to only be powered down when all other external power domains are powered down.
3. Force routers into external power domains despite interdependencies.
   1. Also could allow mapping to internal power domains that are a better but not perfect fit.

All of these alternatives have merits, at least for certain deployment models. At present, while we might predict that one alternative would be more likely than others to be adopted, our approach is to architect a solution that does not preclude any of these alternatives.

Some implications:

1. Fencing and Draining in Masters: needs to be implemented such that for a given slave id, multiple power domain dependencies may be tracked.
2. Signaling to customer logic: we need to explore generating some signals to that provide feedback to customer logic which power domains are safe to power down based on the power state of other domains. E.g., vector per power domain of which domains have traffic flows that depend on it being powered, or a bit per power domain that indicates it is safe to power down given state of all other power domains. See “Section 1.14.7 - Power Domain Interdependency Tracking” for details of currently proposed mechanisms*.*

## NetSpeed Power Management Architecture

The power management architecture is comprised of 3 layers.



Figure 2 NetSpeed Power Management Architecture

### PMU (aka Customer Power Management Logic):

SoC power management logic owned entirely by the customer, responsible for aspects of power management for the device including power gating and clock gating. Generates all power gating controls (isolation, power gate enable, etc.) – NetSpeed logic does not generate these. Generally expected exist in an “always on” power domain.

The PMU coordinates power state changes with the NoC via an AMBA Q-channel interface as defined in the ARM AMBA Low Power Interface Specification.

### NetSpeed Power Supervisor:

NetSpeed power management logic (aka NSPS) that abstracts away NoC microarchitectural details to present a simple well defined interface to PMU for communication of power intent.

For each power domain in the NoC, it maintains a Power Domain State Machine that provides high-level sequencing of the operations required for power removal and power restoration. This FSM drives the Q-channel interface (QREQn/QACCEPTn/QDENY/QACTIVE) to the PMU in conjunction with driving signals to elements in the NoC needed to coordinate power sequencing activity. It may be mapped into the same power domain as the PMU (always on – at a minimum with respect to all NoC power domains), possibly co-located with PMU, but it is possible to map the FSM into the power domain it is managing.

The NS Power Supervisor also has aggregation logic to combine acknowledgment signals and wake request signals returned from NoC elements. This logic is combinatorial and generally distributed in the design to minimize wiring impact, co-located where possible in the power domain of the signals being aggregated, though some gates must live in the PMU domain (particularly those driving *QACTIVE*).

### NoC Element Power Management Logic:

Logic in each of the NoC elements that implements required power management functionality, including supporting coordination with NS Power Supervisor (e.g., fencing and draining, idle status and sleep ack, etc.). Located within each NoC element (bridges & routers).

## Basic Power Sequencing Procedure

The following lists enumerate the basic sequences of operations that are executed for power down and power up events (note: not every arc in FSM is described – see FSM diagram in 1.9.1 for full details). The architecture does not explicitly enforce restrictions on the order and combination of power transitions between power domains. The PMU may request power transitions in any order and combination – it is not bound to adhere to a discrete set of power profiles. However, the NS Power Supervisor may NACK power down requests when affected NoC elements fail to acknowledge they have all achieved safe power removal status within a timeout window, likely due to ongoing activity.

### PMU Interface

Power transitions are controlled by customer logic in the PMU. The PMU issues requests to power down logic, which must be acknowledged by the NSPS, which either accepts or rejects the request. The NSPS may issue requests to wake power domains. The PMU is responsible for waking domains, either in response to NSPS request or other criteria.

We implement the Q-channel low-power signaling protocol according to the ARM AMBA Low Power Interface Specification. Four signals are involved: *QREQn\_<PD>* (driven by PMU), *QACCEPTn\_<PD>*, *QDENY\_<PD>* and *QACTIVE\_<PD>* (latter 3 driven by NSPS), their use briefly summarized here:

#### Normal Operation: QREQn\_<PD>, QACCEPTn\_<PD> are == 1, and QDENY\_<PD> is == 0. QACTIVE\_<PD> may be in either state.

#### Power Down Request

1. PMU drives *QREQn\_<PD>* low
2. NSPS decides whether it can accept power down request or not:
   1. Will accept: drives *QACCEPTn\_<PD>* low
   2. Will not accept: drives *QDENY\_<PD>* high

Note: PMU must hold *QREQn\_<PD>* low until NSPS responds by asserting either *QACCEPTn\_<PD>* or *QDENY\_<PD>*, and it must return to normal operation by raising *QREQn\_<PD>* high and waiting for *QACCEPTn\_<PD>* to go high or *QDENY\_<PD>* to go low before initiating a new power down request (for this domain).

#### Powered Down: QREQn\_<PD>, QACCEPTn\_<PD>, QDENY\_<PD> and QACTIVE\_<PD> all == 0.

#### Wake Request initiated by NSPS: drives QACTIVE\_<PD> high.

#### Power Up

1. PMU drives *QREQn\_<PD>* high.
2. NSPS acks by driving *QACCEPTn\_<PD>* high when logic is safely restored for normal operation.

### Power Down Sequence Details

Starting in the ACTIVE state: *QREQn\_<PD>* and *QACCEPTn\_<PD>* are high, as is *pd\_active\_<PD>* going to NoC elements, and *QDENY\_<PD>* is low.

1. (State = ACTIVE) PMU requests power down by driving *QREQn\_<PD>* low
   1. If NoC elements are not ready to accept the new power down request (some *fence\_ack\_n* and/or *sleep\_ack\_n* signals from NoC elements remain asserted), or we fail TBD interdependency check, the request is NACKed by driving *QDENY\_<PD>* high (advance to SLEEP\_NACK state).
   2. Otherwise, NS Power Supervisor initiates the power down procedure by driving *pd\_active\_<PD>* low and advancing to the FENCE\_DRAIN state.
2. (State = FENCE\_DRAIN) NoC masters observe 1->0 transition on *pd\_active\_<PD>* and initiate fencing and draining. Masters signal completion back to NSPS by asserting *fence\_ack\_<PD>\_<master>\_n*. When all masters have done this, advance to SLEEP\_WAIT state.
3. (State = SLEEP\_WAIT) NSPS waits here for a Sleep Delay Counter to expire before moving on to the SLEEP\_REQ state. The purpose is to allow additional time for traffic to drain from the NoC when protocols don’t allow for all master bridges to definitively determine when all outstanding transactions have completed (e.g., NSIP streaming protocol). This state may be bypassed directly to SLEEP\_REQ when such conditions do not exist.
4. (State = SLEEP\_REQ) NSPS asserts *sleep\_req\_<PD>\_n* which goes to all NoC elements within the affected power domain. Note: if step 2 above and the sub-steps below do not all complete within a timeout window, NSPS will move to SLEEP\_NACK state to return NACK to PMU by raising *QDENY\_<PD>*.
   1. NoC element waits until it is safely idle – no traffic pending.
   2. NoC element de-asserts *link\_available\_<element>\_<ifce>* to upstream interfaces to prevent new traffic arriving. Note these only need to be connected for interfaces crossing the power domain boundary.
   3. NoC element asserts *sleep\_ack\_<element>\_n* to signal power down readiness to NSPS.
   4. When all elements have asserted *sleep\_ack\_<element>\_n*, NSPS advances to SLEEP\_ACK state.
5. (State = SLEEP\_ACK) NSPS changes *QACCEPTn\_<PD>* to be low, acknowledging power down readiness. If *reset\_pd\_<PD>\_n* is driven by NSPS (not default option), it would be asserted here. PMU may now remove clocks and/or power.



Figure 3 Power Down Sequence Waveforms

### Power Up Sequence Details

Starting in the SLEEP\_ACK state: *QREQn\_<PD>*, *QACCEPTn\_<PD>*, *reset\_pd\_<PD>\_n* and *pd\_active\_<PD>* are all low.

1. (State = SLEEP\_ACK) PMU signals clock/power restoration by driving *QREQn\_<PD>* high. Clocks and power should already be up and stable before this happens. NSPS advances to the WAKE\_ACK state.
2. (State = WAKE\_ACK) NSPS de-asserts *sleep\_req\_<PD>\_n*, and if it is controlling *reset\_pd\_<PD>\_n*, it de-asserts this, too. It waits for *sleep\_ack\_deasserted\_<PD>* to go active to confirm all elements in the power domain are ready for activity before advancing to the REMOVE\_FENCE state.
3. (State = REMOVE\_FENCE) NSPS asserts *pd\_active\_<PD>* to NoC to inform masters they should cease fencing and draining for this power domain. It waits for *fence\_ack\_deasserted\_<PD>\_n* to go active to confirm all masters have ceased fencing for the domain before advancing to the ACTIVE state.
4. (State = ACTIVE) NSPS drives *QACCEPTn\_<PD>* high, completing acknowledgment of power up, indicating logic is ready for normal operation.



Figure 4 Power Up Sequence Waveforms

### Power on Reset

At cold reset, all the power domains defined for the NoC and all elements within them must be placed in their sleep state, i.e., the entire NoC is in the sleep state. Each power domain must be explicitly moved from the SLEEP\_ACK (Q\_STOPPED) to the ACTIVE (Q\_RUN) state via the Q-Channel interface after reset de-assertion. This allows different parts of the device to be initially powered up at different times, including scenarios where only a portion of the device is active for booting, and other portions may not receive initial power for an extended period of time. This is consistent with the ARM AMBA Low Power Interface Specification which requires the reset state of the Q-channel interface to be Q\_STOPPED (the sleep state), where both *QACCEPTn* and *QDENY* are driven low. *QREQn* may be driven either high or low at reset de-assertion – if driven high, the PD FSM will immediately begin the transition from SLEEP\_ACK to ACTIVE state.

The low power signaling described in the sections that follow have reset values specified that are consistent with these requirements, in particular:

* *QACCEPTn\_<PD> & QDENY\_<PD>*: 0 for all power domains.
* *pd\_active\_<PD>*: 0 for all power domains.
* *sleep\_req\_<PD>\_n*: 0 for all power domains – PD FSM is asserting request for elements to be asleep.
* *sleep\_ack\_<element>\_n*: 0 for all elements across all power domains, indicating that they are in the sleep state.
* *link\_available\_<element>\_<ifce>*: 0 for all link interfaces.

Relying on the low power mechanisms to explicitly confirm proper exit from sleep state helps guarantee clean exit from cold reset, regardless of the order and timing of reset de-assertion at individual NoC elements. Each element must respond to de-assertion of *sleep\_req\_<PD>\_n* (going high) by de-asserting *sleep\_ack\_<element>\_n* (going high) before the PD FSM will advance from the WAKE\_ACK to the REMOVE\_FENCE state. Until it does, *pd\_active\_<PD>* remains low, causing any all masters to continue to fence incoming traffic that is dependent upon the power domain. In other words, all NoC elements must explicitly confirm they are ready for traffic before any such traffic is allowed to enter the NoC.

## Signaling

Interfaces between the layers of the architecture are detailed below, but the various interfaces share some general characteristics discussed here. The signals involved must cross voltage, power and clock domains. Therefore, the signals are handled asynchronously between source and destination(s). To facilitate simple aggregation of these signals with minimum logic and wiring, wherever possible the protocols are defined such that the signals are level based. When an event causes an agent to assert a signal, it is held until a reciprocal level based transition is observed on another signal, at which point it is de-asserted to complete the handshake.

To ensure the robustness of these interfaces, the following implementation rules apply.

1. All low power output signals must be sourced directly from a register, with no combinatorial logic between flop and module boundary. The only combinatorial logic allowed is the aggregation logic explicitly described in the sections below.
2. All low power input signals must be properly synchronized to the local clock domain.
3. Power intent files (CPF/UPF) must insert proper isolation and level-shifting in these signal paths where they cross power and voltage domain boundaries.

## NetSpeed Power Supervisor Interface to PMU

The interface between the NetSpeed Power Supervisor and the PMU follows the AMBA Low Power Interface Specification, specifically the Q-Channel defined there (see “1.23 - References, #2: ARM AMBA Low Power Interface Specification” for a link to the spec).

The *reset\_pd\_<PD>\_n* signal is also brought out to be driven directly by the PMU.

One set of the signals is implemented per customer defined power domain. The interface will not change through NoC design iterations as long as the power domains are not changed.

| **Signal Name** | **Source** | **Destination** | **Purpose** | **Active/Clamp & Reset Values** |
| --- | --- | --- | --- | --- |
| QREQn\_<PD> | PMU | NSPS | Inform NS logic of intent to remove/restore power from/to power domain <PD>. When transitioning 1->0, requests power-down, when transitioning 0->1, requests wake-up. | 0 / 0 |
| QACCEPTn\_<PD> | NSPS | PMU | Response to *QREQn\_<PD>*, acknowledges request for power down or power up. | 0 / 0 |
| QDENY\_<PD> | NSPS | PMU | Asserted instead of *QACCEPTn\_<PD>* to reject a power down request. Its presence is optional, controlled by the *pd\_qdeny\_present* property. | 1 / 0 |
| QACTIVE\_<PD> | NSPS | PMU | When powered down (*QREQn\_<PD>* and *QACCEPTn\_<PD>* are both low, NSPS in SLEEP\_ACK state), *QACTIVE\_<PD>* is asserted to signal a wake up request. In other states, it serves to indicate the idle status of NoC logic within the power domain. NSPS generates this signal combinatorially as (*idle\_status\_<PD>\_n* || *wake\_req\_asserted\_<PD>*). Its presence is optional, controlled by the *pd\_qactive\_present* property. | 1 / 0 |
| reset\_<PD>\_n | PMU | elements | Per power domain cold reset. This is asserted to establish clean state following the first power-on event for the power domain. Point to multipoint, subject to clock/voltage domain crossing treatment. | 0 / 0 |
| reset\_pd\_<PD>\_n | PMU | elements | Per power domain warm reset. This is asserted to restore clean state after a power removal/restoration sequence. Point to multipoint, subject to clock/voltage domain crossing treatment. | 0 / 0 |
| clk\_nsps | PMU | NSPS | This is the clock used for the PD FSMs in the NSPS. All I/Os for these FSMs are defined to be asynchronous, so inputs are locally synchronized to this clock, and it is expected that PD FSM outputs are synchronized to locally used clocks at their destinations. As such, the frequency requirements for this clock are arbitrary… there is no expected relationship between this clock and others in the NoC. However, it must be running for each PD FSM whenever that PD FSM is not in the SLEEP\_ACK state and/or needs to respond to any change on its inputs, including resets. | toggling |

### QDENY\_<PD>

The ARM low power signaling protocol allows for elements to NACK a sleep request by asserting *QDENY\_<PD>* instead of *QACCEPTn\_<PD>* in response to *QREQn\_<PD>* going low. Support of this signal is optional and is configured via the *pd\_qdeny\_present* property. See section 1.9.1.4 – “QDENY\_<PD> and the SLEEP\_NACK and REMOVE\_FENCE\_NACK States” for details of our implementation.

