© Copyright The SPIRIT Consortium, 2006. All rights reserved.

IP-XACT™ For RTL

Document number:	IP-XACT™ User Guide v1.2
Version:	Released
Date of Issue:	July – 2006 (editorial change only of March 2006 version)
Author:	SPIRIT Schema Working Group Membership
Purpose:	Provide introduction to the IP-XACT Standard and its use model. Document IP-XACT schema specification and methodology.

Abstract

To accelerate the design of large System-on-Chip (SoC) solutions, the semiconductor industry needs a common specification mechanism for describing and handling IP that enables automated configuration and integration through plug-in tools. This is the goal of The SPIRIT Consortium™. The SPIRIT Consortium works to reach this goal through the provision of specifications to: ensure delivery of compatible IP descriptions from multiple IP vendors; better enable importing and exporting complex IP bundles to, from and between Electronic Design Automation (EDA) tools for SoC design (design environments); better express configurable IP using IP meta-data; and better enable provision of EDA vendor-neutral IP creation and configuration scripts (generators / configurators).The founding companies of The SPIRIT Consortium have combined their extensive experience in IP development, supply, integration and electronic design automation (EDA) to deliver these IP-XACT™ specifications.

IP-XACT v1.2 specifically addresses requirements for RTL design, including IP packaging, configuration and SoC integration. In addition, it includes implementation constraints for flow from initial RTL implementation through synthesis.

This User Guide is part of the IP-XACT specification deliverables, 1.2 version. This document describes the IP-XACT Schema and Loose Generator Interface, as applied to open-source examples based around the Leon processor. Also described here are suggested flows, use models, and guidelines for IP-XACT usage and validity. Relative to the materials already provided in the IP-XACT v1.0 and v1.1 User Guides, this document includes some error corrections, additional schema elements to handle design configuration and a definitive hierarchy solution. The SPIRIT Steering Committee companies have developed this document along with the SPIRIT Schema and Generator Interfaces. It has been reviewed by the SPIRIT Reviewing Membership prior to release.

Keywords

IP-XACT, SPIRIT, XML schema, loose generator interface, Design Environment, use models, flows, examples, interoperability, tools, implementation constraints

A meta-data description specification of: The SPIRIT Consortium™

© Copyright The SPIRIT Consortium 2006. All rights reserved.

TABLE OF CONTENTS

1 ABOUT THIS DOCUMENT 5
1.1 PURPOSE OF THIS DOCUMENT 5
1.2 ACCESS AND LICENSE TO MATERIAL IN DOCUMENT .... **ERROR! BOOKMARK NOT DEFINED.**
1.3 TARGETED AUDIENCE AND PREREQUISITES 5
1.4 CONTRIBUTORS 7
1.5 CREATION PROCESS 7
1.6 REFERENCES 7
1.7 ORGANIZATION OF THIS DOCUMENT 7
2 INTRODUCTION TO SPIRIT 9
2.1 GOALS AND VISION 9
2.1.1 Consortium goals 10
2.1.2 Architectural Goals 10
2.2 SPIRIT DESIGN ENVIRONMENT 11
2.2.1 SoC Design Tool 12
2.2.2 Design Intellectual Property 12
2.2.3 Generators and Configurators 13
2.2.4 SPIRIT Interfaces 14
2.3 SPIRIT-ENABLED IMPLEMENTATIONS 15
2.3.1 Design Environment Provider 16
2.3.2 Point-tool Provider 16
2.3.3 IP provider 16
2.3.4 Generator Provider 16
2.3.5 Support of other standards 16
3 SPIRIT INTEROPERABILITY USE MODELS 19
3.1 ROLES AND RESPONSIBILITIES 19
3.1.1 The component IP Provider 19
3.1.2 The SoC Design IP Provider 20
3.1.3 The SoC Design IP Consumer 20
3.1.4 The Design Tool Supplier 20
3.2 SPIRIT IP EXCHANGE FLOWS 20
3.2.1 Component or SoC design IP provider use model 21
3.2.2 Generator / configurator provider use model 21
3.2.3 SoC Design-tool provider use model 22
4 SPIRIT SCHEMA 23
4.1 SPIRIT OBJECTS 23
4.1.1 Definitions 23
4.1.2 Objects interactions 24
4.1.3 VLNV 25
4.2 SPIRIT SCHEMA OVERVIEW 26
4.2.1 Design schema 26
4.2.2 PMD schema 26
4.2.3 Component schema 26
4.2.4 Bus definition schema 26
4.2.5 Generators schemas 27
4.3 SPIRIT DESIGN MODEL 27
4.4 SPIRIT CONFIGURATION 31
4.5 SPIRIT PLATFORM META DATA (PMD) MODEL 32
4.5.1 XSL Stylesheet 33

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.6 SPIRIT COMPONENT MODEL 34
4.6.1 Component interfaces *35*
4.6.2 Whitebox interfaces *36*
4.6.3 Component choices *37*
4.6.4 Component Address space *38*
4.6.5 Component Memory Map *40*
4.6.6 Memory bank *43*
4.6.7 Register description *44*
4.6.8 Component HW Models *46*
4.6.9 Component Implementation Constraints *49*
4.6.10 Component Files *50*
4.7 HIERARCHY REPRESENTED BY A DESIGN FILE 50
4.8 SPIRIT BUS DEFINITION 51
4.9 SPIRIT BUS AND INTERCONNECT MODEL 52
4.9.1 Bus interface *53*
4.9.2 Interfaces connection *54*
4.9.3 Bus internal representation *58*
4.9.4 Memory Map *65*
4.9.5 Remapping *71*
4.9.6 Signal Connections *77*
4.9.7 Clock and Reset Handling *81*
4.9.8 Bus interface parameter declaration *82*
4.9.9 Bus interface parameter *83*
4.10 REFERENCE BUS DEFINITIONS 83
4.10.1 The difference between an external bus and an internal/digital interface *84*
4.10.2 Location of reference BusDefs *85*

5 SPIRIT GENERATORS 86

5.1 GENERATOR REGISTRATION 86
5.2 TIGHT INTEGRATION 86
5.3 LOOSE INTEGRATION 87
5.3.1 Definition *87*
5.3.2 Typical DE flow *88*
5.3.3 Configurators *92*
5.4 GENERATOR CHAIN 92
5.4.1 Generator Naming Convention *93*
5.4.2 Phase Numbers *95*

6 SPIRIT SEMANTIC RULES 97

6.1 CROSS REFERENCES AND VLNVS 97
6.2 INTERCONNECTIONS 98
6.3 CHANNELS AND BRIDGES 100
6.4 MONITOR INTERFACES AND INTERCONNECTIONS 100
6.5 CONFIGURABLE ELEMENTS 101
6.6 SIGNALS 103
6.7 REGISTERS 103
6.8 MEMORY MAPS 104
6.9 ADDRESSING 104
6.10 HIERARCHY 105
6.11 HIERARCHY AND MEMORY MAPS 108
6.12 RULES REQUIRING EXTERNAL KNOWLEDGE 109
6.13 PMD FILES 110
6.14 ADDRESSING FORMULAS 110
6.14.1 Overview *111*
6.14.2 Breaking down the path *112*

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

6.14.3 Connection from "just outside" bus interface A to "just outside" bus interface B.115
6.14.4 Connection from an address block to "just outside" the associated slave bus interface 116
6.14.5 Connection through a channel from "just outside" the mirrored slave bus interface to "just outside" the mirrored master bus interface 120
6.14.6 Connection from "just outside" the master bus interface to master component's address space 121
6.14.7 Connection across a bridge from "just outside" the master bus to "just outside" the slave bus interface. 122

7 BACKWARD COMPATIBILITY 123

8 VERIFICATION SUPPORT IN SPIRIT 124

8.1 MONITOR BUS INTERFACE & INTERCONNECTION 124
8.1.1 Monitor interfaces 124
8.1.2 Monitor Interface connection 125
8.2 WHITE BOX INTERFACE 126
8.3 DESCRIBING VERIFICATION SEQUENCES 126
8.3.1 Representing the sequence 127
8.3.2 Associating the sequence with Design IP 127
8.3.3 Associating the sequence with Verification IP 127

9 APPENDIX: USE CASE EXAMPLES 128

9.1 PACKAGING OF A COMPONENT 128
9.1.1 Introduction 128
9.1.2 Describing the bus interfaces 128
9.1.3 Describing the Memory Map 130
9.1.4 Describing the hardware model 131
9.1.5 Describing the configuration choices 134
9.1.6 Describing the file sets 134
9.1.7 Description of timing constraints 136
9.1.8 Other Clock Drivers 136
9.1.9 Example Source Code 136
9.2 IMPLEMENTATION CONSTRAINTS 137
9.2.1 Timing Constraints 137
9.2.2 External Load/Drive Constraints 137
9.2.3 Point to Point Timing Requirements 138
9.2.4 Design Rule Constraints 139
9.3 LOOSE GENERATOR DUMP 139
9.4 GENERATOR EXAMPLE 140

10 APPENDIX: DEFINITIONS AND NOTATION 142

10.1 DEFINITIONS 142
10.2 NOTATIONS 146
10.3 LAST PAGE OF DCUMENT 146

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

1 ABOUT THIS DOCUMENT

1.1 Purpose of this Document

This document collects the input from the SPIRIT Schema Technical Working Group (TWG) members and describes the structure, semantic and use models of the IP-XACT v1.2 schema. This document is part of the IP-XACT v1.2 deliverables. A separate document, IP-XACT-Release Notes v1.2, provides the summary of schema changes between V1.1 and V1.2 versions.

This document provides information for users to adopt and validate the proposed specification against their own design flows, tools and Intellectual Property (IP).

1.2 Access and license to material in document

This work forms part of The SPIRIT Consortium's IP-XACT specification.

This work contains trade secrets and proprietary information that is the exclusive property of individual members of The SPIRIT Consortium. Use of these materials are governed by the legal terms and conditions outlined in the IP-XACT specification disclaimer available from www.spiritconsortium.org

1.3 Statement of use of Spirit Consortium Specifications

The Spirit Consortium Specification documents are developed within The Spirit Consortium and the Technical Working Groups of The Spirit Consortium, Inc. The Spirit Consortium develops its specifications through a consensus development process, approved by its members and board of directors. This brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers serve without compensation from the Spirit Consortium. While the Spirit Consortium administers the process and establishes rules to promote fairness in the consensus development process, the Spirit Consortium does not independently evaluate, test, or verify the accuracy of any of the information contained in its specifications.

Use of a Spirit Consortium Specification is wholly voluntary. The Spirit Consortium disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other Spirit Consortium Specification or document.

The Spirit Consortium does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. The Spirit Consortium Specifications documents are supplied "AS IS."

The existence of a Spirit Consortium Specification does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of a Spirit Consortium Specification. Furthermore, the viewpoint expressed at the time a specification is approved and issued is subject to change due to developments in the state of the art and comments received from users of the specification. Every Spirit Consortium Specification is subject to review for revision and update. Users are cautioned to check to determine that they have the latest edition of any Spirit Consortium Specification.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

In publishing and making this document available, The Spirit Consortium is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the Spirit Consortium undertaking to perform any duty owed by any other person or entity to another.

Any person utilizing this, and any other Spirit Consortium Specification or document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of specifications as they relate to specific applications. When the need for interpretations is brought to the attention of The Spirit Consortium, The Spirit Consortium will initiate action to prepare appropriate responses. Since the Spirit Consortium Specifications represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, The Spirit Consortium and the members of its Technical Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of Spirit Consortium Specifications are welcome from any interested party, regardless of membership affiliation with The Spirit Consortium. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on specifications and requests for interpretations should be addressed to:

The Spirit Consortium
1370 Trancas Street #163
Napa, CA 94558
USA
–Or–
feedback@spiritconsortium.org

Note: Attention is called to the possibility that implementation of this specification may require use of subject matter covered by patent rights. By publication of this specification, no position is taken with respect to the existence or validity of any patent rights in connection therewith.

The Spirit Consortium shall not be responsible for identifying patents for which a license may be required by a Spirit Consortium specification or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

The Spirit Consortium is the sole entity that may authorize the use of The Spirit Consortium-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

1.4 Targeted audience and prerequisites

This document targets the IP providers and EDA vendors that want to model IP according to the IP-XACT specifications. Chapters 2 and 3 do not require the reader to have any specific understanding of XML terminology and technology. All subsequent chapters assume the reader has knowledge of XML. For XML background, the reader is referred to www.w3.org.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

1.5 Contributors

The following company representatives have contributed to the creation of this document and supporting specifications:

- ARM: Allan Cochrane, Christopher Lennard, Andrew Nightingale, Anthony Berent
- Cadence: Jean-Michel Fernandez, Giles Hall, Saverio Fazzari
- LSI Logic: Gary Delp, Wayne Nation, Gary Lippert
- Mentor: John Wilson, Gary Dare, Mark Glasser
- Philips Semiconductor: Geoff Mole, Ahmed Hemani, Roger Witlox, Greg Ehmann
- ST Microelectronics: Christophe Amerijckx, Serge Hustin, Anthony McIsaac
- Synopsys: Mark Noll, Bernard DeLay, John A. Swanson

1.6 Creation process

This document has been created during the IP-XACT v1.2 development process. As such, it reflects both the key issues found in reviewing the original schema contributions against the IP-XACT requirements, and in developing this specification towards multi-vendor IP and tool support. The document also provides a structural and semantic description which will assist the user in applying the IP-XACT schema and generator interface to complex IP and point tools / scripts. These key areas of application knowledge have been developed from the schema validation process.

This User Guide is released along with the IP-XACT Schema, generator interface and illustrative examples. Feedback provided during the membership review process has been included, clarifying the specification and its use-models.

1.7 References

It is expected that this Users Guide will be read in conjunction with the following resources and materials:

- SPIRIT Web server: http://www.spiritconsortium.org. All of the IP-XACT specifications are posted in their final form on this web site. Any printed copies of the specification should be considered at risk of being out of date.
- IP-XACT Requirements, V1.1: http://www.spiritconsortium.org/releases/1.1
- IP-XACT Schema v1.2 (a gzipped tar file containing the schema files): http://www.spiritconsortium.org/releases/1.2
- IP-XACT Schema on-line documentation, v1.2 (a gzipped tar file containing HTML documentation generated from the schema files with the above filenames): http://www.spiritconsortium.org/releases/1.2
- IP-XACT Leon Examples (IP & Generator), v1.2: http://www.spiritconsortium.org/releases/1.2
- IP-XACT-Release Notes v1.2. Provide a summary of schema changes between v1.1 and v1.2 versions.

1.8 Organization of this document

This document first gives an overview of IP-XACT (chapter 2) and introduces the reader to the entity definitions, nomenclature used in the document, and the use models specifically supported by this work (chapter 3). Following this is a structural description of the IP-XACT schema (chapter 4), the representation of design IP in IP-

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

XACT XML and the provision of Generators that conform to the IP-XACT Schema and Generator Interface (chapter 5). Next the semantic rules associated with the IP-XACT schema are detailed (chapter 6). Followed by notes on backwards compatibility for existing users of IP-XACT v1.0 and v1.1 are given (chapter 7). Then the structures of the IP-XACT schema supporting the construction of verification platforms is described (chapter 8). Detailed examples based on Leon platform IP are described in the first appendix (chapter 9) and lastly the definitions and other notations about this document (chapter 10).

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

2 INTRODUCTION TO IP-XACT

To accelerate the design of large System-on-Chip (SoC) solutions, the semiconductor industry needs a common specification mechanism for describing and handling IP that enables automated configuration and integration through plug-in tools. This is the goal of the SPIRIT consortium. SPIRIT will reach its goals through the provisions of specifications to: ensure delivery of compatible IP descriptions from multiple IP vendors; better enable importing and exporting complex IP bundles to, from and between Electronic Design Automation (EDA) tools for SoC design (design environments); better express configurable IP using IP meta-data; and better enable provision of EDA vendor-neutral IP creation and configuration scripts (generators / configurators). The founding companies of SPIRIT have combined their extensive experience in IP development, supply, integration and electronic design automation (EDA) to deliver these specifications. This document represents the founding membership's guidelines for IP-XACT usage and validity. Each specifications release from the SPIRIT consortium will address technology in three specific areas:

- IP meta-data schema. The meta-data schema will create a common way to describe IP through a common format of IP meta-data, compatible with automated integration techniques and enabling integrators to use IP from multiple sources with IP-XACT-enabled tools.
- Configuration and generation interface: The interface for integration of IP creation and configuration scripts, and IP-XACT-packaged point-tools will provide a standard method for linking into IP-XACT-enabled SoC design environment tools, enabling a more flexible, automated and optimized development flow. IP-XACT-enabled tools and generators, configurators will be able to interpret, configure, integrate and manipulate IP blocks that are valid and remain valid based on the specified IP meta-data description.
- IP-XACT methodology: Use model for the IP meta-data schema, IP configuration and generator interface, including how to define and utilize generator sequencing,

The IP-XACT v1.2 specifications release is intended to comprehensively address Register Transfer Level (RTL) design, including packaging configuration and SoC integration.

Where possible, IP-XACT leverages existing standards. These associated standards include specifications describing: VSIA (particularly Virtual Component Transfer), XSLT, XPath, and XML.