### QACTIVE\_<PD>

*QACTIVE\_<PD>* serves as the “device activity indication” portion of the Q-Channel interface (vs. the “handshake mechanism” comprised of the other 3 Q-signals), and it operates completely independently of the other signals. The handshake signals, *QREQn\_<PD>*, *QACCEPTn\_<PD>*, and *QDENY\_<PD>* are sunk and sourced in the Power Domain FSM, while *QACTIVE\_<PD>* is generated outside the FSM combinatorially as:

*QACTIVE\_<PD> = idle\_status\_<PD>\_n || wake\_req\_asserted\_<PD>;*

Note the ARM AMBA Low Power Interface Specification requires that *QACTIVE\_<PD>* “must be driven either directly by a register or by a number of registers whose outputs are combined using a logical OR.” Our implementation complies as *idle\_status\_<PD>\_n* and *wake\_req\_asserted\_<PD>* are themselves aggregated from flop outputs via an OR function. The roots of these combinatorial trees are generally sourced from multiple power and clock domains in the NoC, and the final stages will need to be in the PMU power domain (likely always on).

Use of *QACTIVE\_<PD>* is optional, controlled by the *pd\_qactive\_present* property. If this is false, *QACTIVE\_<PD>* is removed from the Q-channel interface.

As can be observed from the formula above, there are 2 components to *QACTIVE\_<PD>*, an idle status indication and a wake request. These are related but distinguished by the fact that the former is relevant when the power domain in question is active, while the latter applies when the power domain is inactive. NocStudio provides additional properties described below that allow customer selection of which of the components are included in *QACTIVE\_<PD>*.

Customer use of the idle status component is controlled by the *pd\_idle\_status\_enable* property (see section 1.14.8.1 #1.c). It is intended to support power management decision making, but it is up to customers to decide whether it or not it is useful for their application. If *pd\_idle\_status\_enable* is off/false, then genrtl should not generate the *idle\_status\_<PD>\_n* network.

Customer use of the wake request component is controlled by the *pd\_autowake\_enable* property (see section 1.14.8.1 #1). NoC initiated wake-up (0->1 transition of *QACTIVE\_<PD>* for powered- down domain) may or may not be required depending on the specifics of the implementation. In the case where fencing and draining can be reliably implemented to guarantee no pending activity for a given power domain remains (e.g., pure AMBA deployment), wake requests should never occur. In other deployments such as Gemini and streaming, we have no mechanism to reliably determine that no such pending activity remains in the NoC, so we rely on the wake mechanism to guarantee clean recovery from circumstances where traffic is blocked by powered down logic in the NoC. We would broadly recommend customers respond to *QACTIVE\_<PD>* 0->1 transitions to bring up power domains upon demand to avoid blocking progress in the NoC. If *pd\_autowake\_enable* is off/false, then genrtl should not generate the *wake\_req\_asserted\_<PD>* network.

If *pd\_qactive\_present* is true, but neither *pd\_idle\_status\_enable* or *pd\_autowake\_enable* are on/true, *QACTIVE\_<PD>* should be driven constantly low.

### reset\_<PD>\_n and reset\_pd\_<PD>\_n

Two reset signals are driven by the PMU for each power domain. The first, *reset\_<PD>\_n*, is the power on or cold boot reset which is applied to establish known good state after power is applied for the first time. This is the same chip reset that applies in a non-low power implementation. Per power domain versions are required for low power implementations to allow different portions of the chip to be powered and booted from a cold start at different times.

The second signal, *reset\_pd\_<PD>\_n*, is a “warm” reset signal that is applied to restore known good state in the wake of power cycles that have occurred since initial power up. The principle motivation is to support retention structures that retain their state through power cycle events (e.g., retention flops). These structures are only reset by the cold reset signal, *reset\_<PD>\_n*, whereas the remaining logic is reset by the AND of both signals. A quick wake up state which only deploys clock gating can be implemented where *reset\_pd\_<PD>\_n* is not asserted upon wake up – this allows state to be retained without special structures.

Both reset signals are applied asynchronously from their assertion edge (going low), but are locally synchronized so that their de-assertion (going high) is applied synchronously.

Note, per ARM Q-channel spec, *reset\_pd\_<PD>\_n* and *reset\_<PD>\_n* may only be asserted when Q-channel is in Q\_STOPPED state (PD FSM in SLEEP\_ACK - *QREQn\_<PD>* and *QACCEPTn\_<PD>* both low), but they may be deasserted in either Q\_STOPPED or Q\_EXIT state (in the latter PD FSM is in SLEEP\_ACK state with *QREQn\_<PD>* high and *QACCEPTn\_<PD>* low).

When PD FSM is not in the same power domain as the one it is controlling, it does not see *reset\_pd\_<PD>\_n* or *reset\_<PD>\_n*. However, if either are asserted while PD FSM is in the WAKE\_ACK state, it will wait there until *sleep\_ack\_deasserted* goes high, which should not happen while NoC elements are held in reset.

## NetSpeed Power Supervisor Interface to NoC Elements

The power supervisor maintains a separate state machine that tracks the power status for each power domain within the NoC. This power status is broadcast to all master/initiator elements that need to know the availability of each power domain to determine when to execute fencing and draining transitions as well as when to issue auto-wake requests.

Following is the set of per power domain signals between the power supervisor and master/initiator elements in the NoC. Note: all NoC element output signals enumerated below must be sourced directly from a flop to satisfy requirements for clean clock domain crossing.

| **Signal Name** | **Source** | **Destination** | **Purpose** | **Active/Clamp & Reset Values** |
| --- | --- | --- | --- | --- |
| pd\_active\_<PD> | NSPS | master | Indicates whether targets within the power domain are active and reachable. Indicates power domain is either turned off, or in the process of being turned off. Point to multi-point, subject to clock/voltage domain crossing treatment. | 1 / 0 |
| fence\_ack\_<PD>\_<master>\_n | master | NSPS | Signal indicating *pd\_active\_<PD>* 1->0 transition has been observed and fencing has begun. Should remain asserted until 0->1 transition on *pd\_active\_<PD>*. Point to point, subject to combinatorial aggregation to create *fence\_ack\_asserted\_<PD>\_n* with all such signals targeting the same power domain. | 0 / 0 |
| fence\_done\_<PD>\_<master>\_n | master | NSPS | Inverted version of *fence\_ack\_<PD>\_<master>\_n*, except when master is going to sleep (i.e., it is asserting *sleep\_ack\_<element>\_n*), it forces this signal low. Point to point, subject to combinatorial aggregation to create *fence\_ack\_deasserted\_<PD>\_n* with all such signals targeting the same power domain. | 0 / 0 |
| drain\_ack\_<PD>\_<master>\_n | master | NSPS | Signal indicating draining associated with *pd\_active\_<PD>* 1->0 transition has completed. Should remain asserted until 0->1 transition on *pd\_active\_<PD>*. Point to point, subject to combinatorial aggregation to create *drain\_ack\_asserted\_<PD>\_n* with all such signals targeting the same power domain. | 0 / 0 |
| autowake\_<PD>\_<master> | master | NSPS | Request to restore power in response to new transactions on host interface attempting to reach target within power domain. Point to point, subject to combinatorial aggregation with all wake signals targeting the same power domain. | 1 / 0 |

Masters respond to 1->0 transitions of *pd\_active\_<PD>* signals by initiating fencing and draining processes for transactions targeting the affected power domain. Once any pending transactions have drained, *drain\_ack\_<PD>\_<master>\_n* is asserted.

The power supervisor also manages sleep request/acknowledgment interfaces to each element within the NoC. In response to PMU *QREQn\_<PD>* 1->0 transition, the NS power supervisor must issue sleep requests to all affected NoC elements and aggregate their acknowledgments to generate *QACCEPTn\_<PD>* going back to PMU. Similar aggregation must be done with *idle\_status\_<element>\_n*. The following signals are implemented for each NoC element.

| **Signal Name** | **Source** | **Destination** | **Purpose** | **Active/Clamp & Reset Values** |
| --- | --- | --- | --- | --- |
| sleep\_req\_<PD>\_n | NSPS | element(s) | Inform NS logic of intent to remove power from power domain <PD>. Point to multi-point, subject to clock/voltage domain crossing treatment. | 0 / 0 |
| sleep\_ack\_<element>\_n | element | NSPS | Response to *sleep\_req\_<PD>\_n*, indicates NoC logic is in state safe for power removal. Point to point, subject to combinatorial aggregation with like signals originating from the same power domain. | 0 / 0 |
| wake\_req\_<element>\_<ifce> | element | NSPS | Request to restore power to power domain associated with the element at the far end of an interface that is stalled by its neighbor’s de-assertion of *link\_available\_<element>\_<ifce>*. Only connected by gen\_rtl in NocStudio where an interface crosses a PD boundary for which the *pd\_autowake\_enable* property is set true. Point to point, subject to combinatorial aggregation with all wake signals targeting the same power domain. | 1 / 0 |
| idle\_status\_<element>\_n | element | NSPS | Indicates logic within element is idle such that power removal would be OK. Point to point, subject to combinatorial aggregation with like signals associated with the same power domain. | 0 / 0 |

### Signal Aggregation

The power supervisor must aggregate individual signals coming from multiple NoC elements to make power domain level decisions. The asynchronous level based nature of the signals allows this aggregation to be done combinatorially in a manner that can easily be distributed to facilitate physical design. To implement clean handshakes in the Power Domain FSM, it is important to detect universal assertion and universal de-assertion, and it is also necessary to handle the case where one or more of the signal sources is in a disabled state (i.e., is generated by logic that is in a power domain that is currently turned off).

Consider the case of the *fence\_ack\_<PD>\_<master>\_n* signals. The default “operational” state of these signals is de-asserted (1). When the PMU toggles *QREQn\_<PD>* low, the Power Domain FSM in the Power Supervisor de-asserts the associated *pd\_active\_<PD>* signal. In response to a 1->0 transition of a *pd\_active\_<PD>* signal, all masters in the NoC which have valid traffic flows terminating in the associated power domain must go into a fence and drain state, where new transactions targeting that power domain are held off at the host interface (fencing), and any pending transactions are monitored for completion. Once this process has completed, the master asserts its *fence\_ack\_<PD>\_<master>\_n* and *drain\_ack\_<PD>\_<master>\_n* signals (the latter will generally lag the former).

The power supervisor must aggregate the *fence\_ack\_<PD>\_<master>\_n* signals from all the masters that could be communicating with the affected power domain. This is an OR function creating *fence\_ack\_asserted\_<PD>\_n* – all masters must confirm before the Power Domain FSM can advance to the next state to assert sleep requests to all NoC elements living in that domain.

Once power is restored to a given power domain, the Power Domain FSM must also be able to confirm that all masters have cleanly de-asserted their *fence\_ack\_<PD>\_<master>\_n* signals for an active power domain. This requires generation of another aggregated signal, *fence\_ack\_deasserted\_<PD>\_n*, which is an OR of the individual *fence\_done\_<PD>\_<master>\_n* signals. After driving *pd\_active\_<PD>* 0->1, the Power Domain FSM waits for *fence\_ack\_deasserted\_n* to transition 1->0 before advancing to the ACTIVE state.

The aggregated signals described below are synchronized into the clock domain of the Power Domain FSM that uses them. Following are the set of aggregated signals created in the power supervisor (possibly in a physically distributed fashion).

| **Signal Name** | **Description** | **Active/ Clamp & Reset Values** |
| --- | --- | --- |
| fence\_ack\_asserted\_<PD>\_n | Indicates all masters which can send traffic to a power domain for which *pd\_active\_<PD>* has transitioned 1->0 have begun fencing. OR of *fence\_ack\_<PD>\_<master>\_n* signals. | 0 / 0 |
| fence\_ack\_deasserted\_<PD>\_n | Indicates all masters which can send traffic to a power domain for which *pd\_active\_<PD>* has transitioned 0->1 have asserted *fence\_done\_<PD>\_<master>\_n*, which implies all have de-asserted *fence\_ack\_<PD>\_<master>\_n* (implying they are not fencing) or they are powered down. OR of *fence\_done\_<PD>\_<master>*\_n. See “1.7.1.1 – Special Handling of fence\_ack\_deasserted\_<PD>” below. | 0 / 0 |
| drain\_ack\_asserted\_<PD>\_n | Indicates all masters which can send traffic to a power domain for which *pd\_active\_<PD>* has transitioned 1->0 have completed draining. OR of *drain\_ack\_<PD>\_<master>\_n* signals. | 0 / 0 |
| sleep\_ack\_asserted\_<PD>\_n | Indicates all NoC elements in a power domain for which *sleep\_req\_<PD>\_n* has been issued have reached state for safe power removal. OR of *sleep\_ack\_<element>\_n*. | 0 / 0 |
| sleep\_ack\_deasserted\_<PD> | Indicates all NoC elements in a power domain for which *sleep\_req\_<PD>\_n* has been removed (power cycle or otherwise) have de-asserted their *sleep\_ack\_<element>\_n* signals. AND of *sleep\_ack\_<element>\_n*. | 1 / 0 |
| wake\_req\_asserted\_<PD> | Indicates 1 or more *autowake\_<PD>\_<master>* or *wake\_req\_<element>\_<ifce>* signals associated with power domain <PD> has been raised. OR of autowake and wake\_req signals. Only generated if *pd\_autowake\_enable* is on/true. | 1 / 0 |
| idle\_status\_<PD>\_n | OR of all *idle\_status\_<element>\_n* signals associated with power doman <PD>. Only generated if *pd\_idle\_status\_enable* is on/true. | 0 / 0 |

#### Special Handling of fence\_ack\_deasserted\_<PD>\_n

Some master bridges may be in inactive power domains and therefore unable to properly respond to changes to *pd\_active\_<PD>*. By definition, such masters are already fencing and draining for all traffic, so the reset/isolation clamped state of all their *fence\_ack\_<PD>\_<master>\_n* and *drain\_ack\_<PD>\_<master>\_n* signals is asserted, and for this reason, nothing special needs to be done in handling the aggregation of these signals to create *fence\_ack\_asserted\_<PD>\_n* and *drain\_ack\_asserted\_<PD>\_n*. However, *fence\_ack\_deasserted\_<PD>\_n* aggregation needs to account for powered down masters by masking (forcing inactive) all *fence\_ack\_<PD>\_<master>\_n* signals sourced from inactive power domains.

To support the proper qualification of *fence\_ack\_<PD>\_<master>\_n* signals, the masters drive a 2nd signal that is derived from it. This signal, *fence\_done\_<PD>\_<master>\_n*, is created by inverting *fence\_ack\_<PD>\_<master>\_n* and qualifying it to force it low whenever the master is asserting *sleep\_ack\_<element>\_n*.

*fence\_done\_<PD>\_<master>\_n = ~fence\_ack\_<PD>\_<master\_src\_PD>\_n & sleep\_ack\_<element>\_n;*

#### Clock Domain Crossing (CDC) Considerations

A standard requirement for safe synchronization of signals crossing clock domains is to guarantee glitch free transitions in the clock domain of origin by sourcing signals directly from flops with no logic intervening before the synchronizer input. That is not practical to achieve for several of the aggregated networks described above as they generally are sourced from multiple clock/power/voltage domains across significant distance. The aggregated signals in the list above (and other signals like them such as interrupts).

It is okay to waive this requirement for the aggregated LP networks for the following reasons:

1. The contributing signals are sourced directly from flops at the NoC element outputs.
2. The networks implement simple, non-recombinant functions: AND, OR
3. The contributing signals are level-based
   1. Transitions for many of the signals are controlled via handshake through the PD FSM such that they are guaranteed to be stable through the synchronization process.
   2. Remaining singles, *idle\_status\_<PD>\_n* and *wake\_req\_asserted\_<PD>*, are incorporated into *QACTIVE\_<PD>* and meet requirements for this signal per ARM spec.