2.1 Goals and Vision

The IP-XACT specifications enable increased automation for IP selection, configuration and integration and will enable a multi-vendor IP and design-tool optimized flow from architectural design through system simulation to chip layout. Complying with the specifications will enable IP suppliers and tool vendors to offer immediate, proven solutions, helping system manufacturers develop complex 'first-time-right' SoCs, SiPs, and other complex composed systems facilitating rapid release to market. It will also provide the requirements for creating IP within a company in an IP-XACT-enabled fashion so that company internal and external IP libraries can be handled in a consistent way.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

2.1.1 Consortium goals

- Develop specifications for describing IP and point-tools (generators) so that they may be packaged with consistent meta-data descriptions, as well as interfacing and packaging of supporting scripts for IP generation and configuration. This is to enable more efficient and cost-effective SoC design utilizing IP from multiple sources.
- Test the proposed specifications within multiple live projects, providing a solid proof-of-concept prior to release. Proof of concept to ensure applicability of the specification to multiple IP and EDA vendors in industrially significant tools and design-examples.
- Test the proposed specifications as comprehensive for support of SoC design stages, including RTL design (IP-XACT v1.2), and system-design and verification (IP-XACT with ESL Extentions).
- Transfer the defined specifications to an international standards body.

The IP-XACT specifications can greatly benefit automated design-flow integration when used in a complete and correct manner. It is therefore necessary to define rules for judging a complete and correct implementation. These rules are defined in the Section 2.3, IP-XACT-Enabled Implementations. Only when a tool, IP, configurator, or generator complies with the rules outlined in Section 2.3 can they be classified as 'IP-XACT-Enabled'.

2.1.2 Architectural Goals

The IP-XACT schema and specifications will enable SoC design projects to encapsulate the infrastructure and engineering knowledge that promote correct by construction techniques.

2.1.2.1 IP-XACT v1.2

- Full support of RTL design, including any component type, Hardware Description Language (HDL), configuration or connection-type at this level of abstraction.
- A set of architectural rules that define constraints and guide the connectivity and usage of each IP in a given platform. Example rules would be bus definitions and rules that specify the compatible buses that this IP is allowed to connect to.
- A set of bus definitions that describe the meaningful signal names of each bus type so that it can be connected to compatible IP.
- Provide a mechanism to define any bus structure in terms of signal names that can be used for hooking up other IP.
- A technique of gluing IP together with supportive logic and connections without the need for user interaction.
- An example of this would be supporting IP that needs to be included when certain combinations of IP are put together to form part of the design.
- Methods for defining register information and address spaces.
- A way of separating platform specific meta-data required for a piece of IP that may not be part of the generic IP definition.
- A way of specifying configuration options for a piece of IP, and enabling selection of these options.
- A mechanism to enable associations between configurable elements of a design to be specified and handled.
- A way of storing persistent data that supports the iterative process of user configurations and the options used to derive them.

 © Copyright The SPIRIT Consortium 2006.
All rights reserved.

- A basic interface to enable configuration and generation scripts such as those provided with configurable IP and point-tools, to enable them to be driven by an IP-XACT-enabled environment.
- A mechanism for adding implementation constraints to IP-XACT descriptions of IP to aid in flow to synthesis.

2.1.2.2 IP-XACT with ESL Extensions v1.4

- Extend IP-XACT v1.2 while remaining compatible.
- Full support of Electronic System Level (ESL) design, including any component type, ESL language, configuration or connection-type at this level of abstraction, and mixed with HDL design.
- Full support of system verification, including any verification architecture, verification-language, configuration or connection type.
- A way of selecting and defining which views of an IP can be integrated and / or verified together.
- A full API for integrating configuration and generation scripts and tools into an IP-XACT environment

The remainder of this user-guide will describe IP-XACT for RTL (v1.2). For a description of the scope of IP-XACT with ESL Extensions (v1.4), the reader is referred to the IP-XACT Requirements Document [3].

2.2 IP-XACT Design Environment

It is important to describe the IP-XACT specification in the context of its basic use-model, the Design Environment (DE). This is the co-ordination of a set of tools and IP, or expressions of that IP (e.g., models) such that the system-design and implementation flows of a SoC are efficiently enabled and re-use centric. Co-ordination is managed through creation and maintenance of a meta-data description of the SoC

The IP-XACT specification can be viewed as a mechanism to express and exchange information about design IP and its required configuration. For the IP provider, the IP configuration or generator script provider, the point-tool provider, or the SoC design-tool provider to claim IP-XACT compliance they must adhere to the completeness and IP-XACT semantic rules as outlined in Section 6.

The use of the SPIRIT consortium specified formats and interfaces are shown in Figure 1, and described in the following subsections. The IP-XACT specifications relate directly to the aspects of the DE indicated in bold.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

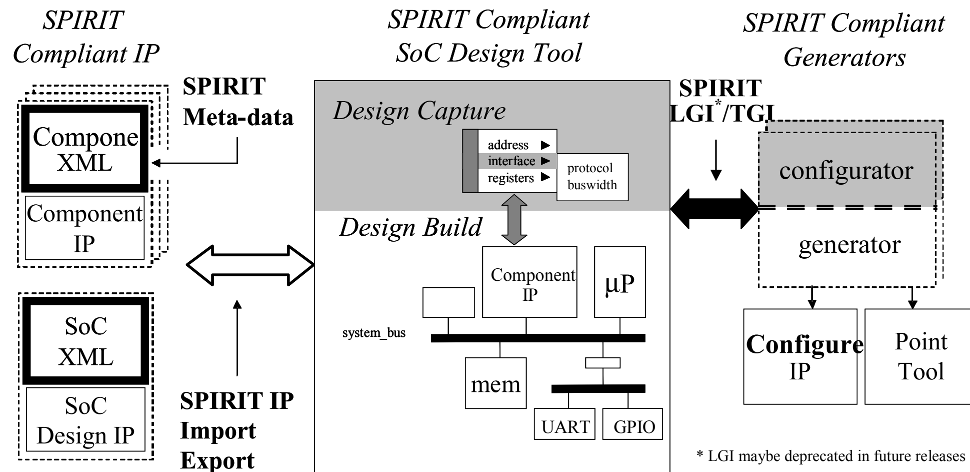


Figure 1 - IP-XACT Design Environment

2.2.1 SoC Design Tool

SoC design tools enable the designer to work with IP-XACT design IP through a coordinated front-end and IP design database. These tools create and manage the top-level meta-description of SoC design and provide two basic types of services: design capture, which is the expression of design configuration by the IP provider and design intent by the IP user; and design build, which is the creation of a design (or design model) to those intentions.

As part of design capture, a SoC design tool must recognize the structure and configuration options of imported IP. In the case of structure, this implies both structure of the design (e.g., how specific pin-outs refer to lines in the HDL code) as well as structure of the IP package (e.g., where design files and related configurators / generators are provided in the packaged IP data-structure). In the case of configuration, this is the set of options for handling the imported IP (e.g., setting base address and offset, bus-width, etc.) that may be expressed as configurable parameters in the IP-XACT meta-data.

As part of design build, generators would either be provided internally with the SoC design tool to achieve the required IP integration or configuration, or provided externally (e.g., by an IP provider) and launched by the SoC design-tool as appropriate.

The SoC design tool set around which a Design Environment (DE) is structured is where the support for the conceptual context and management of IP-XACT meta-data resides. However, the IP-XACT specifications make no requirements upon SoC design tool-architecture, or upon tool internal data structures. To be considered IP-XACT v1.2 enabled, a SoC design-tool must support the import / export of IP expressed with valid IP-XACT v1.2 meta-data for both component IP and systems, and it may support the IP-XACT Loose Generator Interface (LGI) and for v1.3 must support the Tight Generator Interface (TGI) for interfacing with generators / configurators external to the DE.

2.2.2 Design Intellectual Property

IP-XACT is structured around the concept of IP re-use. IP may be considered from the perspective of the object itself, its supporting views, and meta-data description. In IP-XACT v1.2, the specifications are to be comprehensive for all design objects required to support RTL design and integration. These include the following:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

- Design Objects
  - Fixed HDL descriptions: Verilog, VHDL
  - Configurable HDL descriptions (e.g., bus-fabric generators)
  - Design models for RT simulation (e.g., compiled core models)
  - HDL-specified Verification IP (e.g., basic stimulus generators and checkers
- IP Views. This is a list of different views (levels of description, languages) to describe the IP object. In IP-XACT v1.2, these views include:
  - Design view: RTL Verilog or VHDL, flat or hierarchical components
  - Simulation view model views, targets, simulation directives, etc.
  - Documentation view: specification, user guide, etc.
- Meta-data Description
  - AN IP-XACT schema-compatible way of describing the: design-history, hierarchy, locality, object association, configuration options, constraints, and integration requirements of an IP object

2.2.3 Generators and Configurators

Generators and configurators are executable objects that may be integrated within a SoC design tool (referred to as 'internal'), or provided separately as an executable that can be launched (referred to as 'external'). Generators and configurators may be provided as part of an IP package (e.g., for configurable IP such as a bus-matrix generator), or as a way of wrapping point tools for interaction with a SoC design tool (e.g., external design netlister, external design checker, etc.). In IP-XACT v1.2, external configurators and generators may use the Loose Generator Interface (LGI) or in v1.3, the Tight Generator Interface (TGI). IP-XACT is neutral with respect to the language in which generators / configurators are provided (e.g., Tcl/Tk, Perl, Java, C, etc.)

2.2.3.1 Configurators

Configurators are launched during a SoC design-capture phase. They express the options that a user may take in configuring an IP (e.g., selection of number of master-ports on a bus-fabric, setting priorities for an interrupt controller, etc.), or the automatic creation or checking of legal configuration (e.g., master port connection to a master-mirror interface, etc.). When a configurator completes, it sets the SoC design-tool internal database to the desired configuration. In many cases, basic element configuration (e.g., memory size, bus width selection) would be expected to be handled by all SoC design tools, so internal configuration support could be relied upon. In such cases, the only required input to the SoC design tool would be the allowable range of bus-widths specified in the IP meta-data.

In order to use specialized configuration options not supported by the available SoC design tools, an IP provider may chose to provide an external configurator with their IP component. In these cases, a SoC design tool can launch the configurator when the user needs to make the specialized choice. For IP-XACT v1.3, external configurators must operate upon IP-XACT-enabled meta-data through the LGI or the TGI. Following execution, LGI configurators return a difference to the SoC meta-data that the SoC design-tool must interpret, and modify its internal SoC meta-data representation to match. TGI configurators directly modify the SoC meta-data through the Tight Generator Interface to the design tool.

2.2.3.2 Generators

Generators are executables (e.g. scripts) that operate upon an IP or the system design based upon a configuration request. Generators are launched during the build phase of a design environment. i.e., Generators create the design to the

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

specification provided in the design capture phase. Generators may perform multiple tasks, such as IP creation, configuration, post-generation checking, simulation set-up, etc... Generators may be part of a configurable IP package, or a specific design-automation feature such as an architecture-specific design-rule checker. Like configurators, some generation services will be provided internally to SoC design tools, and some specialized generation services may need to be provided externally. For IP-XACT v1.2, external generators operate upon IP-XACT compliant meta-data provided through the LGI or in v1.3 the TGI.

Not all generators will require the ability to modify the internal meta-data representation of the SoC. For example, a generator checking build correctness may just return a pass/fail. However, many generators will need to return some modifications to the meta-data description even if minor. For example, an IP generator will need to express to the SoC design tool where the generated RTL is placed.

Generators can be associated with phases in the design process that enables sequencing of chains of generators. This is critical in providing script-based support of SoC creation and simulation. IP-XACT based generators can be sequenced into generator chains.

2.2.4 IP-XACT Interfaces

There are two obvious interfaces expressed in Figure 1: from the SoC Design Tool to the external IP libraries and from the SoC design Tool to the generators / configurators. In the former case, the IP-XACT specifications are neutral on the use of design-tool interfaces to IP repositories. While being able to read and write IP with IP-XACT meta-data is a requirement of the specification, the formal interaction between an external IP repository and a SoC design-tool is not specified.

In the case of the generators / configurators, two approaches are to be taken. The LGI solution provided with IP-XACT v1.2 and earlier versions provides basic services from launch and meta-data import / export to generators / configurators, and the Tight Generator Interface (TGI) that is being introduced with IP-XACT v1.3 creates a stronger integration.

2.2.4.1 Loose Generator Interface (LGI):

The LGI is not formally an API. Rather, it is a basic meta-data export / import mechanism which provides the following functionality:

- Mechanism for registering a generator / configurator with an IP and defining its sensitivity list.
- Mechanism for dumping the SoC design-tool meta-data description in an IP-XACT compliant format.
- Mechanism for registering success/fail of generator/configurator completion.
- Mechanism for returning a modification (difference) to the meta-data that the SoC design tool must interpret.

As the LGI is based on a dump and difference-return mechanism, it is not interactive with the SoC design tool during execution. As such, loose generators will be identified as ‘READ ONLY’ or ‘READ / WRITE’ in terms of their need to modify the SoC meta-data description. In the latter case, the SoC design-tool will need to ensure that consistency of meta-data is maintained between the generator in execution and its internal version (e.g., blocking execution)

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

As the LGI performs a meta-data dump, the generators / configurators need to be able to parse the exported SoC meta-data themselves to locate the information on which they operate rather than provide specific queries to the SoC design tool.

The LGI is expected to provide sufficient functionality for IP configuration and generation, and execution of external design-automation features, which do not require a high degree of iterative interaction with the SoC design tool.

2.2.4.2 Tight Generator Interface (TGI):

IP-XACT v1.3 supports a TGI that is a full API enabling superior integration between external task-specific tools, configurators and generators. The TGI is markedly more efficient in the support of external configurators than the IP-XACT LGI due to the generally rapid nature of configurator execution. IP-XACT v1.1 generators and configurators are expected to migrate from LGI support to TGI support following the release of the IP-XACT v1.2 specification. More details can be found at: http://www.spiritconsortium.org/releases/tgi/index.html.

2.3 IP-XACT-Enabled Implementations

Complying with the rules outlined in this section allows the provider or tools, IP, generators or configurators to class their products as IP-XACT-Enabled. Conversely, any violation of these rules removes that naming right. This section first introduces the set of metrics for measuring valid use of the specifications. It then specifies when those validity checks must be performed. These validity checks are applied against the various classes of products: Design Environments, Point Tools, IP providers, and Generator providers.

<u>IP-XACT Parse Validity:</u>

- Parsing Correctness: Ability to read all IP-XACT XML files.
- Parsing Completeness: Cannot require information that can be expressed in an IP-XACT-format to be specified in a non-IP-XACT format. Processing of all information present in an IP-XACT description is not required.

<u>IP-XACT Description Validity:</u>

- Schema Correct: IP is described using XML files that pass XML-schema checks against The IP-XACT Consortium schema
- Usage Complete: Extensions to The IP-XACT schema can only express information that cannot otherwise be described in the non-extended IP-XACT XML..

<u>IP-XACT Semantic Validity:</u>

- Semantic Correctness: Must adhere to the IP-XACT Consortium semantic interpretations of IP-XACT XML data. the
- Semantic Completeness: Must pass all automated semantic checks provided by The SPIRIT Consortium. All semantic rules that are non-checkable in an automated way should be inspected through a regression-set of design examples.

The above validity rules can be combined with the product class specific rules to cover the full IP-XACT-enabled space. The following sections describe the rules a provider has to check to claim a product is IP-XACT-Enabled

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

2.3.1 Design Environment Provider

DE tools represent a framework environment to which point-tools, generators and IP libraries can be attached.

Requirements to be IP-XACT Enabled:

- Must follow the Parse, Description and Semantic Validity Requirements.
- Must be capable of read/write without loss of information expressed in as valid IP-XACT description, including the preservation of "so called" vendor extension data.
- Must support the IP-XACT generator interfaces fully for interaction with underlying database.

2.3.2 Point-tool Provider

Point tools have no generator support (i.e. no generator callable through IP-XACT generator interface).

Requirements to be IP-XACT Enabled:

- Must follow the Parse, Description and Semantic Validity Requirements.
- Must be capable of read/write without loss of information expressed in as valid IP-XACT description, including the preservation of "so called" vendor extension data.

2.3.3 IP provider

IP providers can generate static or configurable IP. Static IP providers do not need to supply attached generators. Configurable IP providers also need to provide generators callable through the IP-XACT generator interface. A configurable IP provider will therefore have follow the rules for both an IP provider and a Generator provider.

Requirements to be IP-XACT Enabled:

- Must follow the Parse, Description and Semantic Validity Requirements for static or generated IP.

2.3.4 Generator Provider

A generator provider delivers IP-XACT generators. IP-XACT generators can be packaged together with an IP or can be packaged separately, for example as a generator library.

Requirements to be IP-XACT Enabled:

- Must follow the Parse, Description and Semantic Validity Requirements..
- Must be capable of read/write without loss of information expressed in as valid IP-XACT description, including the preservation of "so called" vendor extension data.
- Must be callable through IP-XACT generator interface.

2.3.5 Support of other standards

IP-XACT compliant tools must also support:

- XML version 1.0 (http://www.w3.org/TR/2000/REC-xml-20001006)

© Copyright The SPIRIT Consortium 2006. All rights reserved.

- XML Schema (http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/, http://www.w3.org/TR/2004/PER-xmlschema-2-20040318/)
- XSLT version 1.0 (http://www.w3.org/TR/1999/REC-xslt-19991116)
- XPath version 1.0 (http://www.w3.org/TR/1999/REC-xpath-19991116)
- XPath version 2.0 (http://www.w3.org/TR/2005/CR-xpath20-20051103/) recommended, future versions of the IP-XACT specification are expected to require version 2.0.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

3 IP-XACT INTEROPERABILITY USE MODELS

To introduce the use-model for the IP-XACT specifications, it is first necessary to identify specific roles and responsibilities, and then relate these to how the IP-XACT specifications impact these. Note that all or some of the roles can be mixed into a single organization. For example, some EDA providers are also providing IP, a component IP provider can also be a platform provider and an IP system design provider may also be a consumer.

For this user-guide, the roles and responsibilities are restricted to the scope of IP-XACT v1.2, HDL system design.

3.1 Roles and responsibilities

3.1.1 The component IP Provider

This is a person, group or company creating IP components or subsystems to be integrated into a one-off SoC design or re-usable platform. These IP's can be hardware components (processors, memories, busses…), verification components and/or hardware-dependent software elements. These may be provided as source, or in a compiled form (i.e., simulation model). An IP is usually provided with a functional description, a timing description, some implementation or verification constraints, and some parameters to characterize (or configure) the IP. All these types of characterization data may be described in meta-data compliant with the IP-XACT Schema. Those elements not already provided in the base schema can be provided using name-space extensibility mechanisms of the specification.

The IP provider can use one or more EDA tools to create/refine/debug IP. During this process, the IP provider may wish to be able to export and re-import his design from one environment to another. The IP-XACT IP descriptions will enable this exchange for RTL component IP.

At some point this IP needs to be transferred to customers, partners and external EDA tool suppliers. IP-XACT compliant XML provides the exchange-format needed to describe these component IP packages exchanged between different companies and tool-flows / methodologies. IP can be characterized into different types:

Fixed IP: Is IP which is straightforward to describe and exchange as there are no configurable parameters. No configurators or generators need to be provided. An example of a fixed-IP is an APB GPIO block with a fixed base address.

Parameterized IP: Those IP block which don't need IP specific generators but have 'standard' customizations (where 'standard' is defined as industry de-facto tool support). i.e., No configurators or generators need be provided for SoC design tools which support these parameterizations. An example of a parameterized IP could be an AHB / APB bridge with configurable bus-widths.

Configurable IP: Created or modified as a direct result of running an IP specific generator to build the IP to the user's specified configuration. Generally will require both configurators (design capture) and generators (design build) to be provided with the IP. An example of a configurable IP is an AHB bus fabric component which has selectable number of masters and slaves, and automatic generation of decode functionality

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

3.1.2 The SoC Design IP Provider

This is a person, group or company that integrates and validates IP provided by one or more IP providers to build system platforms. These platforms are complete and validated systems or sub-systems.

Like the IP provider, the platform provider can use EDA tools to create/refine/debug its platform, but at some point IP needs to be exchanged with others (customers, partners, other EDA tools…). To achieve this operation the platform IP has to be expressed in the IP-XACT specified format as a hierarchical component.

3.1.3 The SoC Design IP Consumer

This is a person, group or company that configures and generates system applications based on platforms supplied by SoC Design IP providers. These platforms are complete system designs or sub-systems.

Like the platform provider, the platform consumer can use EDA tools to create/refine/debug its system application and/or configure the design architecture. To achieve this operation the EDA tool will have to support platform IP expressed in the IP-XACT specified format.

3.1.4 The Design Tool Supplier

This is a group or company that provides tools to verify and/or implement an IP or platform IP. There can be distinguished 3 major tools today (that could be combined) provided in a system flow:

- Platform builder (or SoC Design Environment) tools: help to assemble a platform with some automation (e.g. automatic generation of interconnect).
- Verification point-tools: functional and timing Simulation, Verification, Analysis, Debugging, Co-simulation, Co-verification, acceleration…
- Implementation point-tools: Synthesizer, Floorplanner, Placer/router…

The EDA provider will need to be able to import IP-XACT component or system IP libraries from multiple sources, and if needed could export them in the same format.

Further, IP-XACT EDA tools would be able to recognize, associate and launch configurators and generators that may be provided by a Generator or IP provider in support of configurable IP bundles. The imported IP could be created and/or modified by the tool and could be exported back (e.g. to be exchanged with other EDA vendor tools) to satisfy the customer design flow.

Further to the support of generators supplied with IP bundles, the IP-XACT DE tools will need to be able to recognize and interface with generator-wrapped point-tools. These may be provided by another EDA provider, or by the IP designer / consumer as part of a company's internal design and verification flow. In general, these will support specialized design-automation features such as architectural-rule checking, etc.…

3.2 IP-XACT IP Exchange Flows

This section describes a typical IP exchange flow that the IP-XACT specifications technically support between the roles defined in the previous section. By way of example, the following specific exchange flow can benefit from use of the IP-XACT specification:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

The Component IP provider generates IP-XACT XML package and can send it to a SoC design-tool (EDA tool supplier) or directly to a Platform (i.e. SoC Design IP) provider. The EDA tool supplier can import IP-XACT XML IP and generate platform IP and/or update (configure) the IP components. The Platform provider can generate a configurable platform IP and export it in IP-XACT XML format to be used by the end user to build system applications. The platform provider can also generate its own platform IP into IP-XACT format and send it to the EDA provider.

While we cannot cover here all the different possible IP exchange flows, we can identify the following three main use models, from the point of view of each user:

- IP (Component or SoC Design) provider use model
- Generator / configurator (IP provider and Design tool provider) use model
- SoC design-tool provider use model

3.2.1 Component or SoC design IP provider use model

The IP provider (hardware Component IP designer or a Platform IP architect) will use IP-XACT to package its IP in a standard and reusable format. The first step consists in creating an IP-XACT XML package (XML plus IP views) to export the IP database in a valid format. To express this IP as an IP-XACT IP, the IP provider will have to:

Parse the entire design file tree (composed of files of different types: HDL source files, datasheets, interfaces, parameters…) and convert it into an IP-XACT XML format. This can be a manual (directly edit IP-XACT compliant XML) or automated (scripts generating Schema compliant IP-XACT XML) step.

Provide a script (or indication: could just be unzip, untar) to its IP consumer (Platform provider or end user Platform consumer) to read this IP. This script could be more complex to read the XML file and convert it into a file tree (or a database) at the end user place. This script could be seen as an IP installer that could take parameters from the XML file.

Once the IP has been packaged in an IP-XACT format, the IP provider will use a SoC design-tool to write/debug/simulate/implement its IP.

3.2.2 Generator / configurator provider use model

The author of a generator or configurator expects to interact with the SoC design tool through a fixed interface and at well defined times in the design lifecycle. The well defined times are when components are instantiated, modified or when a generator chain is started.

As described earlier, Configurators are primarily responsible for ensuring that the customizable aspects of an IP are customized in a coherent and sensible manner. Generators are used within the SoC design-tool to extend its capabilities, maybe by wrapping a point tool, e.g. a simulator; or wiring up IP within the design, or checking that the design is correct or maybe modifying the design. Many of these features may also be handled by generators, embedded in the IP itself and supplied by the IP author.

So there are at least two groups of configurator/generator providers; the IP vendor, who will supply generators that are written specifically to support their IP, and generic generator authors who wish to extend the features available within the SoC design-tool. This latter group will be mainly the SoC-design tool vendors at first but will also come to include 3rd party generator vendors.

Both groups of providers will have to understand the special requirements that their generator has and pass this information to the SoC-design tool. The generator will,

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

through the LGI or TGI (see Chapter 5 for more details) and then write out any design changes that may have occurred as a result of the manipulations. Many generators will be read-only, they will have 'side-effects' that are useful, such as starting a simulator. Only a few are expected to actively change the design.

3.2.3 SoC Design-tool provider use model

The SoC design-tool will take as input an IP-XACT component or SoC design, configure it and load it into its own database format. Then it can automate some tasks such as the creation of the platform, the generation of the component interconnect, the generation of the bus fabric and as an output generate or update the IP-XACT IP. This would include providing new or updated XML with the attached information: new source files, parameters, documentation, etc.

Customer design flows are usually composed of a chain of different tools from the same or from different EDA vendors (for example when an EDA provider is not providing all the tool chain to cover the entire user flow or when the customer is selecting the best-in class point tool). To address this requirement, the EDA vendor providing an IP-XACT-enabled tool will read and produce the IP-XACT specified format and utilize and implement the interfaces defined by The SPIRIT Consortium. In this use model, each SoC design-tool will have its own generators (utilizing the IP-XACT LGI/TGI) to build, update and dump its internal meta-data state in an IP-XACT format that can be imported by another IP-XACT-enabled EDA tool.

Note that, in general, communication through an API or shared Database is not likely to happen between EDA competitors unless they agree to use an open access Database. This form of data-sharing is a more likely scenario either within the same company providing different design tools or between two EDA vendors which have a close partnership. In this scenario, the two SoC design tools would communicate through a shared database and bi-lateral agreement on API. This form of communication is not being specified by IP-XACT.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4 IP-XACT SCHEMA

In addition to the in-line documentation with the IP-XACT Schema [4], this section gives some explanation on how the different schema files link to each other, and when to use them.

4.1 IP-XACT objects

The IP-XACT schema is the core of the IP-XACT specification. It is important to first define the top objects manipulated by the Schema and describe the interaction between these objects.

4.1.1 Definitions

Prior to listing the different schema files and go deeper in the details, let's first define the objects used in the Schema and in the rest of this document. These definitions will be largely detailed in the next sections of this document.

IP-XACT **metadata** is a tool-interpretable way of describing the design history, locality, object association, configuration options, constraints against, and integration requirements of an IP object.

An IP-XACT IP appears as two distinct objects: the top design SoC object and the Component objects instantiated in the top design.

An IP-XACT **component** is the central placeholder for the object Meta data and its bus and generator interfaces. Components are used to describe Cores (processors, co-processors, DSPs…), Peripherals (Memories, DMA controllers, Timers, UART…) and Busses (simple buses, multi-layer buses, cross bars, network on chip…). An IP-XACT component can be of two kinds: static or configurable. A static component cannot be changed by a design environment. A configurable component has some parameters that can be configured by the DE and these parameters are also configurable in the RTL. Additionally an IP-XACT component can be a hierarchical object or a leaf object. Leaf components do not contain other IP-XACT IP. Hierarchical means that the IP-XACT description contains IP-XACT sub-components. Clearly this can be recursive thus having hierarchical IP that contain hierarchical IP, etc, thus, leading to the concept of hierarchy depth. Note that the IP being described may have a completely different hierarchical arrangement in terms of its implementation in RTL to that of its IP-XACT description. So a RTL description of a large IP component may be made up of many levels of hierarchy but its IP-XACT description need only be a leaf object because that completely describes the IP. On the other hand some IP can only be described in terms of a hierarchical IP-XACT description, no matter what the arrangement of the RTL hierarchy.

An IP-XACT component may contain a channel, which is a special IP-XACT object that can be used to describe multi-point connections between regular components, which may require some interface adaptation.

An IP-XACT **design** describes the component instances and the interconnection between these instances. The interconnection defines the point-to-point connection between two component Interfaces (for example between a processor interface and a bus interface).

An IP-XACT **generator** can be a leaf object or hierarchical (contain other generator objects) to build a chain of generators. Hierarchical generators are called a generatorChain. In this version of IP-XACT, the chain is executed in sequence (no parallelism is allowed) for the execution of generators. Hence, a hierarchical

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

generator is blocked until all its contained sub generators have completed their execution. A generator is invoked upon user request. There exist two execution modes for a generator: a read-only mode named generatorInvocation and a read-write mode named generatorUpdate. A specific type generator is called a configurator. A configurator is a leaf process that is automatically invoked upon creation of the object. But one can decide to deactivate the configuration. The configurators only apply to configurable objects.

An IP-XACT **bus definition** is an object that describes a bus interface (signal names, direction, width, usage...) and the constraints that apply to these signals. The IP-XACT user should consider the term 'Bus' here in its large chip interconnect sense.

4.1.2 Objects interactions

The objects defined above are those listed in the schema `index.xsd` file.

- Components
- Designs
- Bus definition
- Meta-data (PMD)
- Generators:
  - Loose/Tight Generator Invocation
  - Loose/Tight Generator changes
  - Loose/Tight Generator chain

The links (reference calls) between these objects is illustrated in the following figure. The arrows (A → B) illustrate a reference of object B from object A.

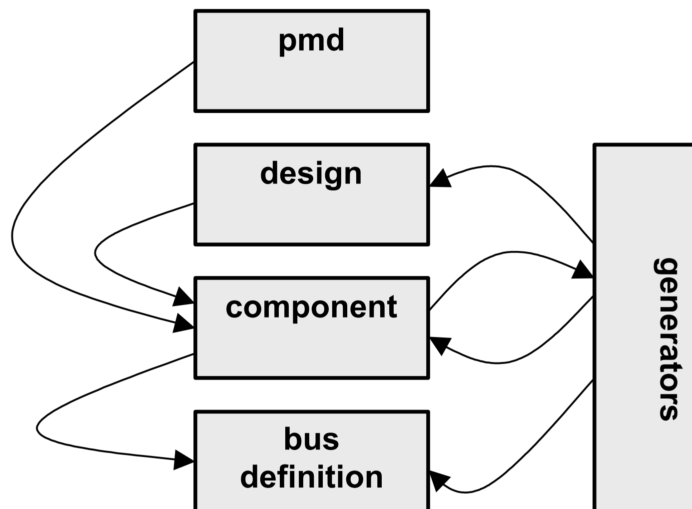


Figure 2 - IP-XACT Object interactions

In order to be uniquely referenced, each of these top objects has a unique identifier in IP-XACT, called VLNV.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.1.3 VLNV

Each IP-XACT object is assigned a **VLNV** (Vendor Library Name Version) that is defined in the header of each XML file. For example:

```
<spirit:vendor>spiritconsortium.org</spirit:vendor>
<spirit:library>Leon2</spirit:library>
<spirit:name>simple_design</spirit:name>
<spirit:version>1.0</spirit:version>
```

The version number (last V of the VLNV) assigned to an IP component by a manufacturer may be more complex than an integer number. The version number may appear as an alphanumeric string and may contain a set of substrings, with non-alphanumeric delimiters in-between. Each IP supplier will have their own cataloguing system for setting version numbers. The only rule for the version number is that it must change each time the XML file is changed.

Sorting and comparing the VLNV string is needed in IP-XACT to determine:

a) Whether an IP is a component that has been previously imported.
b) To allow multiple versions of the same IP to exist in a design.

To sort and compare the VLNV, we must subdivide the version number into major fields and subfields. Major fields may be separated by a non-alphanumeric delimiter such as `/`, `.`, `-`, `_`, `","`, etc. Each major field can be compared to determine equivalence, broken down further into subfields if necessary.

The following examples illustrate the sorting and comparing of VLNV.

The first case will use: `205/75WR16` and `215/50HR15`

Each of these version numbers break down into the following two major fields, separated by the "/" delimiter: `205 75WR16 215 50HR15`

Major fields are compared against each other from left to right. In this example, the first major fields (`205` and `215`) differ between the VLNV strings and our comparison can end there. This case is also simplified by the first major field being an integer (i.e., numeric).

Subfields, within each major field, will need to be examined if the major fields are alphanumeric. Each major field will have alphabetical and numerical subfields that are separated from right to left.

In the next example, we have two VLNV with the first major field being the same, so we must compare their second major subfields: e.g., `205/45R16` and `205/55R15`

The first major field (`205`) is equal between these two VLNV so the second major field is checked. These second major fields are broken down into the following alphabetic and numeric subfields: `45 R 16` and `55 R 15`

The subfields are compared from left to right. The first (and in this case only) comparison is `45` versus `55`, so these subfields are not equal. The major fields are not equivalent.

To summarize the rules for the comparison of each subfield in a major field:

- Numeric - compare the integer values of numeric subfields.
- Alphabetic -
  - String: perform a simple string comparison
  - Case: ignore alphabetic case (e.g., a-A are the same)

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Note that the `version` element is now mandatory in version 1.2 of the IP-XACT schema.

4.2 IP-XACT schema overview

The IP-XACT schema is composed of a set of main files representing the top elements (the root objects defined in the previous section) and sub files included from the main files.

A brief description of the main Schema files is given here after.

4.2.1 Design schema

This schema defines the way in which platform designs (top level designs) can be defined.

Typically a design will include instances of IP, both components and busses. There will also be the interconnections between these objects. Alongside the instance specific configuration information will be the platform configuration information, such as preferred RTL language, simulator of choice.

The design schema file is:

- design.xsd

4.2.2 PMD schema

The Platform Meta Data (PMD) schema defines the configurable parameters at the Design level. This schema also allows transforming a blackbox component (e.g. read-only) based on platform design constraints.

The PMD schema file is:

- pmd.xsd

4.2.3 Component schema

The component schema is the one that defines the description of an IP.