## NoC Elements

Aside from the signals described above, each NoC element implements an availability signal for each interface that when de-asserted holds off incoming transactions at packet boundaries. These availability signals are de-asserted in response to sleep requests.

| **Signal Name** | **Source** | **Destination** | **Purpose** | **Active/Clamp & Reset Values** |
| --- | --- | --- | --- | --- |
| link\_available\_<element>\_<ifce> | element | neighbor | Holds off incoming transactions a packet boundary. | 1 / 0 |

Any upstream neighbor that has pending traffic targeting a blocked interface must raise its *wake\_req\_<element>\_<ifce>* signal to request that the downstream element be woken up to allow traffic to proceed. This replaces and consolidates functionality of “link not available” signals from the LP V1 architecture, and it serves to synchronize the NoC as it comes out of chip reset (cold boot) as well as power state changes. Packets are held pending at the upstream element until the downstream *link\_available\_<element>\_<ifce>* is asserted.

When regbus support is enabled, *wake\_req\_<element>\_<ifce>* events should be captured as a maskable interrupt bit which is masked by default.

### Reset == Idle == Clamp == 1’b0

All NoC element interface signals that are reset (i.e., control signals whose state matters around reset events vs. datapath signals that are don’t-cares at such points) should be assigned polarity such that their value during reset is 0. Exceptions to this rule should be limited to only include those signals that are defined by industry protocol to have non-zero reset value. Furthermore, it is expected that these interface signals will be at their reset value when the NoC element is idle such as would be required to enter sleep/power-down. In other words, before asserting *sleep\_ack\_<element>\_n*, all control interface signals should be at their reset value, which should be low for nearly all such signals.

There are 2 motivations for this requirement:

1. Simplifies CPF/UPF implementation driving insertion of isolation clamps by enabling a blanket directive to clamp all interfaces to low value with few exceptions, yielding a more robust, maintainable and less error prone solution.
2. May simplify static timing analysis by increasing comfort in relaxing requirements on paths related to isolation clamp enables – if this requirement is enforced, the clamp cell value doesn’t change when isolation enable changes, so paths originating from isolation enables may be treated as false or multi-cycle.

Regarding the latter motivation, we don’t own physical implementation of isolation clamps or the details of related timing analysis. It is up to the user to decide how best to handle this. In general, there are paths from isolation clamp enable signals to the destinations of the signals being isolated. The most robust solution is to locally synchronize the isolation enables to the destination clock domain to make these paths synchronous such that they can be timed and implemented to meet setup and hold requirements. Some users will likely choose to do this, others may be comfortable treating the enable paths as false arguing that there is no functional risk given adherence to the reset==idle==clamp rule.

We are choosing not to add additional logic to reset non-control interface signals. Users that implement isolation enables asynchronously must accept that there is risk that the receiving flops could go meta-stable around the transition of the isolation enable. One alternative option that may be available in some cases is to deploy isolation clamps that retain the input signal level during isolation.

*TBD – for consideration as a future enhancement: support for level-preserving isolation clamps (complicates CPF/UPF, requires cell availability in library, etc.). We should only consider resetting datapaths if driven by strong customer feedback.*

### Interrupts

One or more interrupt signals may be asserted at the time the PMU requests a power down of the domain that contains the NoC elements driving those interrupts. We do not implement any logic that prevents normal progression to Q\_STOPPED status on behalf of such interrupts. It is up to the user to determine whether or not their PMU implementation considers interrupt status in managing power domains.

If power is removed while one or more interrupt status bits is set within the power domain, this interrupt status will be lost (assuming no implementation of retention registers to hold the state). Furthermore, when asserted, interrupt outputs represent an exception to the reset == idle == clamp rule. These signals are active high, but we take no action to suppress them as part of the *sleep\_req\_n*/*sleep\_ack\_n* handshake process. If an interrupt is asserted when isolation clamping takes effect, the clamped value will change from 1 -> 0 asynchronously. This is deemed OK as these interrupt signals are aggregated asynchronously in normal operation.

Interrupt status should not be considered in idle status – *it must never block progress in* sleep\_req\_n/sleep\_ack\_n *handshake.*

## Power Supervisor State Machine

The power supervisor maintains a simple state machine per power domain to manage the interfaces to PMU and NoC elements, which includes providing some handshaking to protect against lag in expected transitions in the level based signals from the NoC elements due to delays associated with clock and voltage domain crossings.

### Power Domain FSM

This state machine (one per power domain) handles the *QREQn\_<PD>/QACCEPTn\_<PD>/QDENY\_<PD>* (Q-Channel) interface with the PMU and maintains the *pd\_active\_<PD>* state for the domain. As long as the PMU properly restores power and clocks prior to raising QREQn, the FSM logic may be mapped into the power domain it serves.

module power\_domain\_fsm (

output QACCEPTn,

output QACTIVE,

output pd\_active,

output sleep\_req\_n,

output reset\_pd\_n,

output pd\_status,

input QREQn,

input wake\_req\_asserted,

input fence\_ack\_asserted\_n,

input fence\_ack\_deasserted\_n,

input drain\_ack\_asserted\_n,

input sleep\_ack\_asserted\_n,

input sleep\_ack\_deasserted,

input reset\_n,

input clk

);

Parameter

PARAM\_ROUND\_TRIP[7:0] = 8’h1f;

PARAM\_SLPTMOUT[15:0] = 16’hffff;

PARAM\_SLPDLY[7:0] = 8'h80;



Figure 5 Power Domain FSM

Note: items shown in blue indicate optional or parameterized states, transitions and signals where we may or may not need to include these items on a case by case basis.

#### Sleeping and Waking Master Bridges

When a master bridge is put to sleep via the *sleep\_req\_n*/*sleep\_ack\_n* handshake, it must force its outputs to the idle == reset == clamp == 1’b0 state. This means that it can no longer actively participate in the fencing and draining handshakes for other power domains. Its *fence\_ack\_<element>\_n*, *fence\_done\_<element>\_n* and *drain\_ack\_<element>\_n* outputs are both driven low, which unblocks fencing transitions for other power domains in both directions (initiating fencing and draining for power down and removing fencing for power up).

When the bridge wakes back up, a hazard is created as it re-engages with the fencing and draining handshakes for other power domains in which it participates. This is due to the propagation latency of the asynchronous signals between the master bridge and the PD FSMs managing the other power domains.



Figure 6 Waking Master Bridge Fencing and Draining Hazard Waveforms

Figure 6 above illustrates the hazard where power domain PD0 is transitioning from sleep to active around the same time that power domain PD1 is transitioning in the opposite direction. A master bridge being woken in power domain PD0, upon observing the de-assertion of its locally synchronized *sleep\_req\_local\_sync\_PD0\_n*, begins responding to its locally synchronized copies of all the *pd\_active* signals, including that for PD1. Based on the state of each *pd\_active* input, it determines whether to transition into or out of an active fencing state, and it de-asserts either *fence\_ack\_<element>\_n* or *fence\_done\_<element>\_n* accordingly by driving that signal high. At the time that it observes its locally synchronized *pd\_active\_local\_sync\_PD1* signal, it sees that it is high, so it de-asserts *fence\_ack\_<element>\_PD1\_n*.

However, by the time *fence\_ack\_<element>\_PD1\_n* propagates, gets aggregated and synchronized at the destination, PD1 FSM has already transitioned to the FENCE\_DRAIN state and started driving *pd\_active\_PD1* low. Here it observes a stale version of *fence\_ack\_asserted\_PD1\_n* being low due to the master bridge having been asleep. This causes PD1 FSM to advance to the SLEEP\_REQ state before *fence\_ack\_asserted\_PD1\_n* is observed going high (too late). Some cycles later *fence\_ack\_asserted\_PD1\_n* does go low reflecting synchronized fencing status.

The hazard window is indicated in Figure 6. We would be in trouble if the master bridge received a new transaction on its host interface and forwarded it into the NoC during this window. That transaction would be subject to being indefinitely stalled, corrupted, or lost because PD1 FSM moves on prematurely to SLEEP\_REQ and asserts *sleep\_req\_PD1\_n* (not shown in the Figure 6). Fortunately, this is very unlikely to occur in because the master bridge blocks incoming transactions on its host interface until it sees the locally synchronized *pd\_active* for its own power domain go high. The latency from the moment the bridge wakes to observing its own *pd\_active* going high, which involves a round trip between the bridge and its PD FSM, will generally be longer than the one way latency for its *fence\_ack\_n* signals to propagate to the other PD FSMs and equally important the one way latency for the other *pd\_actives* to arrive at the bridge. This is reflected in Figure 6, where *pd\_active\_local\_sync\_PD0* goes high after *pd\_active\_local\_sync\_PD1* goes low. In this case, no harm comes because the bridge properly starts fencing for PD1 before it even activates its host interface, so no PD1 dependent transactions will be allowed into the NoC.

##### Prop\_delay\_timer\_exp

To provide additional margin for the hazard described above, the PD FSM implements a small timer that compensates for the latency through its synchronizers capturing the *fence\_ack\_asserted\_n*/*fence\_ack\_deasserted\_n* signals. The PD FSM stalls progress in critical states until this timer expires, increasing margin from the hazard window by the duration of the timer. The timer is set to the number of synchronizer stages implemented within the PD FSM plus 1, a value that avoids increasing latency for responding to power sequencing events.

The propagation delay timer is reset and started at the following state transitions which correspond to change in value of *pd\_active\_<PD>*:

* ACTIVE -> FENCE\_DRAIN: blocks FENCE\_DRAIN -> SLEEP\_WAIT/SLEEP\_REQ and FENCE\_DRAIN -> SLEEP\_NACK
* FENCE\_DRAIN -> SLEEP\_NACK: blocks REMOVE\_FENCE\_NACK -> ACTIVE
* SLEEP\_WAIT -> SLEEP\_NACK: blocks REMOVE\_FENCE\_NACK -> ACTIVE
* WAKE\_ACK -> REMOVE\_FENCE: blocks REMOVE\_FENCE -> ACTIVE

##### Avoiding Hazard Entirely

The hazard is completely avoided if only one Q-channel is ever between Q\_RUN and Q\_STOPPED at any point in time. This constraint only needs to apply to Q-channels that control power domains that contain fencing elements. In practice, timing margins should make this constraint unnecessary, particularly if all PD FSMs are placed together and clocked with the same clock.

#### SLEEP\_WAIT

This state is intended support streaming (ORION-NSIP) implementations and perhaps Gemini cases where it is impossible for a source bridge to know when a packet has reached its destination due to lack of responses in the protocol. It provides delay from the time *fence\_ack\_asserted\_<PD>\_n* goes active before entering the SLEEP\_REQ state to allow packets in flight time to reach their destinations. It should be unnecessary for a purely AMBA implementation and is therefore shown in blue to indicate it is not always required.

#### QREQn\_<PD> Abort is Illegal

The AMBA Q-Channel protocol does not allow the PMU to abort the transition initiated by changing state of *QREQn\_<PD>*. After changing the state of *QREQn\_<PD>*, the PMU is required to hold its state until the NSPS responds with *QACCEPTn\_<PD>* or *QDENY\_<PD>*. The FSM ignores *QREQn\_<PD>* except when it is in a state where it is legal for it to change value.

If we find for some applications this capability is required, particularly where the PMU does not readily support *QDENY\_<PD>*, we would have to consider a modification to the Q-Channel protocol that would make a PMU initiated abort possible. The cleanest way to do this would be require the interface to be synchronous between PMU and NSPS so that each side could reliably know that the other has observed changes to the signal states. This is not planned, and the details of such a modification remain to be defined.

#### QDENY\_<PD> and the SLEEP\_NACK and REMOVE\_FENCE\_NACK States

Handling autowake for a power domain requires supporting *QDENY\_<PD>* to allow the PD FSM to abort a power down request from the PMU if a wake request is received during the fencing and draining phase (between ACTIVE and SLEEP\_REQ states). Without it, deadlock can occur in certain autowake scenarios (described further in Staged Fencing and Autowake Confluence Page). Any power domain for which the PMU cannot support QDENY must be configured to disable autowake.

The FSM has arcs to the SLEEP\_NACK state from the FENCE\_DRAIN and SLEEP\_WAIT states to allow it to assert *QDENY\_<PD>* to abort an incoming Q\_REQUEST for one of 2 reasons:

1. A NoC element has asserted a wake\_req for the affected PD.
2. The *sleep\_timeout[15:0]* counter, which is started from the falling edge of *QREQn\_<PD>*, has expired, indicating we have failed to achieve a safe sleep state within the defined time limit.

To avoid breaking the the pd\_active/fence\_ack handshake, the PD FSM waits for *fence\_ack\_asserted\_<PD>\_n* to go low before responding to a wake request with a transition to the SLEEP\_NACK state.

The inclusion of the 2 sources of *QDENY\_<PD>* enumerated above is configurable via the *pd\_autowake\_enable*  and *pd\_qdeny\_timeout\_enable* properties in NocStudio (see section 1.14.8.1). When regbus is enabled for the design, these properties control reset default values for registers that dynamically control use of wake\_req and timeout respectively (including timeout threshold value). When regbus is not present and both of these properties are set false, *QDENY\_<PD>* should be driven constantly low and the SLEEP\_NACK and REMOVE\_FENCE\_ACK states may be eliminated. Furthermore, if *pd\_autowake\_enable* is set false, *wake\_req\_asserted\_<PD>* is not implemented and logic associated with wake\_req is eliminated.

Note that if *QDENY\_<PD>* is not supported, latency in responding to a sleep request is unbounded. For example, if an SoC host takes a really long time to respond to a pending transaction, the FSM will be stalled in the FENCE\_DRAIN state waiting for the response to complete. In the meantime, new transactions could arrive at the fenced interface of the master (fencing starts right away), and we would have no way to abort the sleep transition to allow these transactions to succeed. Generally we might expect that customer implementations will avoid such scenarios, so this is not expected to be a significant problem in practice, but it is something to be considered when choosing not to support *QDENY\_<PD>*.

#### PD FSM Debug Visibility

When regbus is enabled, the following signals should be brought to read-only registers for debug visibility:

1. PD FSM State Vector: what is the current state of the FSM
2. Q-channel interface

When *QDENY\_<PD>* is implemented, first entry into the SLEEP\_NACK state should be logged in a clearable register with indication of the state from which SLEEP\_NACK was entered.

### Power Domain Interdependencies – TBD Interlock

As currently defined above, the FSM does not implement any checking to prevent entering a power state where through traffic is blocked when routers (or other elements) in the pathway are powered down. It could be modified to add a state between ACTIVE and FENCE\_DRAIN where such that the set of power domains requested for power down are checked against the current power state of all domains (*pd\_active\_<PD>*), and for any power domain where dependent power domains will remain active, the response would be to NACK the request (by advancing to SLEEP\_NACK instead of FENCE\_DRAIN). *Need to decide if we should implement this interlock or not.*

## Fundamental Architecture of Fencing and Draining

### Two Phases: Fencing/Draining and Sleep\_req/Sleep\_ack

The NSPS manages 2 distinct phases of preparation for sleeping and waking power domains within the NoC in response to requests from the Q-channel.