Typically an IP will define bus interfaces, memory maps, sub-instances, configuration information, file sets, signal lists, generators and configurators.

The component schema file is:

- component.xsd

4.2.4 Bus definition schema

Bus definitions must adhere to this schema. A bus definition is used to describe the pins that make up a bus and some expected values for signal widths and usage. For example the ADDR pins can be defined as carrying address information and be 16 bits wide. There’s also information on expected pin directions when the signal is on a master interface, a slave interface or a system interface.

The bus definition schema file is:

- busDefinition.xsd

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.2.5 Generators schemas

There are schema files to define how configurators and generators are to be described and how they should interact with the design environment. There are also schemas to describe the loose generator interface for generator invocation and generator modifications back to the design database.

The generator schema files are:

- generator.xsd
- configurator.xsd
- generatorChangeList.xsd
- looseGeneratorInvocation.xsd

4.3 IP-XACT design model

A design (or SoC platform) is the top netlist that contains all the instances and connections of the design. The following sections have to be defined:

- the VLNV of this IP (i.e. Vendor, Library, Name, Version)
- the component instances (e.g. core, peripherals, bus)
- the connections between the bus and the component instances

The best way to explain how to represent a platform design in IP-XACT is to illustrate using a simple example illustrated in the figure below.

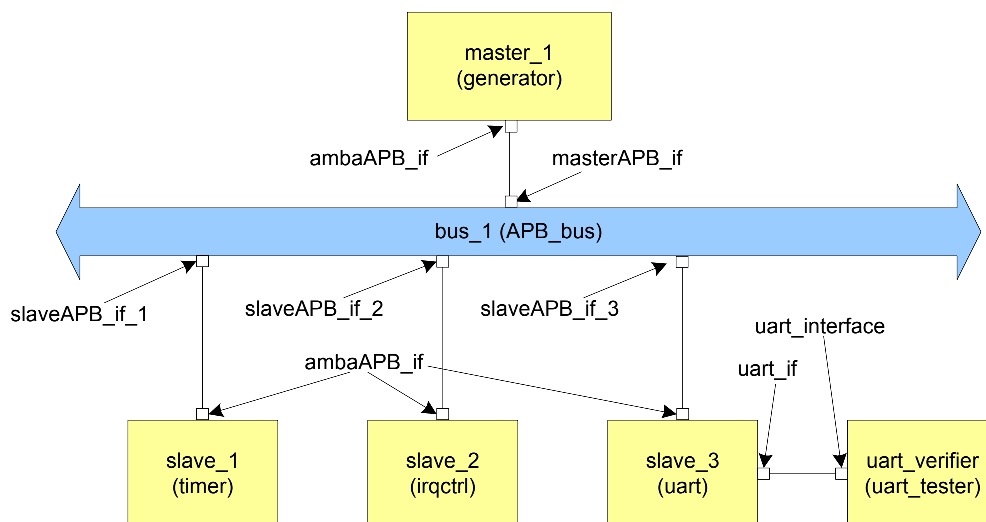


Figure 3 - Simple SoC Design example

The equivalent IP-XACT XML file for this simple design is described below. First the general outline of the XML file is displayed, and then each sub section is refined.

The design starts with the standard XML headers and includes the design's VLNV, there's then a list of components followed by a list of interconnections.

```
<?xml version="1.0" encoding="UTF-8" ?>
<spirit:design
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2/index.xsd">
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>simple_lib</spirit:library>
  <spirit:name>simple_design</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:componentInstances>
  <spirit:interconnections>
</spirit:design>
```

The component instances section (list of all the component instances in this design) is detailed here. In this example the first component instance is called *master_1* that is defined by an IP called *componentdef* from vendor *SPIRIT* in library *simple_lib*. Similarly there are three slave and a bus component instances (called *irqctrl_1*, *timer_1*, *uart_1* and *bus_1* that are defined respectively by the IP components *irqctrl*, *timer*, *uart, APB_bus*) from the same library. Configuration parameters are defined on the uart and the bus. There's also a component instance, *uart_verifier,* which is acting as verification object in this design.

```
<spirit:componentInstances>
  <spirit:componentInstance>
    <spirit:instanceName>master_1</spirit:instanceName>
    <spirit:componentRef
      spirit:vendor=""spiritconsortium.org"
      spirit:library="simple_lib"
      spirit:name="generator"
      spirit:version=""1.1" />
  </spirit:componentInstance>

  <spirit:componentInstance>
    <spirit:instanceName>slave_1</spirit:instanceName>
    <spirit:componentRef
      spirit:vendor="spiritconsortium.org"
      spirit:library="Leon2"
      spirit:name="timer"
      spirit:version="1.03" />
  </spirit:componentInstance>

  <spirit:componentInstance>
    <spirit:instanceName>slave_2</spirit:instanceName>
    <spirit:componentRef
      spirit:vendor="spiritconsortium.org"
      spirit:library="Leon2"
      spirit:name="irqctrl"
      spirit:version="1.00" />
  </spirit:componentInstance>

  <spirit:componentInstance>
    <spirit:instanceName>slave_3</spirit:instanceName>
    <spirit:componentRef
      spirit:vendor="spiritconsortium.org"
      spirit:library="Leon2"
      spirit:name="uart"
      spirit:version="1.00" />
    <spirit:configuration>
      <spirit:configurableElement
spirit:referenceId="EXTBAUD">false</spirit:configurableElement>
     </spirit:configuration>
  </spirit:componentInstance>

  <spirit:componentInstance>
    <spirit:instanceName>uart_verifier</spirit:instanceName>
    <spirit:componentRef
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
      spirit:vendor="spiritconsortium.org"
      spirit:library="vfcn"
      spirit:name="uart_tester"
      spirit:version="1.00" />
  </spirit:componentInstance>

  <spirit:componentInstance>
    <spirit:instanceName>bus_1</spirit:instanceName>
    <spirit:componentRef
      spirit:vendor="amba.com"
      spirit:library="Leon2 "
      spirit:name="APB_bus"
      spirit:version="1.1" />
    <spirit:configuration>
      <spirit:configurableElement spirit:referenceId="address_1">
0x400</spirit:configurableElement>
     <spirit:configurableElement spirit:referenceId="address_2">
0x800</spirit:configurableElement>
     <spirit:configurableElement spirit:referenceId="address_3">
0x1200</spirit:configurableElement>
    </spirit:configuration>
  </spirit:componentInstance>
</spirit:componentInstances>
```

Note that configurable information for an instance is only recorded in the design file if it has changed from the default value that was declared in the component’s definition file. The change may have been made by the user or by a generator.

The interconnection section is detailed here. It describes the bus instance (*bus_1*) and the connections to its master (*master_1*) and its slaves (*slave_1, slave_2* and *slave_3*). The AMBA APB bus here has one master interface (*masterAPB_if_1*) and three slave interfaces (*slaveAPB_if_1, slaveAPB_if_2, slaveAPB_if_2*). The verification object instance, *uart_verifier*, is connected using a monitor interface connection to the uart instance.

```
<spirit:interconnections>
  <spirit:interconnection>
    <spirit:activeInterface    spirit:componentRef="bus_1"
    spirit:busRef="masterAPB_if_1" />
    <spirit:activeInterface     spirit:componentRef="master_1"
    spirit:busRef="ambaAPB_if" />
  </spirit:interconnection>
  <spirit:interconnection>
    <spirit:activeInterface
      spirit:componentRef="bus_1"
    spirit:busRef="slaveAPB_if_1" />
    <spirit:activeInterface
    spirit:componentRef="slave_1"
    spirit:busRef="ambaAPB_if" />
  </spirit:interconnection>
  <spirit:interconnection>
    <spirit:activeInterface
      spirit:componentRef="bus_1"
    spirit:busRef="slaveAPB_if_2" />
    <spirit:activeInterface
    spirit:componentRef="slave_2"
    spirit:Ref="ambaAPB_if" />
  </spirit:interconnection>
  <spirit:interconnection>
    <spirit:activeInterface
```

 © Copyright The SPIRIT Consortium 2006. All rights reserved.

```
          spirit:componentRef="bus_1"
         spirit:busRef="slaveAPB_if_3"
        <spirit:activeInterface
         spirit:componentRef="slave_3"
        spirit:busRef="ambaAPB_if" />  </spirit:interconnection>

      <spirit: monitorInterconnection>
        <spirit:activeInterface spirit:busRef="uart_if"
          spirit:componentRef="slave_3"/>
        <spirit:monitorInterface spirit:busRef="uart_interface"
          spirit:componentRef="uart_verifier"/>
      </spirit:monitorInterconnection>
    </spirit:interconnections>
```

The `monitorInterconnection` (see section 4.9.1.3 for more details) element allows multiple verification objects to be connected to a single component bus interface. This allows verification IP to be introduced and removed from a design without changing the connectivity of the design itself.

Design interconnections (interConnections between active interfaces and monitorInterconnections between active and monitor interfaces) can be given a name. This is illustrated in the following figure below.

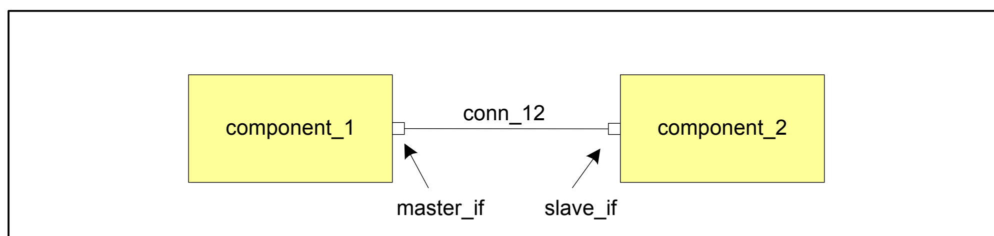


Figure 4 - Connectivity Name Example

The following XML fragment gives an example:

```
<spirit:interConnections>
   <spirit:interConnection>
      <spirit:name>conn_12</spirit:name>
      <spirit:activeInterface
             spirit:componentRef="component_1"
             spirit:busRef="master_if"/>
      <spirit:activeInterface
             spirit:component2Ref="component_2"
             spirit:busInterface2Ref="slave_if"/>
   </spirit:interConnection>
</spirit:interConnections>
```

This fragment illustrates the connectivity between bus interface *master_if* (on component_1) and the bus interface *slave_if* (on component_2). The Design Environment (or the user) can name this connection (e.g. *conn_12*). This name is optional, but if defined, it must be unique inside the design.

In SPIRIT 1.2 there is the concept of hierarchical connectivity expressed in the design file.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

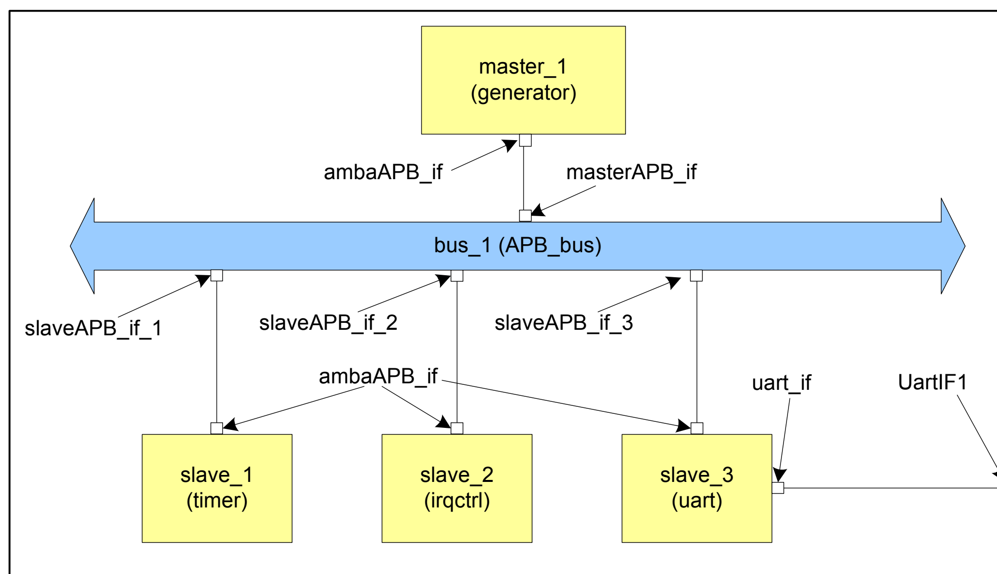


Figure 5 - Hierarchical Connectivity Example

The following XML fragment gives an example:

```
<spirit:hierConnections>
  <spirit:hierConnection spirit:interfaceName="UartIF1">
  <spirit:componentRef>slave_3</spirit:componentRef>
  <spirit:interfaceRef>uart_if</spirit:interfaceRef>
</spirit:hierConnections>
```

This fragment illustrates the connectivity between bus interface *UartIF1* (on the component that is being described by this design) and the bus interface *uart_if* on the UART instance *slave_3* in the design show in Figure 5. It is the responsibility of the design environment to ensure that the interface *UartIF1* exists on the component when inserting the design file into the component.

4.4 SPIRIT configuration

IP-XACT 1.2 includes a schema for documents that store design configuration information. This information is all the configurable information that is not recorded in the design file. The design configuration information is useful when transporting designs between design environments; it contains information that would otherwise have to be re-entered by the designer. It is important to make the distinction that the design itself contains **all** the information regarding configuration of the design, e.g. instance base addresses. The design configuration file contains non-essential ancilliary information.

A design configuration applies to a single design but a design may have multiple configuration files.

The information recorded in a configuration file is:

- Configurable information defined in pmd files
- Configurable information defined in generators and generator chains
- The active, or current, view selected for instances in the design
- Configurable information defined in vendor extensions

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

A brief example is:

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:designConfiguration
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
index.xsd">
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Library</spirit:library>
  <spirit:name>Configs</spirit:name>
  <spirit:version>1.0</spirit:version>

  <spirit:designRef spirit:vendor="spiritconsortium.org"
spirit:library="DesignLibrary" spirit:name="Design1"
spirit:version="1.0"/>
  <spirit:generatorChainConfiguration>
    <spirit:generatorChainRef spirit:vendor="spiritconsortium.org"
spirit:library="generatorLibrary" spirit:name="generator1"
spirit:version="1.0"/>
    <spirit:generators>
      <spirit:generatorName>gen1</spirit:generatorName>
      <spirit:configurableElement spirit:format="string" spirit:id="ID01"
spirit:referenceId="tmpDir" spirit:prompt="Temp dir name:">
        <spirit:configurableElementValue>
my_temp_dir</spirit:configurableElementValue>
      </spirit:configurableElement>
    </spirit:generators>
  </spirit:generatorChainConfiguration>
  <spirit:viewConfiguration>
    <spirit:instanceName>instance_1</spirit:instanceName>
    <spirit:viewName>verilog</spirit:viewName>
  </spirit:viewConfiguration>
  <spirit:vendorExtensions/>
</spirit:designConfiguration>
```

The example shows that the generator *gen1* has been configured to use a certain location as its temporary directory and that the verilog view for instance *instance_1* has been selected.

4.5 SPIRIT platform meta data (pmd) model [Depricated]

The IP-XACT schema allows certain platform-level rules to be encapsulated in XML. The need for this can be summarized by

- IP as delivered by IP vendors normally provides read-only access
- Configurable parameters of an IP may need to be presented in a way that best matches the needs of the platform rather than the supplier's original intention.
- Default signal values and drivers may need to be defined at the platform level without needing to change delivered XML

One way of transforming XML from one view to another is via the use of XSLT transformations and style sheets.

Take an example of a situation where a transformation is to only be applied if two specific IP blocks (such as a UART_3106 and ARM 926) are both present in the design. This could be described using the following XML syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
<spirit:pmd
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation=" http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2/index.xsd">
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>pmd</spirit:library>
  <spirit:name>UART_3106</spirit:name>
  <spirit:version>1.1</spirit:version>

  <spirit:appliesTo>
    <spirit:componentRef spirit:vendor="philips.com"
       spirit:library="SPIRIT_PS"
       spirit:name="UART_3106" <spirit:version="1.1"/>
    </spirit:appliesTo>

    <spirit:dependsOn>
      <spirit:componentRef spirit:vendor="mentor.com"
        spirit:library="PxArm9"
        spirit:name="a926" <spirit:version="1.1"/>
    </spirit:dependsOn>

    <spirit:transformer>
      <spirit:xslt>
        <spirit:styleSheet>src/clk.xsl</spirit:styleSheet>
      </spirit:xslt>
    </spirit:transformer>
</spirit:pmd>
```

4.5.1 XSL Stylesheet

The design environment is required to execute the XSL stylesheet. The actual transformation in the example is to add a specific clock driving waveform to an internal clock signal on the ip_3106 IP block. This is not normally delivered with the IP but always needs to be configured whenever this IP block is used in the target platform. The actual signal to be configured is called `u_clk` and can be found with an XPATH expression to the relevant part of the schema, namely via the spirit elements `model/signalList/signal/name`. A complete XSL example for creating the new XML is given below.

```
<xsl:stylesheet version="1.0"
 xmlns:xsl=http://www.w3.org/1999/XSL/Transform
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2">
   <xsl:param name="schemaNamespace">
     http://www.w3.org/2001/XMLSchema-instance</xsl:param>
   <xsl:param name="targetNamespace">
     http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2</xsl:param>
   <xsl:param name="schemaLocation">
     http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2</xsl:param>

   <xsl:strip-space elements="*"/>

   <!-- matches the root node from where to apply template rules -->
   <xsl:template match="/">
     <xsl:apply-templates/>
   </xsl:template>

   <!-- matches all attributes and nodes -->
   <xsl:template match="@*|node()">
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
      <xsl:copy>
        <xsl:apply-templates select="@*"/>
        <xsl:apply-templates/>
      </xsl:copy>
   </xsl:template>

   <!-- add a clock driver to the u_clk port and export to top-level -->
   <xsl:template
match="spirit:model/spirit:signalList/spirit:signal[spirit:name='u_clk']"
>
     <spirit:signal>
       <spirit:name>u_clk</spirit:name>
       <spirit:direction>in</spirit:direction>
       <spirit:clockDriver>
       <spirit:clockPeriod>40</spirit:clockPeriod>
       <spirit:clockPulseOffset>10</spirit:clockPulseOffset>
       <spirit:clockPulseValue>0</spirit:clockPulseValue>
       <spirit:clockPulseDuration>20
       </spirit:clockPulseDuration>
       </spirit:clockDriver>
       <spirit:export spirit:configGroups="export"
          spirit:id="sig_u_clk"
          spirit:prompt="u_clk">true
       </spirit:export>
     </spirit:signal>
   </xsl:template>

</xsl:stylesheet>
```

In addition to enhancing the clock signal with a driving waveform the `u_clk` signal has been tagged to be exportable to the top-level of the design so that it can be driven from a high-level.

4.6 SPIRIT component model

A component contains the following main sections:

- The VLNV of this IP (i.e. Vendor, Library, Name, Version). This is the only mandatory element.
- The Bus interfaces: see section 4.6.1 for more details.
- The HW model (i.e. signal names, verification environments), see section 4.6.8 for more details.
- The file sets (e.g. source files, libraries): see section 4.6.10 for more details.
- The interconnections between sub component instances (if the component is hierarchical), see section 4.1.1 for more details.
- The sub component instances (if the component is hierarchical), see section 4.1.1 for more details.
- The channels (if this component is a channel), see section 4.9.3.1 for more details.
- The address spaces (if the component is a bus master), see section 4.6.4 for more details.
- The generators and configurators attached to the component, see section 5 for more details.
- The implementation constraints associated with the component, see section 4.6.9 for more details.

Below is an Example of a component (Leon Timer peripheral) in XML format:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
<?xml version="1.0" encoding="UTF-8" ?>

<spirit:component
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.
2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.
2 http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2/index.xsd">

  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Leon2</spirit:library>
  <spirit:name>timers</spirit:name>
  <spirit:version>1.00</spirit:version>
  <spirit:busInterfaces/>
  <spirit:memoryMaps/>
  <spirit:model/>
  <spirit:choices/>
  <spirit:fileSets/>
…

</spirit:component>
```

The next sections will describe the above elements in more detail.

4.6.1 Component interfaces

These are defined in the `busInterfaces` element of the component. It contains a list of interfaces (e.g. bus interfaces, interrupt interfaces). Here follows an example of a Timer slave component with an AMBA APB interface.

```
<spirit:busInterfaces>
  <spirit:busInterface>
    <spirit:name>ambaAPB</spirit:name>
    <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA"
      spirit:name="APB" spirit:version="v1.0" />

    <spirit:slave>
      <spirit:memoryMapRef spirit:memoryMapRef="slaveMMap"/>
    </spirit:slave>

    <spirit:connection>required</spirit:connection>

    <spirit:signalMap>
      <spirit:signalName>
        <spirit:busSignalName>PCLK<spirit:busSignalName>
        <spirit:componentSignalName>clk<spirit:componentSignalName>
      </spirit:signalName>
      <spirit:signalName>
        <spirit:busSignalName>PWDATA<spirit:busSignalName>
        <spirit:componentSignalName>pwdata<spirit:componentSignalName>
        <spirit:left>31<spirit:left>
        <spirit:right>0<spirit:right>
      </spirit:signalName>
      <spirit:signalName>
        <spirit:busSignalName>PADDR<spirit:busSignalName>
        <spirit:componentSignalName>paddr<spirit:componentSignalName>
        <spirit:left>7<spirit:left>
        <spirit:right>2<spirit:right>
      </spirit:signalName>
    </spirit:signalMap>

  </spirit:busInterface>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
</spirit:busInterfaces>
```

For a slave component, the `busInterface` section may contain a reference to the `memoryMap` element defined in the component. Here is an example of a memory map for a Timer component. To simplify, only one register (the timer1Counter) is displayed.

```
<spirit:memoryMaps>
  <spirit:memoryMap>
    <spirit:name>slaveMMap</spirit:name>
    <spirit:addressBlock>
      <spirit:baseAddress spirit:format="long">0</spirit:baseAddress>
      <spirit:range spirit:format="long">64</spirit:range>
      <spirit:width spirit:format="long">32</spirit:width>

       <spirit:register>
         <spirit:name>timer1Counter</spirit:name>
         <spirit:addressOffset>0x0</spirit:addressOffset>
         <spirit:size>32</spirit:size>
         <spirit:access>read-write</spirit:access>
      </spirit:register>

    </spirit:addressBlock>
  </spirit:memoryMap>
</spirit:memoryMaps>
```

The `busInterface` element defines the list of signals of each interface. The list is contained in the `signalMap` element. Each logical bus signal represented by the SPIRIT element `busSignalName` (`PCLK` in this example) must be mapped to a physical pin signal represented by the IP-XACT element `componentSignalName` (`clk` in this example).

```
    <spirit:signalMap>
      <spirit:signalName>
        <spirit:busSignalName>PCLK<spirit:busSignalName>
        <spirit:componentSignalName>clk<spirit:componentSignalName>
      </spirit:signalName>
      <spirit:signalName>
        <spirit:busSignalName>PWDATA<spirit:busSignalName>
        <spirit:componentSignalName>pwdata<spirit:componentSignalName>
        <spirit:left>31<spirit:left>
        <spirit:right>0<spirit:right>
      </spirit:signalName>
      <spirit:signalName>
        <spirit:busSignalName>PADDR<spirit:busSignalName>
        <spirit:componentSignalName>paddr<spirit:componentSignalName>
        <spirit:left>7<spirit:left>
        <spirit:right>2<spirit:right>
      </spirit:signalName>
    </spirit:signalMap>

  </spirit:busInterface>
</spirit:busInterfaces>
```

4.6.2 Whitebox interfaces

Internal elements of a component can be accessed in the design environment by other components, in particular verification components, through whitebox interfaces. These are listed in the `whiteboxElements` section of the component. Each whitebox element has a name, a type, a driveable flag, and a string description.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

The `whiteboxType` element indicates the type of the element. The `pin` and `signal` types refer to elements within an HDL description. The `register` type refers to a register in the memory map. The `interface` type refers to a bus interface in a lower level component definition. In the example below, a status flag is shown that is not visible at the interface of a component, but may be needed for debug, for probing by a testbench component or for use in an assertion.

```
<spirit:whiteboxElements>
  <spirit:whiteboxElement>
    <spirit:name>Name</spirit:name>
    <spirit:whiteboxType>register</spirit:whiteboxType>
    <spirit:driveable>false</spirit:driveable>
    <spirit:description>
       error in most recent transfer
    </spirit:description>
  </spirit:whiteboxElement>
</spirit:whiteboxElements>
```

Connections to whitebox elements are not specified in the SPIRIT design file. The `whiteboxElement` element informs the design environment what internal elements are accessible, and the design environment itself has to make any connections. The details of how the whitebox element is implemented, and the path to it, are contained in the `view` section of the `model` element. The same `whiteboxElement` element may be implemented differently in different views.

4.6.3 Component choices

The `choice` element contains the list of items used by a `modelparameter` or `parameter` element. These elements indicate that it will use a `choice` element by setting the attribute `format="`**`choice`**`"`. The element must also define the attribute `choiceRef="`**`widthOptions`**`"` to pick which choice list to use. The following example shows the addressable size (width) and the word size (Dwidth) of a memory component.

```
<spirit:model>
  <spirit:modelparameters>
    <spirit:modelparameter spirit:name="width" spirit:format="choice"
spirit:choiceRef="widthOptions">1</spirit:modelparameter>
    <spirit:modelparameter spirit:name="Dwidth" spirit:format="choice"
spirit:choiceRef="DwidthOptions">4</spirit:modelparameter>
  </spirit:modelparameters>
</spirit:model>

<spirit:choices>
  <spirit:choice>
    <spirit:name>widthOptions</spirit:name>
    <spirit:enumeration spirit:text="8K">1</spirit:enumeration>
    <spirit:enumeration spirit:text="64K">2</spirit:enumeration>
    <spirit:enumeration spirit:text="256K">3</spirit:enumeration>
  </spirit:choice>
  <spirit:choice>
    <spirit:name>DwidthOptions</spirit:name>
    <spirit:enumeration spirit:text="2Bytes">4</spirit:enumeration>
    <spirit:enumeration spirit:text="4Bytes">5</spirit:enumeration>
    <spirit:enumeration spirit:text="8Bytes">6</spirit:enumeration>
  </spirit:choice>
</spirit:choices>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.6.4 Component Address space

The logical address spaces for each Master interface can be defined for each component. The addressSpace defines the logical addressable space as seen by each master. The different address spaces for a component are defined under the addressSpace element. This addressSpace element is the referenced in each of the component Master interfaces.

Here is an example of addressSpace definition for a component with two master interfaces. One interface has an addressSpace of 1 GB (and a based address of 0x00000) and the other interface has an addressSpace of 2 GB (and a base address of 0x10000). Note that the addressSpaceRef attribute is optional.

```
<spirit:component>
…
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:master>
              …
       <spirit:addressSpaceRef spirit:addressSpaceRef="Memory_M1"/>
        <spirit:baseAddress>0x00000</spirit:baseAddress>
      </spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:master>
      …
       <spirit:addressSpaceRef spirit:addressSpaceRef="Memory_M2"/>
          <spirit:baseAddress>0x10000</spirit:baseAddress>
        </spirit:addressSpaceRef>
      </spirit:master>
    </spirit:busInterface>
  </spirit:busInterfaces>
  …
  <spirit:addressSpaces>
    <spirit:addressSpace>
      <spirit:name>Memory_M1</spirit:name>
      <spirit:range>1Gb</spirit:range>
    </spirit:addressSpace>
    <spirit:addressSpace>
      <spirit:name>Memory_M2</spirit:name>
      <spirit:range>2Gb</spirit:range>
    </spirit:addressSpace>
  </spirit:addressSpaces>

</spirit:component>
```

AddressSpaces are effectively the programmers view looking out from a master port. Some components may have addressSpaces associated with more that one master interface (for instance, a processor that has a system bus and a fast memory bus. Other components (for instance, Harvard architecture processors) may have multiple addressSpaces – one for instruction and the other for data.

The addressSpace view seen by different masters connected to the same channel will depend on the individual architectures of the component implementing the master. So addressSpaces provide a very ‘personal’ view of the world from that point in the design.

As an example, we can think of a UART with an 8-bit data bus and 1K of registers connected via a channel to two processors – one with an 8-bit architecture and one with a 32-bit architecture (see section 4.9.3.1 for more details on channels). Depending on how the channel is implemented, the 8-bit processor will see the

© Copyright The SPIRIT Consortium 2006. All rights reserved.

UART occupying 1K of its addressSpace, while the 32-bit processor may see the same UART occupy 4K of its addressSpace (but with 3 out of every 4 addresses unoccupied).

The exact configuration depends on the addressing and byte steering capabilities implemented by the channel. The bitsInLau indicates minimum size of a data transaction supported in an interface, and this may be used to determine the appropriate address signal alignment (the two least significant bits of the address bus of a byte-capable 32-bit processor may remain unconnected when linked to a peripheral which is capable of doubleword transactions only [bitsInLau=32]) and required byte steering.

The addressSpace seen by a master on one bus may contribute to a different addressSpace seen by a master on another bus if the first address space appears in a bridged slave interface that is connected to the second bus. Bus bridges are the constructs that link addressSpaces across different buses (see section 4.9.3.3 for more details on bridges).

4.6.4.1 Endianness

Endianness is defined under the addressSpace element of the component. In the current Schema there exist (only) two legal values ('big' and 'little') to specify the endianess:

- Big endian: means that the most significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field.
- Little endian: means that the least significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field.

But there are indeed at least two ways for big-endianness to manifest itself, byte invariant and word invariant (also known as middle-endian).
The difference being if data is stored as word invariant then for transfers larger than a byte the data is stored differently, e.g:

- Word invariant: A word access to address 0x0 is on D[31:0]. The MS byte will be on D[31:24], the LS byte will be on D[7:0].
- Byte invariant: A word access to address 0x0 is on D[31:0]. The MS byte will be on D[7:0], the LS byte will be on D[31:24].

In IP-XACT, the interpretation of 'big' is the byte invariant style. However, if there is a need to model middle-endian, a workaround is to extend the addressSpace element via ##other or to use a parameter for those cases where endianness model cannot be represented.
The above discussion is byte centric as that is the most common LAU. However, the discussion in general applies to any size LAU.

4.6.4.2 Local Memory map

Some processor components require specifying their local memory map. This can be done under the addressSpace element of the component.

- Local memory maps (localMemoryMap element) are blocks of memory within a component that can only be accessed by the master interfaces of that component. The XML is identical to standard memoryMap types as would be defined by a slave interface (See next section).

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.6.5 Component Memory Map

The memory map can be defined for each Slave interface of a component. The memory map must be defined at the top of the component, under the memoryMap element. This memory map is then referenced in each component Slave interface.

Here is a sample of a memoryMap element definition for a simple memory (with address map 0x0000 to 0x0FFF). Only the name (my_memory), the baseAddress element (0x0000) and the range element (memory size) are mandatory. Other parameters are optional.

Note that the memoryMapRef attribute on the slave interface is mandatory if and only if at least one signal with an isAddress element is connected in the busInterface element.

```
<spirit:component>
…
<spirit:memoryMaps>
  <spirit:memoryMap>
    <spirit:name>my_memory</spirit:name>
    <spirit:addressBlock>
      <spirit:baseAddress spirit:format="long">0x0000
      </spirit:baseAddress>
      <spirit:bitOffset spirit:format="long">0</spirit:bitOffset>
      <spirit:range spirit:format="long">4096</spirit:range>
      <spirit:usage>memory</spirit:usage>
      <spirit:access>read-write</spirit:access>
    </spirit:addressBlock>
  </spirit:memoryMap>
</spirit:memoryMaps>
…
<spirit:busInterfaces>
  <spirit:busInterface>
    <spirit:slave>
      <spirit:memoryMapRef spirit:memoryMapRef="my_memory"/>
    </spirit:slave>
  </spirit:busInterface>
</spirit:busInterfaces>
…
</spirit:component>
```

In the previous example, the baseAddress element was hard coded. The following example shows how to specify the baseAddress element that is resolved by a user or dependent on other baseAddress elements. The first memoryMap element, mmap, specifies the baseAddress element as user resolved by including the attribute resolve and assigning it a value of “user”. To be able to reference the value of the baseAddress element and the range element elsewhere, the attribute id is also included in each of these elements.

The baseAddress of the second memoryMap, dependent_mmap, is dependent on the baseAddress and range of the first memoryMap. This dependency is specified by using the resolve attribute and assigning it value “dependent” and using this value necessitates including attribute dependency whose value is of the type string and can be any XPATH 1.0 expression. In the example, the XPATH expression references the values of baseAddress and range of the previous memory map to compute the dependent baseAddress. IP-XACT defines four commonly used functions that can be used as part of XPATH expressions; the example below shows three of them being used. Their definition follows the example.

© Copyright The SPIRIT Consortium 2006. All rights reserved.

All dependent expressions may only contain references to fixed tags, or tags with the attribute ‘`resolve="user"`’. This restriction prevents circular expression dependencies being accidentally created, but may require the same or similar XPATH expressions to be repeated more that once in the same XML file. For instance, the third `memoryMap`, which is dependent on the second map, violates this IP-XACT semantics.

In addition, to reduce the dependence on a particular IP-XACT schema version, the XPATH expressions should only reference elements using element id values. They should not use absolute or relative XPATH expressions to navigate to an element. If, however, an absolute expression is used, then the XPATH context for the interpreting this expression will be the root of the document containing the expression. A further restriction in IP-XACT is to not allow values of two elements to have the same `id`

To configure two elements through a single user defined parameter, one must define one element as user defined and make the other dependent on it with a formula that retrieves its value.