1. Fence and Drain: renders the NoC "transaction idle" in the portion affected by a Q-Channel sleep request (QREQn 1->0).
   1. Fencing is initiated in response to pd\_active\_<PD> transitioning from 1->0 either for an element’s own power domain or for a downstream element for which the element has fencing responsibility.
   2. All pending transactions are allowed to complete.
   3. All newly arriving transactions are prevented from reaching the affected area of the NoC in a manner that does not break protocol.
2. Sleep Request: renders the portion affected by Q-Channel sleep request properly "electrically idle."
   1. Interfaces adhere to idle == reset == clamp.
   2. If fencing is working, there should be no volatile state left to drain at this point.

All transaction draining should happen during fence and drain phase. Fencing is implemented only in “master” elements that are in a position to control ingress into a portion of the NoC.

### Fencing and Draining Across Series of NoC Elements that Maintain Transaction State

Many NoC configurations create sequences of master NoC elements, each maintaining some transaction state that must be allowed to drain. These must be carefully managed to ensure that deadlocks are not created where one or more of these elements is blocked from completing the draining process by a downstream element that is fencing transactions.

* Most “simple” master elements have blocking behaviors when they execute fencing. E.g., VD crossing FIFO, metabridge,... These elements disable their upstream interfaces to block all newly arriving transactions. They do not selectively pass transactions forward, nor do they complete transactions locally – they do not respond with DECERR or AutoWake requests.
  + All such elements must track outstanding transactions (simple counters should suffice) to be able to confirm draining.
* Sequences of blocking elements must be configured such that only the furthest upstream element executes fencing... this is necessary to prevent deadlock scenarios where a downstream element disables its upstream interface during fencing, preventing pending upstream transactions from completing.
  + NocStudio must recognize these sequences and identify the furthest upstream element in each sequence, connecting pd\_active/fence\_ack\_n signals to these upstream elements and tying them inactive at the downstream elements in the sequence (pd\_active tied to 1, fence\_ack\_n unused).
* ACE Master Bridges and their derivatives have special fencing properties:
  + Their fencing behavior for downstream power domain dependencies is selective and broadly aware of many power domains, and it is non-blocking – either returning DECERR or generating an AutoWake request.
    - Note: AutoWake requests block the interface upon which the stimulating transaction arrived until the PMU responds. Special care must be taken to prevent these from creating deadlock scenarios.
  + However, their fencing behavior for their own power domain is blocking – they disable their host interfaces (drive READY low).
    - For this reason, the pd\_active table that drives selective fencing as part of address table lookup never includes the ACE Master’s own PD, even if there are downstream elements that are in the same PD.
      * For simple fencing sequences where it is easy to recognize which PDs are being fenced upstream, it would be desirable to leave these out of the address table lookup as well. However, this will not be generally be possible for multi-hop fencing cases.
* The non-blocking nature of fencing for downstream power domains at the ACE Master Bridges allows us to break the fencing dependency sequences across master bridges.
  + Upstream “simple” master elements (e.g., VD crossing FIFO) fence for power domains upto and including that of the first ACE Master in a sequence. Fencing for power domains beyond the ACE Master is handled by the ACE Master.
* Multi-hop dependencies: fencing for each hop is done by upstream elements up to and including each ACE Master Bridge. A new fencing dependency sequence begins at each ACE Master Bridge in the chain.

## Clock Gating

NoC elements implement coarse grained clock gating at interface level, where logic associated with each interface will gate itself if it detects that it is idle. Signaling between elements wakes the clock gated interfaces on demand.

Clock gating will also be supported at power domain boundaries, relying on the same power control signaling used for power gating. It is a customer implementation alternative to choose whether to gate logic at the power domain boundary or to power gate it, or both with perhaps different levels of timeout. Implementation of clock gating circuitry at this level is entirely the responsibility of the customer’s design flow (i.e., we don’t insert clock gates at this level in the NoC RTL).

### Changes to Coarse Grained Clock Gating Implementation

Several refinements are made to our implementation of coarse grained clock gating as enumerated below. A high-level motivation behind some of these changes is to make sure that clock gating override functionality can be completely controlled from either regbus register settings or NoC level system\_cg\_or pins (per clock domain). While a combination may be used, complete control is possible with either mechanism.

1. Regbus Ring Master controls are gone: the clock gating control register (RBSLVCG) has been removed from the ring master with associated adjustments to how the related pins on NoC elements are to be driven, described in the following items.
2. system\_clk\_en: this pin is tied high inside NocStudio generated wrappers for NoC elements as this functionality is no longer used. *In the long run, we may want to remove the system\_clk\_en functionality from NoC elements altogether.*
3. system\_cg\_or: depending on state of mesh\_prop coarse\_clock\_gating\_enabled…
   1. false: tied off high inside the NocStudio generated wrapper for each NoC element.
   2. True: a NoC level pin is provided for each clock domain for which there are any NoC elements that expose a system\_cg\_or pin. The NoC pin drives a network connecting to all the system\_cg\_or pins of a given clock domain. Note: for clock domains where there would be zero connections, the NoC pin is omitted. Note changes to pipeline registers and ring master described below.
4. Pipeline registers: these no longer get system\_cg\_or. The override functionality is controlled indirectly via the busy output at the transmitting end of the link. System\_cg\_or pins for pipeline stages should always be tied low by NocStudio. In the long run, these pins should be removed altogether from the pipeline stages.
5. Ring Master: system\_cg\_or should always be tied off low by NocStudio. Course clock gating in ring master should be controlled completely via signaling from the router driving it.

Further refinements, perhaps deferred to a later release:

1. Clock gating registers should be removed from the design and the documentation when mesh\_prop coarse\_clock\_gating\_enabled is set false.

## Preserving/Restoring State

A NoC that does not deploy regbus should not have any state within NoC elements that must be saved and restored through power transitions. *Need to revisit this if this turns out not to be true.*

A NoC that does deploy regbus may use regbus transactions to save/restore state (e.g., address maps), but that process must be managed by customer logic. We provide no extra low-power logic to automate this process. If configuration is required to restore logic to a usable state, reset defaults should leave the logic in a disabled state so that transactions are held off until regbus programming happens, the final steps of which enable the logic for normal functionality.

### Retention Registers and Reset

We may want to consider supporting automatic conversion by synthesis flow of configurable state into retention registers (e.g., programmed address map registers in master bridges). The key HW requirement is that state that has been configured to be retained must not be reset through power state transitions.

All NoC elements receive two reset inputs: a chip reset (asserted for cold start-up, e.g., initial power-up of the SoC or fundmental reset of the device), and a *reset\_pd\_<PD>\_n* that is asserted/de-asserted at power domain boundaries when cycling through a power-down/power-up sequence. State registers identified for retention should only be reset by the chip reset signal. All other state should be reset by both chip reset and *reset\_pd\_<PD>\_n* (the logical AND of these two signals assuming active assuming they are active low). If no retention registers exist within the element, all registers are reset by the combined reset signal.

Additionally, we would need to have hooks in NocStudio for identifying the registers to be retained, and we would need to generate the appropriate information in the CPF/UPF files to drive the synthesis flow to instantiate retention registers where required.

## Regbus

While regbus is not required to support low-power implementation, our low-power implementation must support designs that use it. In some ways it presents unique challenges as it connects NoC elements in a different topology that can potentially create some new interdependencies between customer power domains. In many other ways, the treatment is similar to the way logic on other NoC layers is handled.

A fundamental change is that regbus can be powered down. However, to make doing so practical, some changes are required to regbus implementation.

### Low Power Friendly Regbus Optimization

There are several restrictions in the current regbus implementation that make it more challenging if not impossible in some cases to support low power designs. To address this, some changes to regbus implementation are required. These are described in detail in the following sections, but the high-level summary of the changes is:

1. The regbus ring master will be modified to support multiple slave rings, where elements attached to each given ring share common clock, power and voltage domains. (This is described further in section “1.13.1.1 - Separate Rings for Elements with Different Power and/or Clock Domains” below.)
2. There will no longer be a dedicated regbus clock that must be used by all routers in the regbus layer. The regbus layer in the NoC will be allowed to have multiple clock, power and voltage domains so that each router may be mapped to a clock/power/voltage domain that matches some or all of the elements in the node it serves.

With these changes, all elements of the regbus layer may be mapped into switchable power domains… none are required to be always on.

1. Ring slaves: these share the voltage/power/clock domain of the element they serve.
2. Ring master/Regbus router: within a given node, these elements should share the same voltage/power/clock domains. In cases where a ring master exists in a different node than the regbus router which drives it (e.g., skip router), the ring master may have different voltage/power/clock domains. Voltage/power/clock domain boundaries between routers of the regbus NoC layer are handled in the same manner as other NoC layers, as are interdependencies (i.e., NocStudio would rely on algorithms in the *tune\_power* command to map these). Where a ring master in different node than regbus router has differing voltage/power/clock domains than the router, boundary crossings are handled in the same manner as router/bridge crossings are handled in other NoC layers. *Note: pending updates, ring master does not support an asynchronous interface to the regbus router, therefore NocStudio must not skip the router in the ring master node until ring master is updated. There may be unavoidable corner cases in virtual nodes at the edge of the NoC which NocStudio will need to recognize and flag as invalid configurations.*
3. Boundaries at ring master interface to slave rings:
   1. Voltage/Power/Clock domain boundaries: these are all implemented at the ring master interfaces to the rings it drives. There should be no other boundaries beyond this interface (all logic associated with a particular slave ring must share common voltage/power/clock domain).
   2. CPF/UPF: when the ring master is in a different voltage and/or power domain than one or more slave rings, proper level shifting and/or isolation will need to be inserted on the slave ring interfaces to the ring master.
   3. One slave ring – single power domain: ring master/regbus router may be in the same power domain as the slave ring.
   4. Multiple slave rings – multiple power domains: ring master/regbus router ideally will be mapped to a power domain that remains on whenever any of the slave rings are on to avoid breaking access to slave rings that remain on. However, they could be mapped to the power domain of one of the slave rings, and fencing and draining at the regbus master would need to account for the power domain interdependencies this creates (via parameters provided by NocStudio).
   5. RTL hierarchy: the Voltage/Clock crossing logic at the ring master interface to each slave ring must be implemented in places such that the hierarchy is split where level-shifters may be inserted to support a voltage crossing.
4. Regbus master: may be mapped into a switchable power domain. It would need to remain on to provide regbus access to registers in any domain that is powered on, but it would be reasonable to power it and regbus layer routers down if regbus is not needed for the current operational state.
5. Regbus tunnel: shares common voltage/power/clock domain with regbus master.

#### Separate Rings for Elements with Different Power and/or Clock Domains

1. Current Implementation:
   1. 1 ring master per node in the NoC
   2. 2 rings: one for all the node elements (including ring master), required to be operating on a common noc\_clk, the other ring operating in the regbus\_clk supporting the regbus router.
2. Problems:
   1. Node elements may be in different power domains creating complex interdependencies… powering down one domain may interrupt traffic going to a domain that remains active.
   2. Node elements may have different noc\_clks (this is an issue independent of low power implementation).
3. Solution: create 1 slave ring per power and/or clock domain.  This avoids the need to do clock crossing within the ring, and it allows ring slaves to be safely mapped into the power domain of connected elements without causing unrelated traffic interruptions.  To support this, the ring master will be updated to support up to 7 rings, each ring supporting elements that share common clock/power/voltage domains. Domain crossing is handled at each ring interface.
4. Additional Considerations:
   1. Regbus rings don’t necessarily need to be associated with nodes… we could consider having regbus rings connect elements across multiple nodes where they share common clock, power and voltage domains.  This could lead to some area savings.

#### Support Multiple Clock Domains in Regbus Routing Layer

1. Current implementation: requires common regbus clock for entire regbus layer.
2. Problem: expect this to be challenging if not impossible for some designs, particularly crossing voltage domains.
3. Solution: routers already support clock domain crossings, so just remove restriction that regbus layer must use 1 common clock.
   1. Regbus routers could use a noc\_clk shared with other elements of the node they are serving, which would in turn allow them to live in the same regbus ring as other elements.

#### Other Considerations

1. Low Power V1 Registers in Ring Master: there are several state bits in the current ring master implementation that support the low power V1 implementation, but these are no longer required. They should be removed along with the ring slave that provided regbus access to them. The goal is to save area by eliminating the need for regbus access to state in the ring master.
2. Area Savings: beyond the scope of low power implementation, but perhaps to be considered while planning for implementing low power motivated changes.

### Regbus Master Fencing and Draining

The regbus master is derived from the ACE Master bridge core and is the only master bridge on the regbus layer. It implements selective fencing and draining via address table lookup in a similar manner to AXI4 master bridges, meaning that it fences new transactions targeting registers in a domain that is down, including consideration of any intermediate power domains in the regbus layer (routers and ring masters), and it drains outstanding transactions to such registers. This means that it must monitor *pd\_active\_<PD>* and acknowledge power domain transitions via *fence\_ack\_<PD>\_<master>\_n* only when affected pending transactions have completed. NocStudio will need to provide parameters for each address range handled by the regbus master that indicate the power domains upon which that range depends. It must also assert *autowake\_<PD>\_<master>* or return a decode error when new requests arrive for registers in powered off domains.

One observation is that we might anticipate that the regbus will be idle in many implementations for most of normal operation. In these cases, it would be reasonable to power down the regbus master as well as the entire regbus routing layer even though the rest of the NoC might be powered up and active. Of course this will be implementation dependent.

#### Compacted Regbus Address Map Only

To properly perform fencing and draining, the address table in the regbus master must resolve destinations down to at least the individual slave ring level, as it needs to distinguish between targets that live in different power domains. In non-compacted address mode, the address table resolves only down to the ring master, and further decode to target the correct slave ring is done in the ring master. So non-compacted address mode cannot be used for low deployments.

Therefore, only compacted regbus address mode is supported for low power enabled designs that are also regbus enabled.

### Ring Master and Regbus Routers

Since the regbus master implements fencing and draining that accounts for power domain interdependencies, no additional work needs to be done at the ring master… traffic should already be halted at the regbus master. Therefore the ring masters can be treated like other slave bridges. Ring masters and regbus routers participate in the same *sleep\_req\_<PD>\_n*/*sleep\_ack\_<element>\_n* signaling protocol as other NoC elements.

### Regbus Tunnel

The regbus tunnel implements a slave interface to an AXI4 slave bridge that sits on the NoC, and it bridges transactions to the master interface of the regbus master. It also provides an optional local host master interface, either AXI4 or APB. When the optional local interface is present, the tunnel arbitrates between it and the NoC slave interface for access to the regbus master.

The tunnel is expected to be physically co-located with the AXI4 slave bridge (rbm/s) and the regbus master (rbm/m). Therefore, it always shares voltage/power/clock domain with these elements. The regbus tunnel participates in the same *sleep\_req\_<PD>\_n*/*sleep\_ack\_<element>\_n* signaling protocol as other NoC elements.

* Voltage domain and power domain: all 3 of rbm/s, tunnel & rbm/m share the same power domain, and therefore the same voltage domain.
* Clock domain:
  + host\_clk for rbm/s, rbm/m must be same as tunnel clock
  + noc\_clk for rbm/s may be different
  + noc\_clk for rbm/m may be different

#### Regbus Tunnel Fencing and Draining

Each of the 2 upstream interfaces to the tunnel require special consideration for fencing and draining.

The slv0 NoC facing interface creates a multi-hop dependency in our NoC where the regbus master creates the next hop in the sequence. Fencing and draining through this interface is handled upstream by masters that communicate with the NoC connected AXI4 slave bridge. This interface and the AXI4 slave bridge in front of it must remain active during fencing and draining for their power domain. I.e., slv0 AxREADY must remain active and transactions accepted until *sleep\_req\_<PD>\_n* is asserted. Note that master bridges for which the regbus tunnel is a target only need to implement fencing and draining w/respect to the power domains required for traffic to reach the regbus tunnel. They do not need to consider dependencies beyond the tunnel through the regbus master to the final destination ring slave – that portion of the problem is handled by the regbus master, which will either return a decerr (which would be returned by the tunnel to the originator) or generate an autowake request as required for a particular transaction.

The optional slv1 local port creates a blocking fencing master in front of the regbus master, and it must implement fencing and draining in response to its own power domain. This implies the following when slv1 (either AXI or APB) is active:

* *pd\_active\_<PD>* is monitored where <PD> is the power domain of the all elements of the tunnel complex (rbm/s – AXI4s, rbm\_tunnel, rbm/m – regbus master).
* When pd\_active\_<PD> transitions 1->0, the slv1 interface is disabled at a clean transaction boundary by de-asserting READY.
* Pending transactions are drained to completion before *fence\_ack\_<PD>\_n* is asserted by then tunnel.
  + The slv1 interface must maintain pending transaction counters to support this functionality.

Whether or not the slv1 interface is active in the tunnel, the downstream regbus master (rbm/m) must have the pd\_active input for its own power domain tied to 1 (always active), as it must never fence for its own power domain when rbm\_tunnel is instantiated in front of it. Correspondingly, its *fence\_ack\_<PD>\_n/fence\_done\_<PD>\_n* outputs for its own power domain should be left unused (floating). However, as noted earlier, the regbus master does implement fencing and draining for downstream power domains.

### User Regbus Bridges

User regbus bridges provide a connection between a regbus slave and a host interface, allowing access to host registers over regbus. The HW requirements are the same as for other slave bridges (e.g., AXI4S)… these bridges must participate in sleep\_req\_n/sleep\_ack\_n handshake for their own power domain.

These bridges invoke additional fencing requirements at the regbus master since the hosts that are connected may be in different power domains than the bridge… i.e., these bridges have both *power\_domain* and *power\_domain\_host* properties, and the ring slave (and therefore the regbus ring to which the bridge is connected) lives in *power\_domain*, where the host lives in *power\_domain\_host*, which may be set differently. NocStudio must configure the regbus master address table entries for user regbus bridges such that they include the *power\_domain\_host* of these bridges in the PD dependency list.

## NocStudio Requirements

### Tune\_power Enhancements

This section directly incorporates a description of tune\_power enhancements documented in an email sent by Sailesh on 4/16/15:

#### High Level Enhancements

1. Power profile is no longer necessary for power optimization.  W/o any power profile, power optimization should do a decent job; w/ power profile, the results can be even better.
2. We are now allowed to create power domain dependencies, however power optimization will minimize this.  In v1, such interdependency creation was not allowed as we were not allowed to modify power profiles.
3. We are adding props to quantify the host power consumption, so we can compare cost of creating various kinds of interdependencies, and decide between keeping various hosts on vs keeping various routers on.
4. We will continue to use SA but will include bridge and host power consumptions in the cost equation (this is imp due to interdependencies that are now allowed).
5. We are making power domains physically aware - so each power domains will be allowed only in certain parts of the grid (overlap is allowed; w/o overlap we can't do any optimization).
6. We are removing auto power domain creation; we will expect all power domains to be created by user if low power is enabled.  We are also removing default system power domain.
7. Significant improvements in the way wildcarded power domains in power profile are handled during optimization - this is also key to optimizing power domain for routers when no power profile is given.
8. TBD: one possibility the new scheme opens up is we can now automatically assign power domains to bridges and hosts (in addition to routers) so that the total system power including hosts power is minimized; in this mode of optimization, the power profile given by user cannot be a collection of power domains but rather must be collection of hosts/bridges and possibly traffic.  This will be an interesting capability which can be very valuable during the exploration phase of the SoC dev.  Pls. share your thoughts on this one.

#### General Changes

1. mesh\_prop *max\_new\_power\_domain* goes away.  Instead all pds are now user created.
2. mesh\_prop  *allow\_always\_on* stays and used in the same way.
3. *allow\_non\_adj\_power\_domain* goes away.  Instead we use position of power domains for pd choices.
4. Add a new mesh\_prop *can\_create\_host\_pd\_inter\_dep*.  Based on this NS will be allowed to create host (not bridge) pd interdependency.  If false then tune\_power may fail to assign pd to certain routers.
5. Add a bridge prop *buffer\_cost\_host* associated with bridge prop *power\_domain\_host*.  This will indicate the cost of powering on the host of a bridge; must be in same units as routers and bridges, i.e. rough flop count of the host.
6. Can assign one or more pds to a Grid position, by def all pds are assigned at all Grid positions.  Overlap is fine.
   * a bridge pd is available for routers at that position and its immediate neighboring positions.  Imp for bridges connected to routers direction ports or j,k,l ports.
7. Remove auto creation of new pds; user must create them manually.
8. Bridge pd and its host pd are same (by default).  User can however put bridge in diff pd than host.
9. Add a mesh\_prop *low\_power\_enabled*.
   * If false, hide pd and vd props; tune/analyze\_power won’t do anything; gen\_ip will ignore pd checks and power ip generation.
10. Remove system pd and vd and add unassigned pd and vd (if not already present).
    * upon add bridge will have the first pd from the pd choices at the bridge position. If no pd then pd remains unassigned.
    * upon move bridge pd does not change.
    * upon mapping router will have the first pd from the pd choices at the bridge position. If no pd then pd remains unassigned.
    * bridges cannot have unassigned pd for map and tune/analyze\_power to work (if *low\_power\_enabled* is true).
    * all NoC elements must have non unassigned pd for gen\_ip (if *low\_power\_enabled* is true).
11. tune\_power will always reassign pd to all routers; or pd assignment can be done manually.
12. More per pd props (will be added later).

#### Tune\_power Optimization

##### When Power Profiles Are Present:

1. Add a small cost for every pd crossing.
2. Add hosts (buffer\_cost\_host) and bridges cost based on interdep into the cost equation.
3. Allow interdep to be created (i.e. host/bridge pds can be added to pp).
4. Only use one of the allowed pds at a position for the routers (#6 above).
5. Change the way { } in pp is handled.
   * Compute cost like w/o wildcard but divide the cost contribution of a bridge w/ wildcarded by half (as they are on/off 50:50).
   * If we use a wildcard pd for a router and at least one other pd of pp traffic go through the router, then
   * increase the cost by sum(B(pd))/2, for all bridges of pd whose traffic go through router in this pp

##### When No Power Profile Is Present

1. Assume a pp {all pds}.  {} treatment above will automatically optimize.

### Generate and Communicate Power Domain Information to Master Bridges

NocStudio must provide information to each master bridge, including regbus master, to support fencing and draining. For each slave that is reachable by the master, a list must be provided (format TBD) of power domains that must be active for communication to succeed.

### Generation of NetSpeed Power Supervisor Module

NocStudio must model the NetSpeed Power Supervisor Module as a new NoC element that the user places within the grid. NocStudio will be responsible for generation of the RTL for this module. In addition to instantiating a copy of the Power Domain FSM for each power domain, there may be logic that checks power domain interdependencies which would be used by the FSMs to determine whether or not it is safe to accept a new power down request from the PMU (see section “1.14.7 - Power Domain Interdependency Tracking”).

Some, maybe most of the logic within the NSPS, may be physically distributed throughout the NoC. The combinatorial gates used to aggregate the signaling from NoC elements to the FSMs and QACTIVE may be placed near the sources they are combining to reduce wiring, and the FSMs themselves can be mapped into the power domains they are controlling. Therefore, the NSPS RTL may need to be distributed at the NoC top-level. We may want to provide some options for how this is done.

### Wiring between NSPS and NoC Elements

NocStudio already contains facilities for generating HFNs and combinatorial logic trees (AND and OR functions) that should support most if not all the wiring requirements between NSPS and NoC elements. The logic trees are physically aware, but this might need to be enhanced with awareness of power domain boundaries.

Signals traveling from NSPS to NoC elements are point-to-multipoint, implemented as high fanout nets. NocStudio will need to use understanding of power domain mappings to determine which NoC elements must receive a given signal. For example, for *pd\_active\_<PD>*, only master bridges that have targets dependent upon the indicated power domain need to receive this signal.

Signals traveling from NoC elements to NSPS will be combinatorially aggregated on the way. In some cases, 2 different aggregation functions are required (both AND and OR). See section 1.7.1 for details.

Link availability signals must be wired between NoC elements where interfaces cross power domains. See section 1.8.

#### Explicit Enumeration of Genrtl HFNet Requirements

Pulled from a summary Eric extracted and refined 4/19/15…

HFNet is a N:1 or 1:N communication network, with aggregation of values or broadcast of a value.

Root: this is the "1" side of the network; this generally identifies the network

Leaves: these form the "N" side of the network

Operation: for aggregation: and / or, for broadcast: "broadcast"

======================

Root: fence\_ack\_asserted\_<PD>\_n on NSPS

Leaves: fence\_ack\_<PD>\_<master>\_n pins on all master bridges (technically, only some PDs may be connected to a given master)

Operation: Or

======================

Root: fence\_ack\_deasserted\_<PD>\_n on NSPS

Leaves: fence\_done\_<PD>\_<master>\_n pins on all master bridges (technically, only some PDs may be connected to a given master)

Operation: Or

======================

Root: sleep\_ack\_asserted\_<PD>\_n on NSPS

Leaves: sleep\_ack\_<PD>\_<masterelement>\_n pins on all master bridges elements within the power domain

Operation: Or

======================

Root: sleep\_ack\_deasserted\_<PD> on NSPS

Leaves: sleep\_ack\_<PD>\_<masterelement>\_n pins on all master bridges elements within the power domain

Operation: And

======================

Root: wake\_req\_asserted\_<PD> on NSPS

Leaves: autowake\_<PD>\_<master> pins on all master bridges (only some PDs may be connected to a given master) and wake\_req\_<element>\_<ifce> pins on noc elements associated with driving a link into the PD

Operation: Or

======================

Root: idle\_status\_<PD>\_n on NSPS

Leaves: idle\_status\_<element>\_n pins on all noc elements in PD

Operation: Or

======================

Root: sleep\_req\_<PD>\_n on NSPS

Leaves: sleep\_req\_n on all noc elements in PD

Operation: broadcast

======================

Root: pd\_active on NSPS <PD>

Leaves: pd\_active\_<PD> on masters (only some PDs may be connected to a given master)

Operation: broadcast

### Regbus Optimization

See section 1.13, specifically section 1.13.1.

### Power Aware Routing

It may be interesting to explore optimizations to the routing algorithms that take account of power domains to minimize interdependencies.

### Power Domain Interdependency Tracking

Routers in the NoC may have traffic flows passing through them that are sourced from and sent to power domains that are different from their own power domain. In such cases, choosing to bring down the domain that includes the router may have the unintended side effect of interrupting traffic that is flowing between still active power domains. NocStudio knows about all traffic flows and their dependency on power domains, so it can export this information in a format that can be used to detect these interdependencies, and if desired, to adapt power control decisions to avoid such side effects. See section “1.9.2 – Power Domain Interdependencies – TBD Interlock.”

It is proposed that NocStudio create a table of bit vectors that capture this information. These bit vectors can be used by logic in the NSPS to detect when it is safe to bring a power domain down and when a power domain should be woken up to support traffic flows involving other power domains that are being woken up.

#### Dependency Bit Vectors

A set of static dependency vectors should be generated from traffic and power domain information in the NoC configuration via the following steps.

1. Create per traffic flow vector pairs: generate a pair of related bit vectors where each bit corresponds to a power domain.
   1. Full dependency chain: set bit position to 1 for each power domain that is part of the traffic path from source through destination.
   2. Through dependency chain: subset of above vector that only sets bits for power domains that are neither source nor destination.
2. Merge and Purge: execute the following steps to compress the data into a minimum set of vectors.
   1. Merge: for all identical full dependency chains, OR together all of their related through dependency chains to create 1 merged through dependency chain associated with that full dependency chain pattern.
   2. Purge: throw away any vector pairs for which no bits are set in the through dependency chain vector.

The result is a set of static vectors that can be used by logic to determine in conjunction with power state of all other power domains when it is safe to bring down a particular power domain without side effects, and when a domain should be woken up to allow traffic flow.

#### Power Domain Status

To use the dependency vectors to drive power control decisions, we need a dynamic vector that represents power domain status. This is similar to *pd\_active\_<PD>*, but it needs to differ slightly as we want to know when a domain has a pending wake request, not just when it is already awake.

A new internally consumed signal, *pd\_status\_<PD>*, can be generated from the Power Domain FSM that asserts early relative to *pd\_active\_<PD>*, back in the WAKE\_ACK or SLEEP\_NACK state, but like *pd\_active\_<PD>*, it de-asserts upon entry to the FENCE\_DRAIN state.

The *pd\_status* vector is used as described below to drive sleep and wake decisions.

#### Safe to Sleep

1. AND each full dependency chain vector with the current *pd\_status* vector and bitwise OR the result to create a valid bit.
2. Mask each through dependency vector with the associated valid bit created in step 1.
3. OR together all the masked dependency vectors and invert (NOR).
4. *safe\_to\_sleep* vector: A 1’b1 in the result indicates it is safe to power down the associated power domain (i.e., there are no potential through traffic flows that are alive given current power domain status).

#### Wake Request

1. Invert both *safe\_to\_sleep* and *pd\_status* vectors, AND them together (NOR)
2. *pd\_status\_wake\_req* vector: Each 1’b1 bit in the result indicates a power domain that should be woken up to avoid potential blockage of through traffic paths.

#### Using the Results

There are some implementation options to consider.

1. Export static dependency vectors in a format that customers could integrate and use in their own PMU logic.
2. Implement logic to generate *safe\_to\_sleep* and *pd\_status\_wake\_req* vectors, use them in one or more of the following manners…
   1. Bring *safe\_to\_sleep* and *pd\_status\_wake\_req* vectors to top level pins for customers to use directly.
   2. Incorporate *safe\_to\_sleep\_<PD>* in generation of *QACTIVE\_<PD>* to PMU.
      1. This is currently our only relatively direct means of communicating idle status to the PMU.
   3. Incorporate *pd\_status\_wake\_req\_<PD>* into *wake\_req\_asserted\_<PD>* to cause Power Domain FSM to generate wake requests (assert *QACTIVE\_<PD>*) for interdependent power domains.
   4. Implement logic that directly controls the PMU interface for power domains that are purely internal to the NoC… could completely absolve customer PMU from having to manage these power domains, though that would also mean we are generating all power control signals (e.g., clock enables, isolation enables, power gate enables) from our own logic for these domains. *May not want to go here.*

It is proposed to pursue 2(b) and 2(c), but not in phase I of low power deployment.