To compute two elements with the same value, they must both have the same `dependency` formula and to compute two elements with one depending on the other, the formula in the dependent element must include the formula of the depended element and not just its `id` () reference.

```
<spirit:memoryMaps>
  <spirit:memoryMap>
    <spirit:name>mmap</spirit:name>
    <spirit:addressBlock>
      <spirit:baseAddress  spirit:resolve="user"
spirit:id="baseAddress">0</spirit:baseAddress>
      <spirit:bitOffset>0</spirit:bitOffset>
      <spirit:range spirit:id="range">786432</spirit:range>
      <spirit:width>32</spirit:width>
      <spirit:usage>memory</spirit:usage>
      <spirit:access>read-write</spirit:access>
    </spirit:addressBlock>
  </spirit:memoryMap>

  <spirit:memoryMap>
    <spirit:name>dependent_mmap</spirit:name>
    <spirit:addressBlock>

<!-- The baseAddress in this memoryMap is dependent on the previous
memory map and the formula to compute the baseAddress from the
baseAddress of previous map is expressed as an XPATH expression -->

      <spirit:baseAddress  spirit:resolve="dependent"
spirit:dependency="spirit:pow(2,spirit:log(2,
spirit:decode(id('baseAddress'))+ spirit:decode(id('range')))+1)"
spirit:id="dependentBaseAddress">0</spirit:baseAddress>
      <spirit:bitOffset>0</spirit:bitOffset>
      <spirit:range>4096</spirit:range>
      <spirit:width>32</spirit:width>
      <spirit:usage>register</spirit:usage>
      <spirit:access>read-write</spirit:access>
    </spirit:addressBlock>
  </spirit:memoryMap>

<!-- The following memoryMap is illegal because the dependent formula
used to calculate baseAddress is dependent on the dependent element
(id('dependentBaseAddress')) -->
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
    <spirit:memoryMap>
      <spirit:name>illegal_dependent_mmap</spirit:name>
      <spirit:addressBlock>
      <spirit:baseAddress  spirit:resolve="dependent" spirit:dependency="
  spirit:pow(2,spirit:log(2, spirit:decode(id('dependentBaseAddress'))+
  spirit:decode(id('range')))+1)">0</spirit:baseAddress>
        <spirit:bitOffset>0</spirit:bitOffset>
        <spirit:range>4096</spirit:range>
        <spirit:width>32</spirit:width>
        <spirit:usage>register</spirit:usage>
        <spirit:access>read-write</spirit:access>
      </spirit:addressBlock>
    </spirit:memoryMap>
  </spirit:memoryMaps>
```

In addition to the standard XPATH 1.0 functions, SPIRIT defines four extra functions to aid expressions calculations.

4.6.5.1 spirit:containsToken

boolean spirit:containsToken(string, string)

The `spirit:containsToken` function returns true if the first argument string contains the second argument string as a token and otherwise returns false. To be interpreted as a token, the second string must be found within the first string, and be separated by white space from any other characters in the first string that are not white space characters.

Example: `spirit:containsToken('default spine driver','pin')` evaluates to false, whereas the standard XPath function contains would have evaluated true with the same arguments.

Purpose: Some attributes in SPIRIT are a list of tokens separated by white space. This function allows XPath selection based on whether the attribute contains a specific token.

4.6.5.2 spirit:decode

number spirit:decode(string)

The `spirit:decode` function decodes the string argument to a number and returns the number, or returns the NaN number if the string cannot be decoded. If the argument is omitted, it defaults to the context node converted to a string. If the string argument is a decimal formatted number, it is returned unchanged. If it is a hexadecimal representation starting with “0x” or “#”, it is converted to a decimal number and returned. If it is in engineering notation ending in a ‘k’, ‘m’, ‘g’, or ‘t’ suffix, case-insensitive, the numeric part is multiplied by the appropriate power of two.

Example: `spirit:decode('0x4000')` evaluates to 16384. `spirit:decode('4G')` evaluates to 4294967296.

Purpose: SPIRIT allows numbers to be expressed in hexadecimal format and engineering format. When setting up dependencies on configurable values, it is sometimes necessary to perform some arithmetic in the dependency XPath expression. However, XPath only supports arithmetic on numbers and it only recognizes decimal strings as numbers. This function allows the alternate formats to be converted to numbers recognizable by XPath.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.6.5.3 spirit:pow

number spirit:pow(number, number)

The `spirit:pow` function returns a number which is the first argument raised to the power of the second argument.

Example: `spirit:pow(2, 10)` evaluates to 1024.

Purpose: It is common for a component to have a configurable number of address bits. When this happens, the size of the address range it occupies on a memory map varies exponentially with the number of address bits. This function gives XPath the mathematical capabilities needed to describe this relationship in a dependency expression.

4.6.5.4 spirit:log

number spirit:log(number, number)

The `spirit:log` function returns a number that is the log of the second argument in the base of the first argument.

Example: `spirit:log(2, 1024)` evaluates to 10.

Purpose: This is the inverse of the `spirit:pow` function. It is intended to express the reverse of the dependency described for the `spirit:pow` function. In this case, the range of an address block might be configurable and the number of address bits might be expressed as a dependency of the address range using the log function.

A more detailed example is given in Appendix section: Describing the bus interfaces

4.6.6 Memory bank

Banks allow multiple address banks to be grouped into a single address block. If it's a parallel bank then each address block under the bank is understood to be located at the same address with different bit offsets to accommodate the widths of the preceding blocks.

If it's a serial bank then the first block under the bank is understood to be located at the bank's address, the next block is at the bank's address plus length of the first block (adjusted for LAU and bus width considerations).

This means that it is possible to specify one address for a bank of memory, and the members of the bank all line up correctly without an excessive number of `resolve="dependent"` expressions.

In the following example XML, the only address specified is 0x10000, but it causes memories ram0, ram1, ram2 and ram3 to be mapped to addresses 0x10000, 0x14000, 0x18000 and 0x1C000 respectively.

```
<bank bankAlignment="serial">
    <baseAddress>0x10000</baseAddress>
    <addressBlock>
      <range>0x1000</range>
      <width>32</width>
    </addressBlock>
    <addressBlock>
      <range>0x1000</range>
      <width>32</width>
    </addressBlock>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
        <addressBlock>
          <range>0x1000</range>
          <width>32</width>
        </addressBlock>
        <addressBlock>
          <range>0x1000</range>
          <width>32</width>
        </addressBlock>
    </bank>
```

In the following XML, memories ramA, ramB, ramC and ramD are all mapped to address 0x10000, but at bit offsets 0, 8, 16 and 24 respectively.

```
    <bank bankAlignment="parallel">
        <baseAddress>0x10000</baseAddress>
        <addressBlock>
            <range>0x1000</range>
            <width>8</width>
        </addressBlock>
        <addressBlock>
            <range>0x1000</range>
            <width>8</width>
        </addressBlock>
        <addressBlock>
            <range>0x1000</range>
            <width>8</width>
        </addressBlock>
        <addressBlock>
            <range>0x1000</range>
            <width>8</width>
        </addressBlock>
    </bank>
```

In some cases the 'parallel' example could be described by a single memory block, but there are a number of cases where tools need to know more details about how a memory bank is setup. For instance, if the memory bank is ROM, and the tools need to understand the setup to understand how to setup the ROM image.

4.6.7 Register description

Registers may be defined within the `memoryMap` element of a slave. Registers have the following attributes: `name`, `addressOffset` and `size`. The `name` element allows the register to be identified with a string. The `addressOffset` element indicates the offset that this register has from the containing address block's base address. The first register in the register bank may be at offset 0x10 whilst the base address is 0xC000 so the register is located at absolute address 0xC010. The final mandatory element, `size`, indicates how many bits wide the register is. This is independent of any of the attributes of the slave bus interface that can access the register, but is usually linked in some way by the hardware.

There are a number of optional elements that add to the description of the register:

- **`dim`**: This is used to assign a dimension to the register, so that it is repeated as many times as the value of the dim elements. For multi-dimensional register arrays the memory layout is assumed to follow the C language rules.
- **`volatile`**: Indicates that the data in the register is volatile, defaults to false.
- **`access`**: This element may take the values 'read-write', 'read-only' or 'write-only' to indicate the accessibility of the register

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

- **`dependency`**: If a register is dependent on another register then this element allows the dependency to be specified. There may be multiple dependencies specified. Dependencies are described below.
- **`reset`**: Indicates the value of the register's contents when the device is reset. This uses a value and an optional mask. When present, the mask value is to be 'and'ed with the current value before comparing it to the reset value. The default value for the mask is all ones (11111….).
- **`field`**: If a register has bitfields then they may be described. Fields are further described below.
- **`description`**: Allows a descriptive text to be associated with the register.
- **`parameter`**: If the register width can be parameterized in some way then the parameter names and types can be described in this element.

4.6.7.1 Bitfields

Bitfields within registers can be well described in IP-XACT. These are the mandatory elements:

- **`name`**: assigns a name to the bitfield
- **`bitOffset`**: describes the offset (from bit 0 of the register) where this bitfield starts
- **`bitWidth`**: this is the width of the field, counting in bits
- **`access`**: like the access element of register, this indicates whether or not the bitfield is read-only, etc.

These optional elements:

- **`description`**: allows a textual description of the bitfield
- **`values`**: lists the set of legal values that may be written to the bitfield, this includes the value, a description for the value and a name for the value that may be used as a token when programming the register
- **`parameter`**: allows for parameterization of the bitfield width

There is an example of a bitfield in Section 4.6.7.3.

4.6.7.2 Register dependencies

Registers that have dependencies on other registers in the component can be described in IP-XACT through the dependency element. An example is:

```
<spirit:dependency>
   <spirit:registerRef>ctrlReg</spirit:registerRef>
   <spirit:fieldRef>ctrlField</spirit:fieldRef>
   <spirit:value>0x0a</spirit:value>
   <spirit:mask>0x0f</spirit:mask>
</spirit:dependency>
```

Here, the register is controlled by a field *ctrlField* within another register called *ctrlReg*. If *ctrlField* has the value `0xa` after being 'and'ed with `0xf` then the controlled register is valid.

If there are multiple dependencies then *all* of the dependencies must be satisfied to enable the controlled register, i.e. the dependencies must be 'and'ed together.

4.6.7.3 Example register definition

The following register definition demonstrates many of the elements described above.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
<spirit:register>
  <spirit:name>VectCnt</spirit:name>
  <spirit:addressOffset>0x200</spirit:addressOffset>
  <spirit:size>32</spirit:size>
  <spirit:access>read-write</spirit:access>
  <spirit:resetValue>
    <spirit:value>0x0</spirit:value>
  </spirit:resetValue>
  <spirit:field>
    <spirit:name>E</spirit:name>
    <spirit:bitOffset>5</spirit:bitOffset>
    <spirit:bitWidth>1</spirit:bitWidth>
    <spirit:access>read-only</spirit:access>
    <spirit:description>Set if any other bits are
set</spirit:description>
  </spirit:field>
  <spirit:field>
    <spirit:name>IntSource</spirit:name>
    <spirit:bitOffset>0</spirit:bitOffset>
    <spirit:bitWidth>5</spirit:bitWidth>
    <spirit:access>read-write</spirit:access>
    <spirit:description>Set if interrupt occurred</spirit:description>
  </spirit:field>
  <spirit:description>This is the interrupt vector control
register</spirit:description>
</spirit:register>
```

So although the register is 32 bits wide, only the first 6 bits have been defined with bit fields.

4.6.8 Component HW Models

The `model` element of a component describes the 'physical' view of that component. By physical we mean the real implementation and interface whatever the view is. For example, a RTL view would describe the source hardware module/entity with its pin interface; a SW view would define the source device driver C file with its .h interface; a documentation view would define the datasheet of this IP.

Example: of model section for a Timer component describing the view of the IP in terms of compatibility, language, file set reference and model name (for example entity(architecture) or module). In this example the view is a RTL view for simulation and the fileSetRef *fs-vhdlSource* refers to an element section defined later in the file.

```
<spirit:model>
  <spirit:signals>
  </spirit:signals>
  <spirit:modelParameters>
  </spirit:modelParameters>
  <spirit:views>
    <spirit:view>
      <spirit:envIdentifier>:modelsim.mentor.com:</spirit:envIdentifier>
      <spirit:envIdentifier>:ncsim.cadence.com:</spirit:envIdentifier>
      <spirit:language spirit:strict="true">vhdl</spirit:language>
      <spirit:fileSetRef>fs-vhdlSource</spirit:fileSetRef>
      <spirit:modelName>leon2_Timers(struct)</spirit:modelName>
    <spirit:view>
  </spirit:views>
</spirit:model>
```

4.6.8.1 HW model view

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

The view specifies the representation level of the component (e.g. VHDL source for simulation). An example of description is given below.

```
<spirit:view>
  <spirit:envIdentifier>:modelsim.mentor.com:</spirit:envIdentifier>
  <spirit:envIdentifier>:ncsim.cadence.com:</spirit:envIdentifier>
  <spirit:language spirit:strict="true">vhdl</spirit:language>
  <spirit:fileSetRef>fs-vhdlSource</spirit:fileSetRef>
  <spirit:modelName>leon2_Timers(struct)</spirit:modelName>
</spirit:view>
```

The `envIdentifier` element is a string which designates and qualifies information about how this model view might be deployed in a particular tool environment.

The format of the element is a string with three colon [:] separated field in the format of "`Language:Tool:VendorSpecific`". The format is enforced by the schema; the regular expression which is used to check the string is `[A-Za-z0-9_+*\.]*:[A-Za-z0-9_+*\.]*:[A-Za-z0-9_+*\.]*`. This format divides the element's value into three sections each separated by a colon. The sections are:

Language:

The "Language" field indicates that this view may be compatible with a particular tool but only if that language is supported in that tool. For instance, different version of some simulators may support one or two or more languages. In some cases, knowing the tool compatability is not enough and may be further qualified by language compatability. For example, a compiled HDL model may work in a VHDL-enabled version of a simulator, but not in a SystemC-enabled version of the same simulator.

Tool:

The "Tool" field indicates that this view contains information that is suitable for the named tool. This might be used if this view references data that is tool-specific and would not work generically. Examples of this might be HDL models that use simulator-specific extensions, or models shipped in a binary/pre-compiled format.

Vendors will publish lists of approved tool identification strings. These strings should contain the tool name as well as the company's domain name, separated by dots. Some examples of well formed tool entries are:

- "`designcompiler.synopsys.com`"
- "`ncsim.cadence.com`"
- "`modelsim.mentor.com`"

This field can alternatively indicate generic tool family compatability such as '`*Simulation`' or '`*Synthesis`'. An initial list of identification strings for this element is available from the public area of the `spiriticonsortium.org` web site, and will be published by the individual tool vendors. To support transpotablity of created datafiles, when referencing a tool, it is important to use the published, therefore generally recognised, tool designation.

Vendor Extension: (e.g., further qualifications)

The "Vendor Extension" field can be used to further qualify tool and language compatability. This might be used to indicate additional processing information may be required to use this model in a particular environment. For instance, if the model is a SWIFT simulation model, the appropriate simulator interface may need to be enabled and activated.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Any or all of the <envIdentifier> fields may be used. Where there are multiple environments for which a particular <view> is applicable, multiple <envIdentifier> elements can be listed.

If the view contains an implementation of any of the whitebox elements for the component, the view section should include a reference to that whitebox element, with a string providing a language-dependent path to enable the DE to access the whitebox element.

Example: A status flag implemented in a VHDL file for a DMA component

```
<spirit:view>
   …
  <spirit:language>vhdl</spirit:language>
  <spirit:whiteboxElementRef spirit:whiteboxElementRef="error_flag">
    <spirit:path>dma_top/dma_registers/status_register(0)</spirit:path>
  </spirit:whiteboxElementRef>
</spirit:view>
```

4.6.8.2 HW model signals

The component signals (RTL ports) are listed in this section. The name, direction, size of the signals described in here should match the RTL entity/module definition.

Example: of signals section inside the model of a Timer component. Here only one signal is displayed (a 32 bits address input signal).

```
<spirit:signals>
  <spirit:signal>
    <spirit:name>paddr</spirit:name>
    <spirit:direction>in</spirit:direction>
    <spirit:left>31</spirit:left>
    <spirit:right>0</spirit:right>
    <spirit:export spirit:configGroups="export" spirit:id="sig_paddr"
spirit:prompt="paddr">false</spirit:export>
  </spirit:signal>
</spirit:signals>
```

The export element is to be used for signals that are exported to a higher level of hierarchy.

The left and right elements values are those specified in the RTL. They specify the left and right vector bounds. Signal width is max(left,right)-min(left,right)+1. When the bounds are not present, a scalar signal is assumed.

For example:

```
data: out std_logic_vector(7 downto 0);
```

Would be defined in SPIRIT as: left=7 right=0 with all bits in decreasing order.

```
output [29:3] data;
```

Would be defined in IP-XACT as: left=29 right=3 with all bits in decreasing order.

```
input [0:7] addr;
```

Would be defined in IP-XACT as: left=0 right=7 with all bits in increasing order.

The scalar signals; that can be written for example as:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
reset: in std_logic_vector(0 downto 0);
```

would be defined in IP-XACT with no `left` and `right`. Scalar signals must have no `left` or `right` attribute in IP-XACT. Even if both notations can coexist in HDL, one in the scalar signal name space the other one in the vectored signal name space.

4.6.8.3 HW model parameters

Component HW parameters allow the user to configure the component.