### Low Power Related Properties

#### Power Domain Properties

Power domains have the following properties associated with them:

1. *QACTIVE\_<PD>* related:
   1. *pd\_qactive\_present* – default = on/true: when true, *QACTIVE\_<PD>* is implemented as part of the Q-channel interface, otherwise, it is left out.
   2. *pd\_autowake\_enable* – default = off/false: indicates whether power domain supports autowake or not. See section 1.15.1 - Controlling Fence Response: Decerr vs. Autowake for description of how NocStudio uses this property to generate parameter values for master bridges. Also determines whether or not *wake\_req\_asserted\_<PD>* is included as part of *QACTIVE\_<PD>*.
   3. *pd\_idle\_status\_enable* – default = on/true: determines whether or not *idle\_status\_<PD>\_n* is included as part of *QACTIVE\_<PD>*.
2. *QDENY\_<PD>* related:
   1. *pd\_qdeny\_timeout\_enable* – default = on/true: indicates whether a timer is implemented in the PD FSM that will assert *QDENY\_<PD>* to abort a power down sequence that fails to complete within the time allotted by the *pd\_qdeny\_timeout\_value* property below.
   2. *pd\_qdeny\_timeout\_value* – default = 0xffff: 16-bit timeout threshold for *QDENY\_<PD>* timer.
3. *pd\_merged\_reset\_enable* – default = on/true: indicates that NocStudio should only bring out the *reset\_<PD>\_n* pin at the NoC interface and connect it to both *reset\_n* and *reset\_pd\_n* pins of NoC elements. This would be typically be done for power domains that have no retention structures that maintain state through power cycles.
4. *pd\_isolation\_enable* – default = on/true: indicates whether or not power intent file (CPF/UPF) should insert isolation clamps for all outputs of the power domain.

#### Bridge Properties

Bridges have the following properties associated with them:

1. *XXX\_autowake\_enable* – default = off/false: indicates whether a master bridge that is fencing will generate autowake or decerr for fenced traffic. If off, the master bridge will return decerr for all fenced traffic, regardless of power domain support for autowake. If enabled, bridge will use info about power domain support for autowake from address table look up to determine if a given fenced transaction should generate autowake or return decerr. This property applies only to master bridges, where “XXX” indicates master bridge type (e.g., “axi4m” for AXI 4 master bridge).

#### Mesh Properties

The following global low power properties are defined:

1. require\_autowake\_pd\_for\_routers – default = on/true: controls legal power domain choices for routers during default power domain assignment post mapping and during tune\_power. When on, NocStudio will not allow routers to be assigned to non-host power domains that have their pd\_autowake\_enable property set to off.  Otherwise, such assignment is allowed. Note: host power domains are allowed regardless of pd\_autowake\_enable setting as long as traffic dependencies ensure no traffic could be flowing when that host power domain is down.

## Master Bridge request Fencing

FIG below shows the interfaces added to master bridge for low power functionality.



Figure 7 Master Bridge low power interfaces

When a PD needs to be power gated, any requests to that PD must be fenced at sources on the NoC, this is the master bridge in case of AMBA NoC. Requests and corresponding responses from/to a master bridge can traverse multiple power domains on the NoC. Even routes between the same pair of master and slave bridges can flow through different set of PDs depending on the QoS of the flow.

FIG shows schematic of implementing fencing in a master bridge. The *pd\_active* bit vector from the NSPS gives information of power domains to which requests must be fenced (any bit 1’b0 indicates that PD is gated or is in the process of being gated). Fencing operation is performed after address look up. Locally unique SlaveID generated by address lookup is used to index into another constant parameter table programmed by NocStudio. This table produces a bit vector of system power domains with bits set for the superset of power domains through which requests and responses are exchanged between the master and current request’s slave. A comparison is made between this vector and *pd\_active* bit vector to check if all power domains needed for the flow are active. If this check fails, then one or more of power domains required for the flow is gated and hence the request should not be sent on to the NoC. There are two options to be pursued in this case. First option is to accept the command and respond with decerr. It may be useful to log the cause of decerr as ‘unavailable PD’ in a status register. Second option is to stall the interface and raise a request to NSPS to wake up the power domains required for the current request to make progress. Note that, PD wakeup request stalls the entire AR or AW interface.



Figure 8 Master bridge request fencing

### Controlling Fence Response: Decerr vs. Autowake

The decision about how to respond for a fenced transaction is controlled as a property of the master bridge and by properties of the power domains upon which the fenced transaction depends. The basic function combining these two sources of control is AND – autowake will only be asserted when both the bridge control and the power domain controls agree that autowake is enabled, otherwise the transaction will be terminated with a decerr.

The bridge property is *XXX\_autowake\_enable* (see section 1.14.8.2 #1), which is used by NocStudio to set the value of the master bridge parameter P\_MST\_AUTOWAKE\_ENB. When regbus is implemented, a programmable register is provided that defaults to the value of the parameter, otherwise the parameter is use statically. The property/parameter applies to all transactions, regardless of address range.

The power domain control is implemented per address range as part of the address table lookup. NocStudio communicates this information to the master bridge via the following parameters (1 bit for each address table entry):

parameter [P\_RD\_NUM\_PD\_LIST-1:0] P\_RD\_PD\_AUTOWAKE\_ENB;

parameter [P\_WR\_NUM\_PD\_LIST-1:0] P\_WR\_PD\_AUTOWAKE\_ENB;

Each power domain has a property, *pd\_autowake\_enable* (see section 1.14.8.1 #1), that indicates whether or not autowake is supported by that power domain. For each bit the parameter lists above, NocStudio ANDs the *pd\_autowake\_enable* settings for all power domains upon which the associated address range depends (except for the master bridge’s power domain if it is different from the others) to generate the bit’s value. In other words, an address range will support autowake only if all the power domains upon which it depends support autowake, otherwise it will return decerr, regardless of the bridge level control. Note that decerr will be returned even when only one of multiple power domains does not support autowake, even if that domain is powered up and only domains that support autowake are powered down blocking the traffic.

Pseudo-code for calculating the final decerr vs. autowake decision is:

do\_autowake = PD\_MST\_AUTOWAKE\_ENB & PD\_<RD|WR>\_PD\_AUTOWAKE\_ENB[ADDR];

### Fencing Atomicity

Our master bridges implement a feature where transactions above a configurable size threshold will be split into multiple smaller transactions before being forwarded through the NoC. There are corner cases where the *pd\_active* vector changes state while the master bridge is processing a large transaction, where 1 or more of the sub-transactions have already been injected into the NoC, but more are pending. In the case of writes, even non-split transactions require multiple cycles to be absorbed into the bridge, and there are corner cases to consider where the *pd\_active* vector changes while data is still arriving.

In all such cases, the master bridge must take care to make sure that fencing is done on an atomic boundary with respect to the transaction as it appears on the host interface. Once any part of a transaction has been accepted for forwarding into the NoC, all parts of the transaction should be allowed to complete normally (i.e., drain). This will be supported by snapshotting the *pd\_active* vector in the master bridge on host transaction boundaries.

#### Fencing Atomicity Exception – Multiple Targets with Differing Routes

The master bridges cannot guarantee atomicity for cases where the host transaction is split such that the sub-transactions hit 2 or more address lookup table entries. Each address table entry can specify a different routing path and possibly different target. If the set of power domains involved differs for sub-transactions, they may have differing fencing requirements, but this is only discovered at the time of address table lookup for each sub-transaction. If the first sub-transaction is fenced, then all sub-transactions will be fenced with the same behavior (DECERR vs. autowake). If the first sub-transaction is allowed to proceed, and a later sub-transaction is fenced, the master bridge will terminate the entire transaction with a SLVERR.

A NocStudio enhancement to use memgroups to force common fencing treatment across multiple targets will largely address this exposure. In this case, the user identifies a set of slaves that should be treated as a unit for address table lookup decisions, including fencing. A common conservative set of power domain parameters would apply to all address table entries for the slaves in the memgroup such that all sub-transactions split across this memgroup would be fenced atomically.

## Master Bridge Response draining

In addition to fencing requests to power gated domains, master bridge is also required to drain responses for all outstanding requests to any power domain which needs to be gated. Before a PD can be gated by the NSPS, all masters in the system must confirm that responses for all outstanding requests to that PD have been completed. This functionality is also implemented in the master bridge using the AID tracking tables. FIG below shows the schematic of the draining functionality.



Figure 9 Master bridge response draining

AID table is a register based structure and hence it is possible to do combinatorial decode status of requests outstanding to different slaves and PDs associated with them.

Each entry in the AID table corresponds to an outstanding request and stores the locally unique ID of slave to which the request is outstanding. A valid bit indicates if the entry is outstanding or unused. SlaveID field in each AID table entry is used to index a constant parameter table created by NocStudio. This table produces a bit vector of system power domains with bits set for the superset of power domains through which responses from the entry’s slave return to the master. This vector is gated and qualified with the valid bit of the AID table entry. Bit wise ORing of active power domain vectors generated by each AID table entry produces a bit vector indicating power domains to which responses are outstanding in the AID table. This vector is then bitwise ORed with *pd\_active* bit vector to generate *fence\_ack\_<PD>\_<master>\_n* handshake signal in response to a power gating request for a PD from the NSPS.

## Streaming Bridge

### Streaming Bridge Fencing

A streaming bridge implements fencing in a similar manner as an AMBA bridge by using the destination ID of incoming transactions (transmit direction) to look up power domain dependency from a parameter table provided by NocStudio and comparing that against power domain state indicated by the *pd\_active* vector. Any transaction that has a dependency upon a power domain that is indicated to be down by the *pd\_active* vector will not be transmitted, meaning it will remain in the input FIFO and progress on the interface will ultimately stall as credits are exhausted. This must be done at clean packet boundaries, so once packet transmission has started, it will be completed regardless of the indicated power domain state (subject of course to normal back pressure from the NoC, which could possibly include *link\_available* signaling along the path).

Presence of a fenced packet in the input FIFO of a streaming bridge transmit interface would generate an autowake request for the dependent power domains if that option is enabled.

#### Streaming Error

As currently defined, there is no error/nack option for the streaming interface. If we want to support the equivalent of the Decerr option that exists for AMBA interfaces (i.e., so fencing is not blocking traffic to unfenced destinations), we would need to define an error/nack mechanism and signaling to support it. *We need to decide if we need this.*

### Streaming Bridge Draining

The streaming protocol is fundamentally point-to-point with no acknowledgement that a packet has reached its destination. So on the surface there is little to be done to confirm draining from the master bridge other than to make sure all flits of a packet in flight have been transmitted.

To account for “typical” latency of packets propagating through the network, we deploy a delay timer in the Power Domain FSM SLEEP\_WAIT state to wait some amount of time after *fence\_ack\_asserted\_<PD>\_n* has gone active before advancing from the FENCE\_DRAIN to the SLEEP\_REQ state.

#### Streaming Bridge Response Pending

We also need to consider that customers may implement a request/response protocol at the host level on top of our streaming protocol, where the transmitting host is expecting to receive an acknowledgement returning on a receive interface. We may want to consider modifying the streaming bridge to add an input vector to the host transmit interface with one bit per legal destination ID that indicates that the host has initiated transactions that have pending responses. This vector would be incorporated into the fencing/draining logic to hold off *fence\_ack\_asserted\_<PD>\_n*  where a response is pending from a destination that is affected by a power domain transition. One common vector could support all transmit interfaces. *Need to decide if we need this.*

## Compound Bridges

Compound bridges implement a range of AMBA protocols (and in the future other protocols, e.g., IMGBus4, IMGBus2, OCP,…) by combining some converter logic with our AXI4 master and slave bridges. A current list of compound bridges follows:

Masters:

* AHB-lite

Slaves:

* AHB-lite
* APB
* AXI3
* AXI4-lite
* IMGBus2

Note the AXI and ACE master bridge variants are implemented via parameter and wrapper I/O options on the baseline ACE master bridge, so they are not included in this list.

The compound bridges require some special consideration due to the need to implement voltage/power/clock domain boundaries between their converter component and the AXI bridge with which it is paired.

In general, RTL implementation for these bridges must be modified to allow genrtl to separately instantiate the converter and the AXI bridge portions in ns\_fabric, inserting voltage crossing logic between these instances when necessary. Since the converter portion exists as a separate instance that can live in a different power domain than the AXI bridge to which it is connected, it must communicate with the NSPS via the following signals:

* *sleep\_req\_<PD>\_n*
* *sleep\_ack\_<element>\_n*
* *idle\_status\_<element>\_n*

For all cases, even though it would be possible to support more options for some of them, we are restricting power domain mapping of the two halves to the following:

* The converter logic must always be in the same voltage/power/clock domain as the host.
* The AXI master/slave bridge may be in a different voltage/power/clock domain as the host and converter.

Specific requirements are described further in the sections below.

### AHB-lite Master Bridge

The AHB-lite protocol implements pipelined address and data phases, where the address phase of a new transaction can overlap the data phase of the prior transaction. The result is that a protocol compliant slave must always accept the address phase of a new transaction – there is no way for the slave to idle/stall the bus cleanly between transactions. Therefore, this is not possible to cleanly power down the slave side independently from the master side. For this reason, we specify the following restrictions to low power usage of our AHB-lite master bridge implementation:

1. The AHB-lite converter logic must always share the same power domain as the host logic that drives it.
2. The PMU must guarantee that the host only drives IDLE transfers any time Q-channel for the host/AHB-lite converter power domain is not in the Q\_RUN state (this restriction does not apply to the AXI4 master bridge’s PD when it differs from the host’s). If this latter restriction is violated, we should still behave in a sane manner…
   1. We should detect and log an error in a status register in the AHB-lite converter and raise an interrupt if any non-IDLE transfers occur when *pd\_active\_<host\_PD>* is low.
   2. We should fence any non-IDLE transfers.

The downstream AXI4 master bridge driven by the AHB-lite converter can be in a different voltage/power/clock domain, so we need to handle independent power down of the two sides of the compound bridge. A voltage domain boundary between converter and AXI master bridge is supported by inserting a VD crossing FIFO structure between the two.

#### AHB-lite Converter (ahb2axi): Fencing and Draining

The AHB-lite converter is a master element that must monitor pd\_active bits to implement fencing and draining. It behaves in some ways like a “simple” master as described in section 1.10.2 in the sense that it does not do selective address based fencing – all transactions are fenced regardless of destination. However, unlike other “simple” masters, its fencing behavior is not blocking – it terminates fenced transactions with DECERR (or OK for posted writes).

The non-blocking nature of the AHB-lite converter would allow it to be used to break up a fencing sequence in a similar manner as the ACE master bridge does. However, it is currently anticipated that only host logic will be connected to the AHB-lite host interface, so this quality will likely never be exploited. The AHB-lite converter is expected to be the furthest upstream master in the series formed with the AXI4 master bridge and possible VD crossing FIFO, so it fences on behalf of these elements for their power domains.

The specific fencing and draining requirements of the AHB-lite converter are:

1. Track transaction state: must maintain outstanding transaction counters to confirm completion of all pending transactions to implement draining.
2. Fencing Behavior: transaction is locally completed with an ERROR response, with the exception that posted writes always return an immediate OK response, but they are locally dropped (not forwarded downstream).
   1. A maskable interrupt should be generated for dropped posted writes. A similar interrupt should be considered for non-fenced posted writes that eventually return an ERR response from the AXI4 master.
3. pd\_active: monitor and implement fencing and draining for all transactions at clean transaction boundary.
   1. AHB-lite (host) PD
   2. AXI4 master bridge PD (when different from above)
4. Downstream AXI4 master bridge and possibly VD crossing FIFO – fencing handled by AHB-lite converter…
   1. AXI4 master bridge: *pd\_active\_<PD\_mstrbrdg>* input is tied to 1’b1 by NocStudio, and corresponding *fence\_ack\_<PD\_mstrbrdg>\_n* output is unconnected.
   2. VD crossing FIFO (if present): sleep\_req\_n/sleep\_ack\_n are connected instead of pd\_active/fence\_ack\_n to the relevant I/Os.

In summary, the AHB-lite converter (ahb2axi) acts as a “simple” master as the furthest upstream element in series with AXI4 master bridge and possible VD crossing FIFO. It fences and drains for its own PD, and when AXI4 master bridge is in a different PD, it fences for that PD as well – it will be connected to 1 or 2 pd\_active inputs.

#### AHB-lite Converter (ahb2axi): Sleep\_req\_n/Sleep\_ack\_n

The AHB-lite converter participates in the sleep\_req\_n/sleep\_ack\_n handshake like other NoC elements. As with other elements, it uses these signals to force its interfaces to conform with idle==reset==clamp==1’b0. For example, AxREADY to AXI4 master bridge is de-asserted in response to sleep\_req\_n.

However, as noted above, AHB-lite protocol prevents the converter from stalling its host interface. HREADY will remain asserted high even after sleep\_ack\_n is asserted.

While our spec requires the PMU to ensure the host does not initiate new transactions while not in the Q\_RUN state (must only present IDLE transfers), the AHB-lite converter must be designed to handle a violation of this restriction as gracefully as possible.

1. sleep\_ack\_n should not be asserted as long as the host interface remains in a non-IDLE condition. However, once it has been asserted, it should remain asserted until sleep\_req\_n is de-asserted to preserve the 4-phase handshake.
2. Non-IDLE transfers while *pd\_active\_<ahb-lite>* is low should generate a maskable interrupt.

#### AHB-lite Converter (ahb2axi): Other LP considerations

The AHB-lite converter generates an *idle\_status\_<element>\_n* signal that is aggregated as part of QACTIVE generation.

### Compound Slave Bridges

As with the AHB-lite master compound bridge, the various slave compound bridges support a voltage/power/clock domain boundary at the interface between the AXI slave bridge and the converter logic. Unlike the AHB-lite master, a power domain boundary could be supported with these bridges at the host interface to the converter logic since the bridge is mastering this interface. However, as currently implemented, all the host interfaces only support synchronous operation, with a clock domain crossing supported only at the AXI slave bridge. Supporting 2 options for where a power domain interface is implemented would also require changes to NocStudio to provide this flexibility. Therefore, we will currently implement the restriction that the converter portion of the compound bridge must share the same power domain as the host logic.

#### Powering Down the Slave Converter

As enumerated above, the slave converter must be able to respond to *sleep\_req\_<PD>\_n* with a *sleep\_ack\_<element>\_n* after it has ensured that it is truly idle. This seems perhaps unnecessary as this is slave logic and fencing and draining should be engaged to ensure there are no outstanding transactions pending. However, for completeness, particularly when the AXI slave bridge is in a different power domain, we should support the sleep\_ack/sleep\_req handshake. Since the slave converter does not necessarily have visibility into all pending transactions (i.e., some may be queued in the AXI slave), we need to add a sideband signal driven from the AXI slave to the converter logic to indicate when there are pending transactions (*TBD: does VALID on the AXI interface suffice?)*. While this is asserted, the converter should hold off responding with *sleep\_ack\_<element>\_n*. It should also factor this sideband signal into generation of *idle\_status\_<element>\_n*. The converter must also drive a *link\_available\_<element>\_<ifce>* to the AXI slave to explicitly indicate when the converter is available (distinct from READY based flow control). Finally, the converter should deassert READY back to the AXI slave bridge, which is the correct power clamped state even if it is otherwise functionally unnecessary as there are no pending transactions to arrive on that interface.

While fencing and draining should be preventing new transactions from arriving that target the powered down slave endpoints, should a new transaction arrive, the AXI slave bridge should be designed to respond in one of two ways:

1. Return SLVERR response to all newly incoming transactions.
2. Stall newly incoming transaction while asserting *autowake\_<PD>\_<slave>* for the power domain of the slave converter.

The latter option should be parameterized in NocStudio so that the autowake\_<PD>\_<slave> is only connected and used if specified in the configuration. By default, the AXI slave bridge should immediately respond with an error. If regbus is deployed and use of autowake is specified, a register should be provided to select between autowake and error response.

#### Powering Down the AXI Slave Bridge

No special action beyond normal communication with NSPS (sleep\_req/sleep\_ack) is required when the AXI slave bridge is powered down independently of the slave converter as the AXI slave will only assert *sleep\_ack\_<element>\_n* when all pending transactions have completed. READY should be deasserted to the slave converter.

#### Slave Converter and AXI Slave Bridge Share Same Power Domain

In this case, the sleep\_req/sleep\_ack/idle\_status signals from the converter are redundant with those of the AXI slave, and they can be tied off/left unconnected.

## Other Bridges and Agents

### IMGBus4 Master

This bridge is implemented as a parameterized/generated permutation of the ACE Master core bridge, and as such it is expected to match the low power behavior of other such master bridges (e.g., AXI4M).

### IMGBus2 Slave

This bridge is implemented as a compound bridge in a similar manner to the AHB Lite slave bridge (AHBLS), with an AXI4 slave coupled with an IMGBus2 protocol converter. As such, it is expected to match the low power behavior of AHB Lite slave bridge, including supporting a voltage domain boundary with a VD crossing FIFO inserted between AXI4S bridge and IMGBus2 converter.

### Regbus Tunnel

See section 1.13.4 - Regbus Tunnel.

### Metabridge

The metabridge is an AXI port aggregator that sits in front of an AXI4 master bridge. Two or more AXI host interfaces (AXI3, AXI4-lite or AXI4) are multiplexed into our AXI4 master bridge. Since the metabridge has some transaction state storage, it creates a serial fencing & draining dependency with the AXI4 master it feeds, and it must support “simple” fencing and draining for its interfaces. This implies that it must keep track of outstanding transactions (e.g., via transaction counters) for each of its host facing interfaces.

#### Metabridge Basic Low Power Requirements

These requirements cover configurations where no additional logic sits between user hosts and metabridge host interfaces (e.g., VD crossing FIFO, AHB converter, AXI pipeline).

* Metabridge is assumed to operate entirely within a single power domain, though it may support multiple clock domains.
* Downstream AXI4 bridge may be in a different power domain, though this may not be a common configuration.
  + A voltage domain crossing between metabridge and downstream AXI4 master may be supported by instantiating an AXI4 master VD crossing FIFO between these 2 elements. In this case metabridge and AXI4 master must be in different power domains.
* Metabridge must monitor *pd\_active\_<PD>* for its own power domain and implement fencing and draining for that power domain.
  + Fencing: at clean transaction boundary, AxREADY is driven low. No new transactions are accepted.
  + Draining: all pending transactions for all interfaces must complete prior to asserting *fence\_ack\_<PD>\_n*.
* When AXI4 master is in different power domain: metabridge must monitor *pd\_active\_<PD\_mstrbrdg>* in addition to its own PD, and it must perform the same fencing and draining steps described above.
* Downstream AXI4 master bridge and optional VD FIFO: NocStudio should tie off pd\_active inputs such that they do not fence for their own power domains, and their fence\_ack\_n outputs for their own power domains should be left unconnected. Metabridge is responsible for fencing.
  + Downstream AXI4 master continues to do selective fencing based on address lookup.
* *Sleep\_req\_<PD>\_n/sleep\_ack\_<PD>\_n*: metabridge uses participates in this handshake for its own PD and executes standard process of making sure it is “electrically idle,” specifically ensuring all its interfaces conform to reset==idle==clamp==1’b0.

#### Metabridge Advanced Low Power Requirements

Fencing and draining requirements become more complex if we must support configurations where NetSpeed logic is instantiated upstream of one or more of the metabridge host ports. For example, if an AXI pipeline or a VD crossing FIFO is implemented in front of one of the metabridge host ports, that upstream element must implement fencing and draining for the metabridge and AXI4 master bridge power domains, and fencing must be disabled within the metabridge for the affected interface. If configurations are supported where some, but not all of the metabridge host ports have fencing elements in front of them, then the metabridge must be able to selectively disable fencing for some of its interfaces. This could be configured via a NocStudio provided parameter.

### Reorder Bridge

The reorder bridge is a new agent with a slave bridge interface on the NoC that forwards transactions back onto the NoC via a master bridge. Upstream masters send traffic to the slave interface, and the traffic is forwarded through the master interface to its ultimate destinations. This creates a multi-hop dependency, and it is handled as described in section 1.10.2 - Fencing and Draining Across Series of NoC Elements that Maintain Transaction State.

* Power domains of the master and slave bridge elements of the reorder bridge: there is no architectural requirement for these to be the same, and a voltage boundary could be supported between them with an appropriate crossing structure.
  + However, there is no compelling reason to support a power domain boundary between the master and slave elements, so we will not allow this. NocStudio will enforce the requirement that master and slave elements belong to the same PD.
* Fencing/Draining: upstream masters implement fencing and draining for traffic flowing between them and the reorder bridge, including the power domain(s) of both master and slave elements of the reorder bridge.
  + The reorder master bridge does not fence for its own power domain, but it does implement selective fencing for downstream power domains.
  + If a VD crossing FIFO is implemented between reorder master and slave, it also does not fence, as that is covered by upstream masters.

## Multi-Voltage Microarchitecture

### Voltage Domain Crossing FIFO

One key structure to be implemented is a voltage domain crossing FIFO. Three options regarding how to partition the FIFO design to allow flexibility in placement of level shifters are described in detail in a separate document – see section “1.23 - References, #1 Voltage Crossing.docx.” Each option has distinct costs and benefits in the trade-off between number of level shifters required, timing challenges and complexity of implementation. In the long run, we may find that we need to support multiple of these options to support different needs, but our initial implementation should target “option C,” which leaves the read mux and flop array in the write-side voltage domain and puts level shifters on read pointer interface and on the output of the read mux. This looks like the best compromise between ease of timing analysis and quantity of level shifters required.

#### Properly Resetting Voltage Domain Crossing FIFO

Special care must be taken during initial power on reset and through subsequent power state changing events to make sure that both sides of the FIFO (read pointers and write pointers) are always restored to the same safe starting point and there are no race conditions that can create false non-empty indications.

Whenever reset is applied to one side of the FIFO, it must also be applied to the other side. Since the 2 sides of the FIFO structure are part of a different voltage domains, by definition they are part of different power domains, and therefore they have different power domain specific reset signals, i.e., *reset\_<PD>\_n* (cold reset) and *reset\_pd\_<PD>\_n* (warm reset). To meet the requirement that both sides must always be reset together, we must combine the resets from both sides’ power domains to create a reset signal that is applied to both sides that is active when any reset signal from either side goes active. Far side reset signals (those originating in the other side’s power domain) must pass through level shifters and isolation clamps before being synchronized and combined with the near side reset signals (after they too have been locally synchronized) to create the reset signal that is applied to the FIFO. The reciprocal logic applies to the other side of the FIFO. To minimize overhead, these cross coupled reset signals can be generated at a wrapper level above the multiple VD crossing FIFOs that will be required for a bridge interface.

There remains a small hazard upon reset assertion… there is a race between pointers being reset on one side and pointers being reset on the other where empty/full logic might generate false signals indicating non-empty status. To address this, both sides of the VD crossing FIFO participate in the *sleep\_req\_n*/*sleep\_ack\_n* handshake for both power domains, and when *sleep\_req\_n* for either power domain is asserted, the empty signal is qualified before it is used to generate VALID signals for downstream logic. It is expected that reset will not be asserted any time the Q-channel state for the associated power domain is outside of the Q\_STOP state (*QREQn* and *QACCEPTn* both low).

### Voltage Boundary Locations

Choosing optimal locations for supporting voltage domain boundaries is a difficult challenge that must balance physical design constraints against area, performance, and complexity costs. This is also discussed in greater detail in the separate Voltage Crossing.docx, so the choices will simply be summarized here. There are 2 basic cases we need to support.

#### Voltage Boundary at NOC Periphery: Host Interface to Bridge

Voltage boundaries may occur at the periphery of the NoC, so we must implement voltage crossing on the host interfaces to our bridges. While our AXI bridges already provide a mode that supports an asynchronous clock crossing boundary on the host interface, to minimize complexity in partitioning RTL, we use a synchronous version of the AXI bridge, and we instantiate clock crossing logic implemented with voltage crossing FIFOs between the host and the bridge. A similar solution will be deployed with our streaming bridge.

To minimize complexity, voltage crossing for the compound bridges (e.g., ahb-lite master/slave, apb slave, etc.) is also handled at the AXI bridge interface, so the voltage crossing FIFOs are inserted between the converter logic and the bridge. This requires changes to the compound bridge RTL where the top-level wrapper is eliminated, and genrtl in NocStudio instantiates the pieces within ns\_fabric.v. This provides the necessary flexibility to instantiate the voltage crossing logic in between converter and AXI bridge, and it allows genrtl to group the modules according to voltage and power domain boundaries.

##### Voltage Boundary at AXI Bridge



Figure 10 VD Crossing FIFO Implementation for AXI Master Bridge

Multiple VD Crossing FIFOs are required, one for each data direction for each of the read (AR) and write (AW) interfaces at the AXI bridge. These are instantiated inside a pair of wrappers, one for the host side voltage domain and one for the bridge side voltage domain. These wrappers implement some additional control logic to generate the cross coupled resets (as described above) and to implement interfaces to the NSPS for the *sleep\_req\_n*/*sleep\_ack\_n* handshakes. This logic ensures that the FIFO is truly empty and idle before asserting *sleep\_ack\_n*, and as described above in section 1.20.1.1, it qualifies VALID outputs. AxREADY outputs should also be deasserted (for those that are not already deasserted due to fencing as described in section 1.20.2.1.1.1.

###### Fencing and Draining for Master Bridge Interface

Fencing and draining logic is required in the host side wrapper for the VD FIFO that sits in front of an AXI master bridge since the AXI master bridge lacks visibility into the FIFO’s state. Because we rely on the *sleep\_req\_n*/*sleep\_ack\_n* handshake to qualify VALID outputs, the host side wrapper monitors *pd\_active* bits for both host side and bridge side power domains and implements fencing and draining for both domains. No fencing logic is required for VD crossing FIFOs supporting slave bridge interfaces as that function is handled upstream.

For fencing, the interfaces are halted at clean transaction boundaries by deasserting AxREADY. No new transactions are accepted, so there is no DECERR nor auto-wake response (this is comparable to AXI bridge behavior on power down without voltage domain boundary). For draining, the host side wrapper maintains 2 small counters that keep track of outstanding read and write transactions (these could potentially be merged into a single counter as long as coincident events are handled properly). All outstanding transactions, as monitored by the transaction counters, are allowed to complete (drain). At this point, *fence\_ack\_n* is asserted.