Example: of parameter section inside the model of a UART component. Here only one modelparameter is displayed (i.e. the Baud rate) but more can be added to configure the component.

```
<spirit:modelParameters>
  <spirit:modelParameter spirit:choiceRef="EXTBAUDChoice"
        spirit:choiceStyle="combo"
        spirit:configGroups="requiredConfig"
        spirit:dataType="boolean"
        spirit:format="choice"
        spirit:id="EXTBAUD"
        spirit:name="EXTBAUD"
        spirit:prompt="Set baud rate externally:"
        spirit:resolve="user">
      false
  </spirit:modelParameter>
</spirit:modelParameters>
```

The following modelparameter attributes can be defined:

- `dataType` (optional string representing the data type)
- `minimum` (optional string value indicating minimum legal value)
- `maximum` (optional string value indicating maximum legal value)
- `rangeType` (optional numeric value appearing automatically everywhere that minimum and maximum appear)

The `minimum` and `maximum` attributes are of type 'string' (and not float) and their value should be interpreted based on the value of `dataType`. This allows, for example, defining a maximum value of "0xffffffff" (which is an illegal float value) or a value of “64”.

The `rangeType` (defined in the SPIRIT common.att attribute group) can take the values: float, int, unsigned int, long, or unsigned long. If `rangeType` is not set, it is assumed to be “float”. Note that 'int' and 'unsigned int' are interpreted as 4 bytes and that 'long' and 'unsigned long' are 8 bytes.

4.6.9 Component Implementation Constraints

Implementation constraints can be defined to document requirements that must be met by an implementation of the component. Constraints are defined in groups called constraint sets (in SPIRIT elements `componentConstraints` and `signalConstraints`) to allow different constraints to be associated with different views of the component. A particular set of constraints is tied to a component view by the `constraintSetId` attribute in the constraint set and the matching `constraintSetRef` element in the view. Signal constraints specified within the component override corresponding constraints defined within bus definitions (if any). See the Appendix for a detailed definition and example utilizing implementation constraints.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.6.10 Component Files

Example: of `FileSets` section for a Timer component. Here only a few, among the complete list, source files are displayed. The `fileSetID` is the one referred to in the model in the previous section.

```
<spirit:fileSets>
  <spirit:fileSet spirit:fileSetId="fs-vhdlSource">
    <spirit:file>
      <spirit:name>../../common/config.vhd</spirit:name>
      <spirit:fileType>vhdlSource</spirit:fileType>
    </spirit:file>
    <spirit:file>
      <spirit:name>hdlsrc/timers.vhd</spirit:name>
      <spirit:fileType>vhdlSource</spirit:fileType>
      <spirit:logicalName>leon2_timers</spirit:logicalName>
    </spirit:file>
    </spirit:fileSet>
  </spirit:fileSets>
```

Note that SPIRIT allows file association to be specified, with defaults and alternatives.

The default order for file association and dependency shall be the same as the order in which entries appear in the XML file, the first file or dependency recorded in the XML shall be taken first.

4.7 Hierarchy represented by a design file

It is possible to describe hierarchical designs in IP-XACT. In any IP-XACT design the design file references components files. In a hierarchical design some or all of these component files in turn have views which reference further design files, or design configuration files, describing the design of those components. This linking allows for unlimited levels of hierarchy in a design. All referencing of designs, configurations of designs and components in IP-XACT are done through the VLNV. Four attributes (`vendor, library, name, and version`) uniquely identify a design, a configuration of a design or a component. Below is an example of a hierarchical design.

Example of the highest Design file in a hierarchical design:

```
<spirit:design>
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Example</spirit:library>
  <spirit:name>Top</spirit:name>
  <spirit:version>1.00</spirit:version>
…
  <spirit:componentInstance>
     <spirit:instanceName>APB</spirit:instanceName>
    <spirit:componentRef
      spirit:vendor="spiritconsortium.org"
      spirit:library="Example"
      spirit:name="APB_top"
      spirit:version="1.00" />
  </spirit:componentInstance>
```

Example of a Component file in a hierarchical design:

```
<spirit:component>
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Example</spirit:library>
  <spirit:name>APB_top</spirit:name>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
  <spirit:version>1.00</spirit:version>
…
  <spirit:model>
    <spirit:views>
      <spirit:view>
        <spirit:name>Hierarchical</spirit:name>
          <spirit:envIdentifier>::</spirit:envIdentifier>
             <spirit:hierarchyRef
               spirit:vendor="spiritconsortium.org"
               spirit:library="Example"
               spirit:name="APBSubSystem"
               spirit:version="1.2"/>
      </spirit:view>
```

Example of the lower Design file in a hierarchical design, showing the hierarchical connection of a bus interface:

```
<spirit:design>
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Example</spirit:library>
  <spirit:name>APBSubSystem</spirit:name>
  <spirit:version>1.2</spirit:version>
…
    <spirit:hierConnections>
      <spirit:hierConnection spirit:interfaceName="UartIF1">
      <spirit:componentRef>slave_3</spirit:componentRef>
      <spirit:interfaceRef>uart_if</spirit:interfaceRef>
    </spirit:hierConnections>
```

4.8 IP-XACT bus definition

In IP-XACT a Bus IP is defined by the busDefinition.xml, which is referred by the components bus interfaces. The Bus definition only contains the definition of the interfaces (i.e. the signals for wire to wire connection, or a collection of wires connected together and constraints to apply to these signals). The behavior of the bus (address decoding, arbitration, configuration…) is handled by the component's bus generator (which is not part of the bus definition).

Sample of an AHB busdefinition:

```
<?xml version="1.0" encoding="UTF-8" ?>
<spirit:busDefinition
xmlns:spirit= http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2/index.xsd">

  <spirit:vendor>amba.com</spirit:vendor>
  <spirit:library>AMBA</spirit:library>
  <spirit:name>AHB</spirit:name>
  <spirit:version>v1.0</spirit:version>
  <spirit:extends spirit:vendor="amba.com"
      spirit:library="AMBA"
      spirit:name="AHBlite" />
  <spirit:maxMasters>16</spirit:maxMasters>
  <spirit:maxSlaves>16</spirit:maxSlaves>
  <spirit:signals>
    <spirit:signal>
      <!—
A signal from bus master to the bus arbiter which indicates that the bus
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
master requires the bus. There is an HBUSREQ signal for each bus master
in the system, up to a maximum of 16 bus masters.
      -->
      <spirit:logicalName>HBUSREQx</spirit:logicalName>
      <spirit:onMaster>
        <spirit:direction spirit:fixedDirectionType="out" />
      </spirit:onMaster>
…
    </spirit:signal>
  </spirit:signals>
</spirit:busDefinition>
```

The bus definition file contains a list of logical signals that may appear on a bus interface of that bus type. Note that this doesn't mean that all these signals must exist or be present on every interface.

The `onMaster`, `onSlave`, `onSystem` tags associated with each logical signal are optional constraints. So if none of these constraints are specified, then any of the logical signals can appear in any format in any of the bus interfaces. The busDefinition author has the choice of how far to constrain the definitions. Generally speaking, more constraints in the definitions reduce implementation flexibility for whoever is creating bus IP that conforms to the busDefinition.

The `group` tag defined under the `onSystem` element is there to distinguish between different sets of system interfaces. Usually, all the arbiter signals would be processed together, or all the clock/reset signals would be processed together. So this is really a mechanism to specify any sort of non-standard bus interface capabilities for a bus definition.

Default implementation constraints can be defined for signals defined in the bus definition using the `busDefSignalConstraints` element. These constraints are treated as default constraints for the corresponding signal of the component. See the appendix for more details and an example utilizing implementation constraints.

The `maxMasters` and `maxSlaves` elements specify the maximum number of masters or slaves that may appear on a bus. If the `maxMasters` element is not present then the numbers of masters is unbounded. If the `maxSlaves` element is not present then the numbers of slaves is unbounded.

4.9 IP-XACT Bus and interconnect model

Though busses are components in IP-XACT, bus models need a specific section because of their importance in SoC platforms and their potential complexity.

A bus (or more generally, a design interconnect) can be modelled with two concepts:

- Its interfaces
- Its memory map

Two main categories of busses can be distinguished according to their interface:

- Symmetric busses (such as OCP, VCI, STbus, crossbars, network on chip) which can be modeled with direct interfaces
- Asymmetric busses (such as AHB, APB, CoreConnect) which can be modeled with mirrored interfaces

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

The following sections describe, first, the bus interfaces and how to connect them. This is followed by a description of the how to model the internal representation of a bus, and of the relationship between bus interfaces and memory maps.

4.9.1 Bus interface

Each IP component normally has one or more bus interfaces identified in the component XML file. Bus interfaces are groups of signals that belong to an identified bus type (i.e. a reference to a busDefinition).

Before introducing the bus model, it is useful to introduce the IP-XACT terminology (where commonly used words have very specific meanings).

Components are connected together by linking the **bus interfaces** together. There are three classes of bus interface: **master**, **slave** and **system** each with two flavors: **direct** and **mirrored**. Additionally a monitor interface is supported for connecting monitor IP into the design for verification

4.9.1.1 Direct interfaces

A **master interface** is the bus interface that initiates a transaction (like a read or write) on a bus. For example, processors and DMA controllers implement master bus interfaces. Master interfaces tend to have associated address spaces (address spaces with programmers view)

A **slave interface** is the bus interface that terminates/consumes a transaction initiated by a master interface. Typically, memories, UARTs and other peripherals implement slave bus interfaces. Slave interfaces often contain information about the registers that are accessible through the slave interface.

A slave interface may contain a `fileSetRefGroup` element. This element may seem out of place but is needed to allow each slave port to reference a unique `fileSet` element. This element could then be used to reference a software driver that can be made different for each slave port.

A **system interface** is an interface that is neither a master nor slave interface, and allows specialized (or non-standard) connections to a bus. System interfaces might be used to interface external arbiters to a bus. System interfaces help to handle situations not covered by the bus specification, or deviations from the standard.

Modeling guidelines for system interface:

In general, if a signal's functionality is documented in the bus's documentation, then it should be included in master and slave interfaces; only those signals that do not have documented functionality should be included in system interfaces. Some examples follow:

- For an AMBA bus, HCLK and HRESETN are specified bus signals. Therefore we propose to include them with other signals in master and slave bus interfaces (i.e. inputs on both master and slave interfaces, and not split out into separate system interfaces). If a clock generator is connected to that AMBA bus, then it would have one or more clock outputs. But nothing in the AMBA spec specifies how HCLK should be driven (nothing says that the clock generator has to be included) so in that case, we recommend that HCLK outputs appear in a system interface.
- For an AMBA bus with an external arbiter. A signal like HGRANT is an input to a master and doesn't exist on a slave interface. Some implementations of an AMBA bus might have an internal arbiter, so there is no component that has an output HGRANT signal. However some implementations might support an external arbiter, so HGRANT might also be specified as an output

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

in a system interface of group "external arbiter". This doesn't mean that there must always be an external arbiter; but that if one exists, then the signals are constrained in this way.

- For Altera Excalibur, the HGRANT on the Excalibur device is an output, not an input. This is because Excalibur has the arbiter built into the IP module, but external masters that can be connected. So the Excalibur IP would have a bus interface (in addition to its AMBA master and slave interface) that is a system arbiter interface.

Some buses have specialised sideband signals. If these are tied or related to the standard signals in the bus (as opposed to being completely standalone), we would also expect these signals to have some sort of `system` element designator in the busdef.

4.9.1.2 Mirrored interfaces

As the name suggests, a mirrored interface has the same (or similar) signals to its related direct bus interface, but the signal directions are reversed. So a signal that is an input on a direct bus interface would be an output in the matching mirror interface.

A mirror bus interface (like its non-mirror counterpart) support master, slave and system classes, and are always associated with a particular bus definition.

4.9.1.3 Monitor interfaces

A **monitor interface** is an interface used in verification that is neither a master, slave nor system interface. This allows specialized (or non-standard) connections to a bus that will not count as a connected interface. Monitor interfaces are used to connect verification IP used to monitor an interface of type master, slave or system and do not count as a connected interface in the design environment. A monitor interface is identified by the `monitor` element in the interface definition, with an attribute to specify the type of active interface being monitored (master, slave, system).

A monitor interface is declared in the component XML as show in the example below, for monitoring a master bus interface:

```
<spirit:component>
 …
  <spirit:name>MyMonitor<spirit:name>
  …
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>C</spirit:name>
      <spirit:monitor spirit:interfaceType="master" />
    </spirit:busInterface>
  </spirit:busInterfaces>
</spirit:component>
```

4.9.2 Interfaces connection

4.9.2.1 Connection rules

As defined in the previous section, a bus interface can have direct or mirrored interfaces. Each of 3 kinds: master, slave and system. Direct interfaces can be: Master (M), Slave (S) or System (Y). Mirrored interfaces can be: Mirrored Master (MM), Mirrored Slave (MS), Mirrored System (MY) or Monitor.

The following rules apply to connect two component interfaces:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Direct Master to Slave interface connection is allowed only under the following conditions:

- The bus definition permits direct connection.
- For addressable busses, the address range defined on the slave interface is less than or equal to the address range defined on the master interface
- For addressable busses, the values of bitsInLau at the master and the slave match.

USAGE NOTE: IP-XACT does not provide mechanisms for describing additional constraints that must be met when connecting like but parameterizable interfaces. For example, such an issue may occur when connecting a 32-bit bus to a 16-bit interface on an IP. In this case, the Least Addressable Unit must be identified for connection to be completed automatically

In the case of connecting like but parameterizable interfaces, two alternatives may be supported: (i) the IP provider may supply a set of generators required for completing integration between the parameterized interfaces, or (ii) the specifier of the interface protocol must list the set of possible parameters, and a design tool may be able to guide the user through a set of steps to complete the connections.

In all other cases, a direct connection is not allowed. The master interface has to connect to a Mirror Master interface (respectively, the Slave interface has to connect to a Mirror slave interface). In other words, a component with mirror interfaces must be inserted between components with direct interfaces each time there is not a point to point connection in the design, or if the component interfaces do not match.

Direct to Mirror interface connection is allowed for:

- Direct Master to Mirrored Master
- Direct Slave to Mirrored Slave

A Mirror-to-Mirror interface connection is not permitted. To connect two mirror interfaces one has to insert a component with direct interfaces in between. But a channel cannot connect directly to another channel (because two Mirrored Interfaces cannot directly be connected). To connect two channels together, a regular component must be inserted in between, just as would be required when connecting two different types of bus together.

4.9.2.2 Direct Master to Direct Slave interface connection

The interface connection (between a component master interface and a component slave interface) is always a point-to-point (i.e. logical) connection.

A design.xml (or a hierarchical component.xml) will have an interconnection section containing an `interconnection` element which has two `activeInterface` elements each containing two attributes: `componentRef` and `busRef`.

The connection of these elements and attributes is illustrated in the following picture for a master to slave interface.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

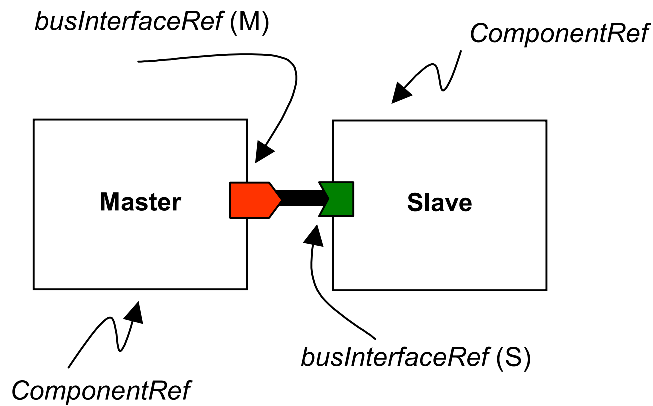


Figure 6 - Master to Slave interface connection

A sample of the interconnection section of the design.xml file is given below.

```
<spirit:interconnections>
  <spirit:interconnection>
    <spirit:activeInterface
        spirit:componentRef="Master"
        spirit:busRef="M" />
    <spirit:activeInterface
        spirit:componentRef="Slave"
        spirit:busRef="S" />
  </spirit:interconnection>
</spirit:interconnections>
```

4.9.2.3 Direct Master Interface to Mirrored Master interface connection

This connection is made in the same way as the direct master to slave connection. Consider the following connectivity:

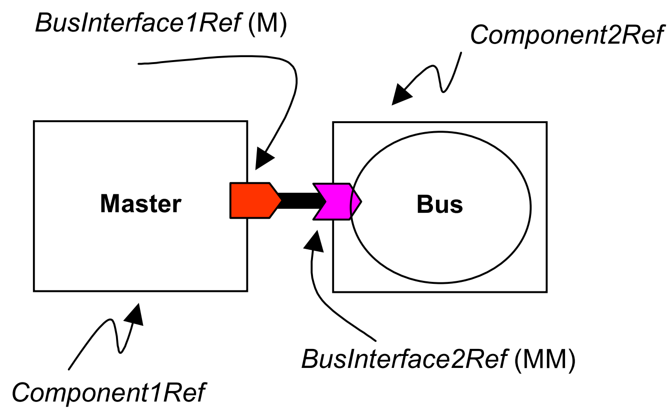


Figure 7 - Master to Mirror interface connection

The interconnection section of the design.xml file is given below.

```
<spirit:interconnections>
  <spirit:interconnection>
    <spirit:activeInterface
       spirit:componentRef="Master"
       spirit:busRef="M" />
    <spirit:activeInterface
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
        spirit:componentRef="Bus"
        spirit:busRef="MM" />
  </spirit:interconnection>
</spirit:interconnections>
```

4.9.2.4 Mirrored slave Interface to direct slave interface connection

This is the counterpart of the previous section. It describes the connection between a channel mirror interface and a component bus interface. This connection is defined in the platform design. Again, the connection between a mirror interface and a bus is always a point-to-point (i.e. logical) connection.