#### Voltage Boundary within the NoC

Voltage boundaries may also occur between elements within the NoC. This will be supported by inserting voltage safe clock crossing FIFOs in the links that cross voltage boundaries. These appear logically to be additional pipeline stages in the link. Correspondingly, while the receiving FIFOs at destination end of the link may operate synchronously (since the clock crossing is handled in the link), these FIFOs will need to increase in depth to compensate for the increased link latency. Unfortunately, the area penalty is multiplied by the number of VCs. Additionally, this new link FIFO is a constrained resource that must provide its own flow control back to the source end of the link, so a separate flow control mechanism must be implemented to provide back pressure when the link FIFO fills.

An alternative would instead be to convert the receiving FIFOs to be voltage safe. This avoids the area penalty of the additional link FIFO structure plus the increase in receive FIFO depth, but the number of level shifters required is multiplied by the number of VCs. This approach also suffers from the problem that voltage boundaries will generally be highly localized in physical design, yet the receivers on bi-directional links are generally physically separated, often by large distance. For this reason, it is not viable to force voltage boundaries only to occur at our receiving FIFOs.

In the long run, we may need to support both options, as the area savings are significant if the physical design allows the domain to be handled at the receiving FIFOs. However, for our initial implementation, we will only support in link voltage boundaries with in link voltage crossing FIFOs. This provides a more general and flexible solution, and it has the side benefit of allowing our routers to live entirely within a voltage domain, avoiding the need to partition our router RTL code.

Note: we may need to consider clock crossing in links even without voltage boundaries, as we may find designs where it is not possible to provide the same clock at both source and destination due to physical constraints.

* IMG feedback: aside from the need to partition RTL at voltage domain boundaries, they have expressed need to partition RTL at clock domain boundaries, which will likely match voltage domain and/or power domain boundaries, but likely applies even when the voltage/power domains are common across the 2 clock domains.

### DVFS – Dynamic Voltage Frequency Scaling

We need to handle cases where voltage within voltage domain and/or frequency within clock domain can change dynamically beyond just being gated. There are several aspects to address:

1. Specification in NocStudio: we need to define how customers will specify the range of operating points each voltage domain may have. Similarly, and more importantly for NoC optimization, customers will need to be able to specify how clock frequency can vary. This has implications on traffic analysis (do we need different traffic profiles for different DVFS operating points?) and overall NoC construction (need to make sure that correct structures are generated to handle the full range of operating points).
2. CPF generation: NocStudio will need to support generation of CPF (and ultimately UPF/IEEE 1801 compliant output) that properly describes DVFS domains.
3. RTL Implementation: HW needs to correctly operate across the complete range of valid DVFS operating conditions.
   1. Simple Approach – Treat all DVFS changes as power sequencing events: PMU must put affected logic in power-down safe state, adjust DVFS operating point, then restore logic to power-up state.
   2. Dynamic Changes: As long as certain conditions are met, our HW is designed to support dynamic changes to clock frequency and voltage without any special sequencing operations. The exception is for support of ratio-sync clock crossing – see section “1.20.3.2 - Special Handling of Ratio-Sync Boundaries with DVFS.”

#### Conditions for HW support of DVFS

In general, our HW can support DVFS without any additional enhancements, as long as the following conditions are met by the SoC implementation.

1. Clock frequency changes are glitch-free and in particular do not generate any runt pulses (clock edges that transition in less than the minimum cycle time supported at the current PVT conditions). This can be satisfied in at least a couple of ways.
   1. Glitch-free muxing between clock sources: muxing is controlled in such a way that that the low phase is stretched until a valid edge of then new clock arrives.
   2. Special clock circuitry (e.g., PLL) that can gradually transition between two frequencies.
2. Voltage and Clock shifts are managed to stay within valid PVT operating conditions: when lowering voltage, clock should be slowed first, and vice versa when raising voltage.
3. NocStudio config properly specifies the voltage and clock domains and their relationships so that proper voltage/clock crossing structures are inferred in the NoC RTL. Generally where clock frequency can change, the interfaces to other clock domains should be specified to be asynchronous. Ratio-sync clock relationships may be specified but do require special handling as described in the next section.

#### Special Handling of Ratio-Sync Boundaries with DVFS

The ratio-sync crossing structure must be modified to be controlled with a clock enable which indicates when the slow clock and fast clock share a rising edge. Changing frequency will require that the clock enable is always driven correctly, and that the new clock or clocks must continue to be synchronous with shared rising edges. Until such modified structure is available, we cannot support dynamic change of ratio-sync clock frequencies.

## RTL & CPF Structure and Related Back-End Considerations

### RTL Grouping by Power Domain

When low power features are enabled, gen\_rtl will implement an automatic layer of RTL grouping by power domain within ns\_fabric.v. Further user driven RTL grouping may be implemented within each power domain RTL group.

In the examples shown in Figure 11 and Figure 12 below, each of the 6 power domains (PD1-PD5 & PD\_System) will result in an RTL group within ns\_fabric.v, with the contents of the power domain (including an HFN module if it exists) instantiated within that RTL group. Note that within PD1, a user group has been created that will result in another layer of hierarchy within PD1.

### Pipeline Registers

Pipeline registers are assigned to the power domain of the transmitting side of the link and will be instantiated within that domain’s RTL group.

### High Fan-out Nets

High fan-out net generation for combinatorially aggregated signals is updated to be low power friendly by deploying 2 layers of aggregation.

1. Starting at the leaf sources, aggregation is done per source power domain, resulting in 1 sub-tree per source power domain.
2. If the ultimate destination power domain differs from one or more of the sources, a second layer of aggregation is implemented in the destination power domain, aggregating the outputs of all the sub-trees in the first layer.

Some HFNs, e.g., *fence\_ack\_asserted\_<PD>\_n*, will always differ in source vs. destination power domains and much of the time will originate from multiple different source domains, therefore requiring both layers of aggregation. Other HFNs, e.g., *sleep\_ack\_asserted\_<PD>\_n*, will be sourced homogeneously from one power domain, and when the PD FSM is mapped into the power domain it serves, only a single layer of aggregation is required, contained within that power domain.

In all cases, gen\_rtl will produce a single module for each power domain that contains all the HFN sub-trees that are part of that power domain, and that module will be instantiated within the appropriate power domain specific RTL group. That module will have CPF and SDC files associated with it, and there will be no lower level CPF or SDC regardless of the internal structure.

#### HFN Structure When PD FSMs Live in System Power Domain

One implementation option is to have all PD FSMs and live in the system power domain (always on – same domain as PMU). This results in a single layer HFN implementation where all the aggregation is done in the system power domain as indicated in the example diagram that follows.

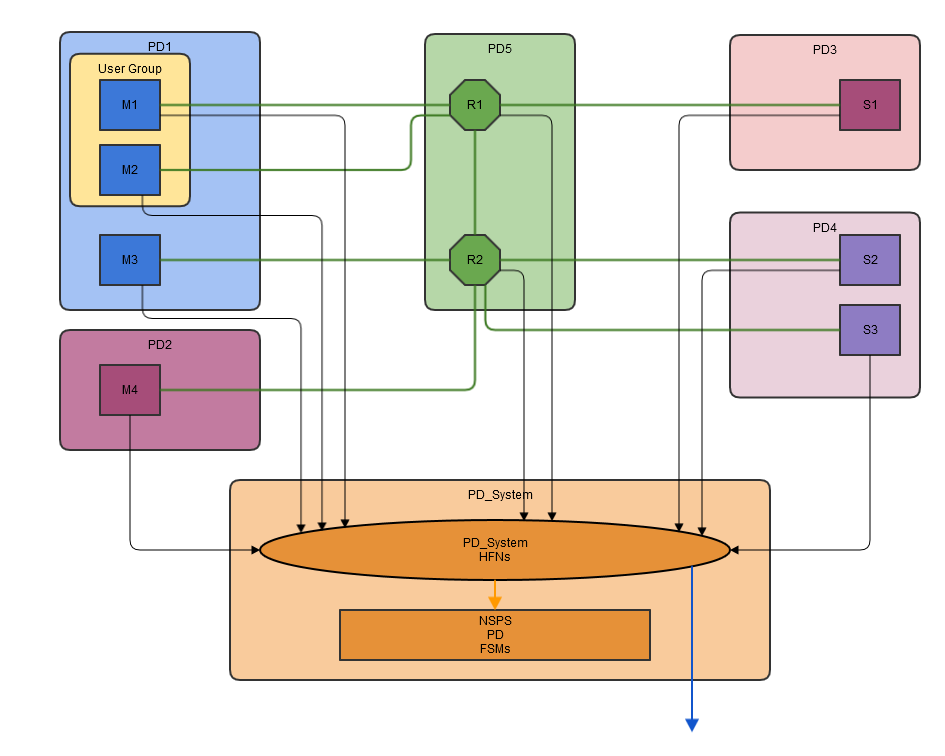


Figure 11 HFN Structure When PD FSMs Live in System Power Domain

In this figure:

* Blue Line: roots of aggregated signals that are sourced from multiple power domains that feed into NoC pins (e.g., *QACTIVE\_<PD>*, interrupt), aggregated entirely in system power domain and driven from there.
  + *wake\_req\_asserted\_<PD>*, *idle\_status\_<PD>\_n*, interrupt sources
* Orange Line: roots of aggregated trees that feed PD FSMs.
  + *fence\_ack\_asserted\_<PD>\_n, fence\_ack\_deasserted\_<PD>\_n, sleep\_ack\_asserted\_<PD>\_n*, *sleep\_ack\_deasserted\_<PD>*
* Black Lines: async outputs of NoC elements that must be aggregated in HFN combinatorial networks.
* Green lines: mission mode NoC datapath (i.e., NoC links).

#### HFN Structure When PD FSMs are Mapped to Native Power Domain

When PD FSMs are mapped into the power domains that they serve, some of the aggregated HFNs require a dual layer structure, others only a single layer as indicated in the example diagram that follows.

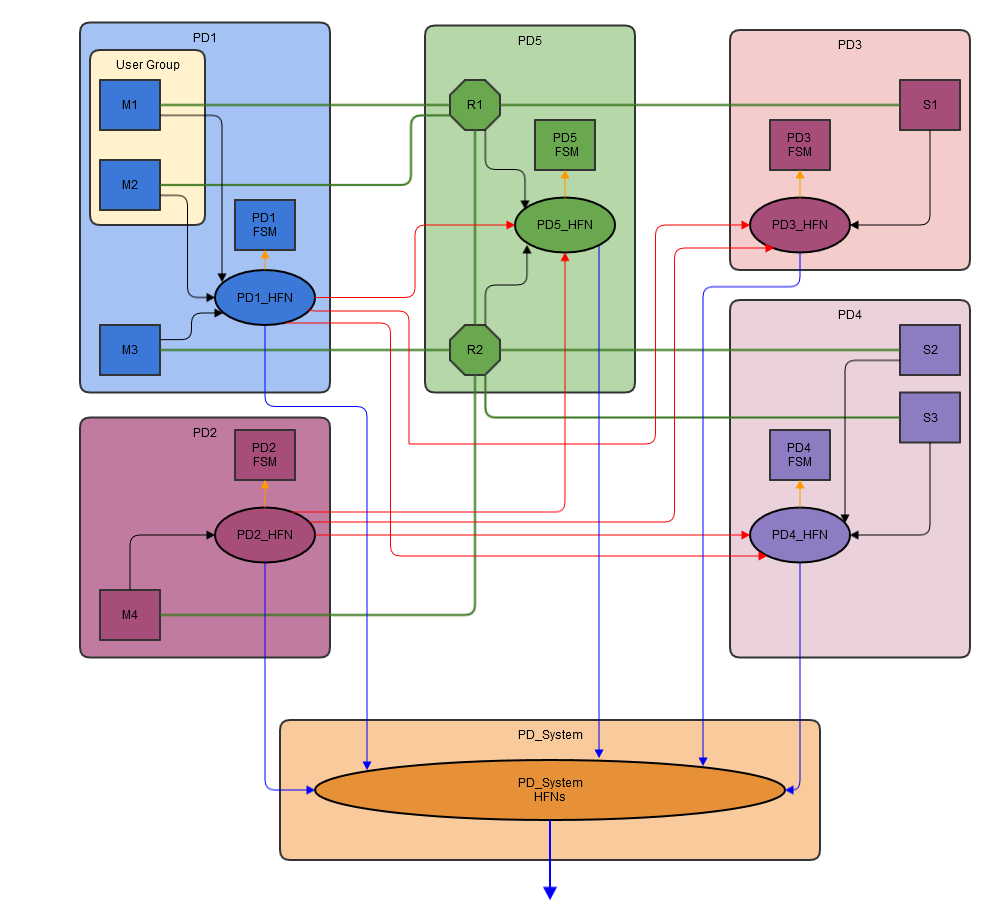


Figure 12 HFNs When PD FSMs Mapped to Native Power Domains

In the figure above:

* Red Lines: aggregated signals that are sourced from multiple power domains that feed into the power domain of the PD FSM.
  + *fence\_ack\_asserted\_<PD>\_n, fence\_ack\_deasserted\_<PD>\_n*
* Blue Lines: aggregated signals that are sourced from multiple power domains that feed into NoC pins (e.g., *QACTIVE\_<PD>*, interrupt) that are driven from the system power domain (or some particular power domain identified by the user).
  + *wake\_req\_asserted\_<PD>*, *idle\_status\_<PD>\_n*, interrupt sources
* Orange Lines: roots of aggregated trees that include inputs from local power domain and possibly sub-tree inputs from other power domains (red lines).
  + Local PD HFNs: for some networks, all sources and destination are in the same power domain, so only 1 layer is required: *sleep\_ack\_asserted\_<PD>\_n*, *sleep\_ack\_deasserted\_<PD>*
* Black Lines: async outputs of NoC elements that must be aggregated in HFN combinatorial networks.
* Green lines: mission mode NoC datapath (i.e., NoC links).

## Phased Development & Optional Features

This section has been moved into a Confluence page found in the following location:

[Low Power Phased Development Confluence Page](https://netspeed.atlassian.net/wiki/display/EN/Low+Power+V2+Phased+Development)

## References

1. Voltage Crossing.docx
   * [svn/doc/ccNoC/Voltage Crossing.docx](https://www.assembla.com/code/netspeed_noc/subversion/nodes/33992/doc/ccNoC/Voltage%20Crossing.docx)
2. [ARM AMBA Low Power Interface Specification](https://www.assembla.com/code/netspeed_noc/subversion/nodes/32897/doc/external_specs/IHI0068B_low_power_interface_spec.pdf)
3. <https://www.semiwiki.com/forum/content/3697-nocs-system-level-power-management.html>
4. http://info.arteris.com/SoC\_Power\_Management
5. <http://www.ocpip.org/uploads/documents/OCP_Disconnect_Protocol_15032007.pdf>
6. <http://www.ocpip.org/uploads/dynamic_areas/Ct9Rr6XmkN84Y6MvTouu/947/OCP_Disconnect_Final_SAME2009.pdf>
7. [Staged Fencing and Autowake Confluence Page](https://netspeed.atlassian.net/wiki/display/EN/Staged+Fencing+and+Autowake)