The connection is illustrated in the following figure for a channel mirror to slave interface.

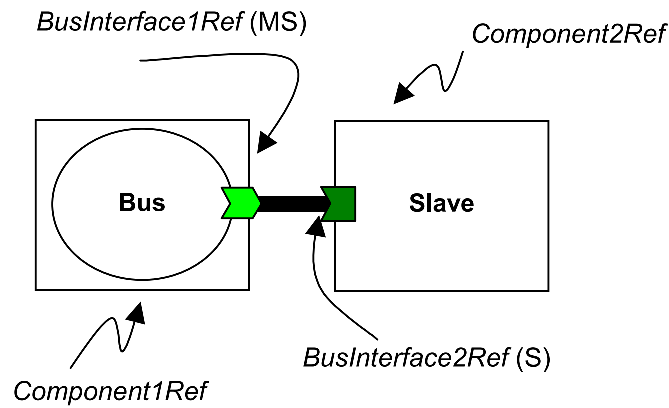


Figure 8 - Mirror to Slave interface connection

A sample of the interconnection section (in design.xml) for the figure above is given below:

```
<spirit:interconnections>
  <spirit:interconnection>
    <spirit:activeInterface
        spirit:componentRef="Bus"
        spirit:busRef="MS" />
    <spirit:activeInterface
        spirit:componentRef="Slave"
        spirit:busRef="S" />
  </spirit:interconnection>
</spirit:interconnections>
```

4.9.2.5 Monitor Interface connection

With a monitor interface it is possible to have multiple connections between an interface of a design IP and an interface of verification IP.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

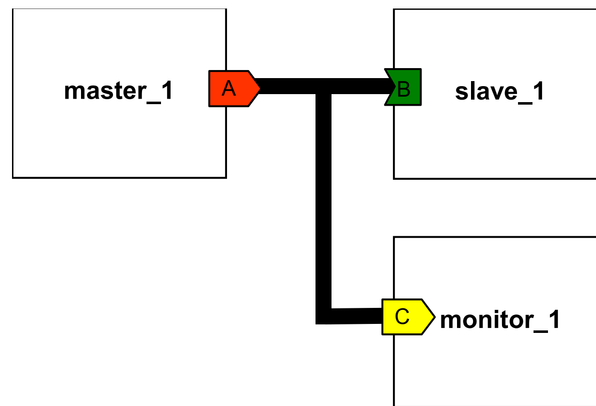


Figure 9 - Master, Slave and Monitor interface connection

This figure represents the connection between a master interface (A) on Master_1 and a slave interface (B) on Slave_1. The monitor (Monitor_1) is used to check e.g. the protocol. Monitor_1 has a monitor interface (C).

Two interconnections are used to enable this:

- Master_1(A) → Slave_1(B) – a standard interconnection
- Master_1(A) → Monitor_1(C) – a monitorInterconnection

A sample of the interconnection section (in design.xml) for the figure above is given below:

```
<spirit:interconnections>
   <spirit:interconnection>
     <spirit:activeInterface
      spirit:componentRef="Master_1"
      spirit:busRef="A" />
    <spirit:activeInterface>
      spirit:componentRef="Slave_1"
      spirit:busRef="B" />
   </spirit:interconnection>
   <spirit:monitorInterconnection>
     <spirit:activeInterface
      spirit:component1Ref="Master_1"
      spirit:busInterface1Ref="A" />
     <spirit:monitorInterface
     spirit:componentRef="Monitor_1"
     spirit:busRef="C" />
   </spirit:interconnection>
</spirit:interconnections>
```

4.9.3 Bus internal representation

There are two constructs used to connect standard components (traditional components usually with 'masters' and 'slave' interfaces) together. These constructs are also encapsulated into components.

A channel is used to connect together component master, slave and system interfaces on the same bus. All masters connected to a channel see all slaves at the same physical address, and only one transaction can be active in a channel at any point in time. This does not preclude bus protocols which utilize pipelining.

 © Copyright The SPIRIT Consortium 2006.
All rights reserved.

A bus bridge is a component that is used as an interface between one bus and another (often a peripheral bus to the main system bus). Such a component always has at least one master interface (onto the peripheral bus) and one slave interface (onto the main system bus). Crossbar bus infrastructure (for instance ARM Multilayer AMBA) is also treated as a bus bridge – such examples might have multiple master and multiple slave interfaces. A bus bridge can support multiple simultaneous transactions, and the slaves existing in the master interface address spaces may appear at different address to masters connected (by a channel) to each of the bus bridge's slave ports. The mechanism for specifying how each of the master interface address spaces appears to the slave interface is specified by the `bridge` element.

4.9.3.1 Channel

All the component internal connections between mirrored masters and slave interfaces can be encapsulated within a structure called a **channel**. A channel can represent a simple wiring interconnect or a more complex structure such as a bus.

The channel is a general name that denotes the collection of connections between multiple internal bus interfaces. The memory map between these connections is restricted so that, for example, a generator can be called to automatically compute all the address maps for the complete design.

As illustrated in the following figure, the channel encapsulates the connection between master and slave components. A channel is the construct, which represents the bus infrastructure and allows transactions initiated by a master interface to be completed by a slave interface.

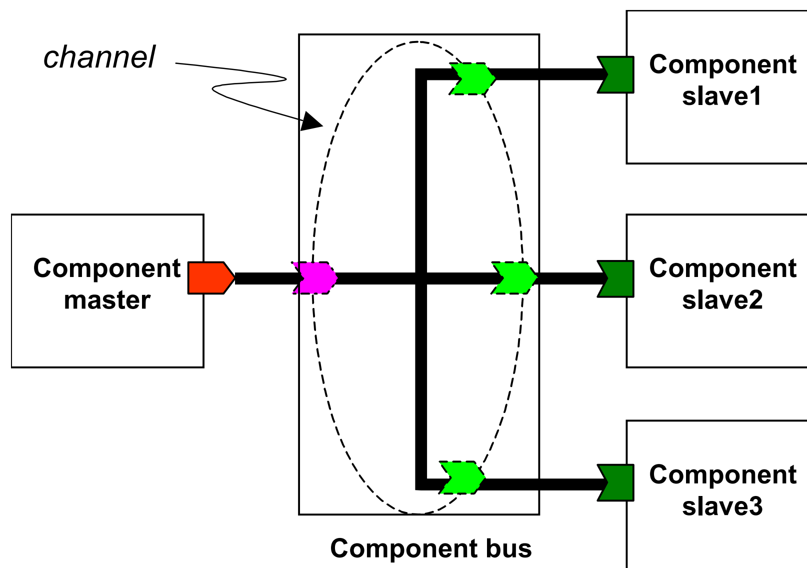


Figure 10 - Master – Slaves connection encapsulated in a channel

There are some very specific rules associated with channels and it is important to use channel concepts in the correct place (and, conversely, not use channel concepts in inappropriate places). Here is the list of rules that apply for the channel:

- A channel can only have one address space (a.k.a. transmission / transformation matrix). In other words, a slave connected to a channel will always have the same address as seen from all masters connected to this channel. This guarantees the slave addresses (as seen by each master) is consistent for the system. For example, if we have three slaves connected to

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

a channel with the following address map: Slave1: 0x000 to 0x1000 (4Kb); Slave2: 0x000 to 0x1000 (4Kb); Slave3: 0x000 to 0x1000 (4 Kb), the channel will create the address mapping so that the master can see an address space of 12Kb (from 0x000 to 0x0FFF it will address Slave1, from 0x1000 to 0x1FFF it will address Slave2 and from 0x2000 to 0x2FFF it will address Slave3). As a consequence, all slave interfaces connected to a channel will see the same address (if they don't, then they are connected to different channels); and if more than one master/slave interface pair is active/selected simultaneously, then there is more than one channel present.

- A channel can only relate mirrored interfaces because some busses can have asymmetric interfaces (e.g. AHB). Therefore, to cover all type of busses, the channel interfaces are always Mirrored interfaces. As a consequence, a channel can only connect to a direct interface (it can not connect directly to another channel). Note however that not all mirror interfaces of a channel have to be connected.
- A channel cannot be hierarchical
- A channel supports Memory mapping and re-mapping.

Note that simple wire connections (e.g. a clock signal connecting to all components of the system) may be modelled as an IP-XACT `channel` or as IP-XACT `signal` object.

A sample of the XML code describing the channel and its mirror interfaces for a simple AHB-like bus component is given below.

```
<spirit:component
   …
  <spirit:busInterfaces>
    <spirit:busInterface spirit:id="AHB_MS">
      <spirit:name>AHB_mirror_slave</spirit:name>
      <spirit:busType spirit:library="AMBA" spirit:name="simpleAHB"
spirit:vendor="spiritconsortium.org" />
      <spirit:mirroredSlave/>
      <spirit:connection>required</spirit:connection>
…
    <spirit:busInterface spirit:id="AHB_MM">
      <spirit:name>AHB_mirror_master</spirit:name>
      <spirit:busType spirit:library="AMBA" spirit:name="simpleAHB"
spirit:vendor="spiritconsortium.org" />
      <spirit:mirroredMaster/>
…
    </spirit:busInterface>
  </spirit:busInterfaces>
  <spirit:channels>
     <spirit:channel>
      <spirit:busInterfaceRef>AHB_mirror_slave</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>AHB_mirror_master</spirit:busInterfaceRef>
    </spirit:channel>
  </spirit:channels>
…
</spirit:component>
```

4.9.3.2 Channel internal connection

The `channel` internal connection is described in IP-XACT by the list of Mirror interfaces defined inside the `channel` element of a `component`. For example, if there are 3 master `component`s connected to a bus (through the bus Mirror Master interfaces), and 3 slave `component`s connected to the same bus (through the bus

 © Copyright The SPIRIT Consortium 2006. All rights reserved.

Mirror slave interfaces). To create a path from all mirror master interfaces to all mirror slave interfaces (MMi to MSj with I,j =1..3). Then create a single `channel` including all the interfaces. This is illustrated by the following figure.

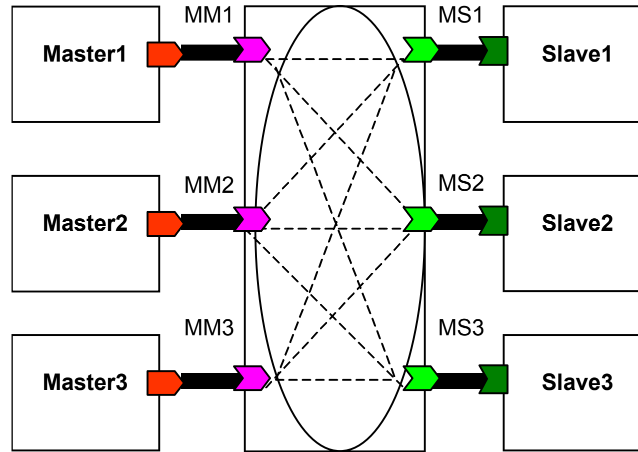


Figure 11 - Channel Internal connection

A sample of the XML code describing the component channel internal connections for the figure above is given here after. Note that the Mirror masters to Mirror slave internal connections are implicit. Only the interfaces are listed.

```
<spirit:component
   …
  <spirit:channels>
    <spirit:channel>
      <spirit:busInterfaceRef>MM1</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MM2</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MM3</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS1</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS2</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS3</spirit:busInterfaceRef>
    </spirit:channel>
  </spirit:channels>
…
</spirit:component>
```

Note that the order of the `busInterfaceRef` elements may hold some meaning to the design and this order should be maintained.

Note also there could be more than one channel in a bus component, and possibly different memory map for each channel, as long as the different channels do not intersect. An example is described in the next figure.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

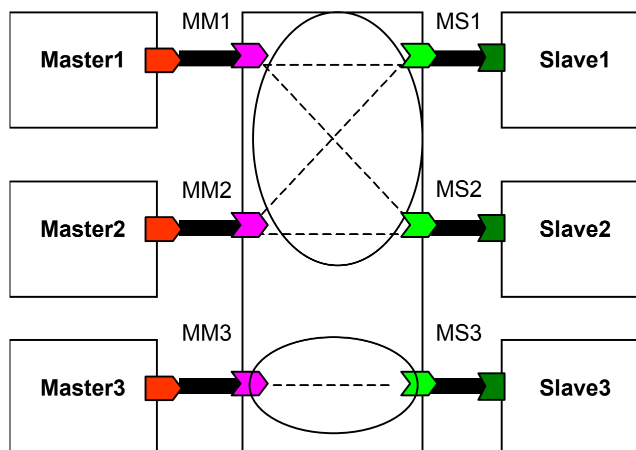


Figure 12 - Internal connection using two channels

The two channels would be written in IP-XACT as:

```
<spirit:component
   …
  <spirit:channels>
    <spirit:channel>
      <spirit:busInterfaceRef>MM1</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MM2</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS1</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS2</spirit:busInterfaceRef>
    </spirit:channel>
    <spirit:channel>
      <spirit:busInterfaceRef>MM3</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS3</spirit:busInterfaceRef>
    </spirit:channel>
  </spirit:channels>
…
</spirit:component>
```

4.9.3.3 Bridge

Some busses (such as OCP-based, VCI, STbus, crossbars, network on chip) can be modeled using component bridges. The bridge is a mechanism to model the internal relationship between master interfaces and slave interfaces inside a component. In a bridge multiple transactions can occur at a time. For example if two masters, addressing two distinct slaves, want to access the bus at the same time, both can be granted as long as a 'bridge path' has been defined in IP-XACT.

There are some very specific rules associated with bridges and it is important to use the bridge concept in the correct place (and, conversely, not use bridge concepts in inappropriate places). Here is the list of rules that apply for the bridge:

- A bridge can have multiple address spaces. Specifically a bridge will have one or more master interfaces, and each master interface may have a local address space associated with that interface.
- A bridge can only have direct interfaces. As a consequence, a bridge (like any other regular component) can directly connect to another component (Master interface to Slave interface connection) under the conditions defined in the previous section. Or it can connect to a channel (e.g. Master interface to Mirror Master Interface).
- A bridge can be hierarchical

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

- A bridge supports Memory mapping and re-mapping.

4.9.3.4 Bridge internal connection

A bridge is another mechanism besides the channel for describing the bus internal connections. As defined in the introduction of this section, bridges may better match partial crossbar busses or symmetric interconnects.

Bridges, unlike channels explicitly describe the internal point-to-point connections between the component interfaces. This is illustrated by the Figure below. Three master components are connected to a bus interconnect (modelled as a bridge), to which connect three slaves. The bus is a partial crossbar.

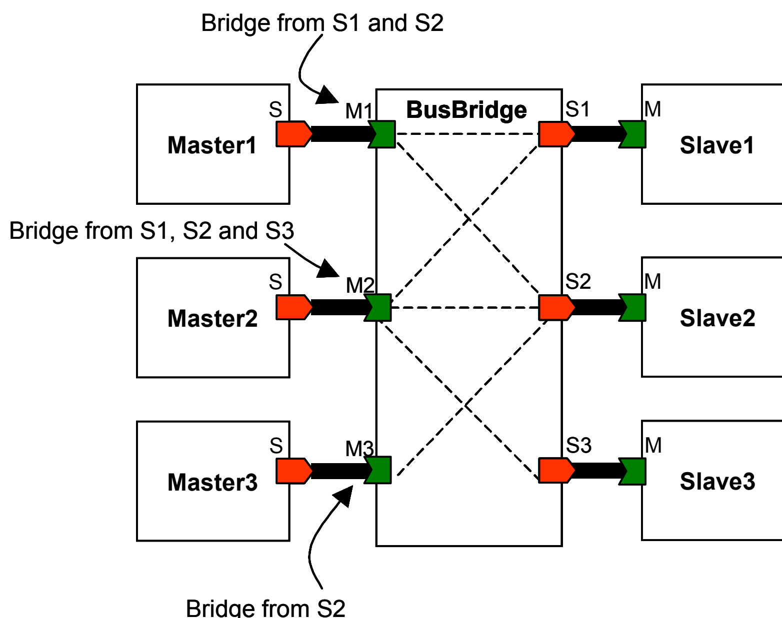


Figure 13 - Bus bridge component

The bus bridge of the figure above would be described is IP-XACT as follows:

```
<spirit:component>…
  <spirit:busInterfaces>
    <spirit:busInterface spirit:id="BB_S1">
      <spirit:name>S1</spirit:name>
      <spirit:busType spirit:vendor="spiritconsortium.org"
spirit:library="my_lib" spirit:name="simpleBB"/>
      <spirit:slave>
        <spirit:bridge spirit:masterRef="M1"/>
        <spirit:bridge spirit:masterRef="M2"/>
      </spirit:slave>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>S2</spirit:name>
      <spirit:busType spirit:vendor="spiritconsortium.org"
spirit:library="my_lib" spirit:name="simpleBB"/>
      <spirit:slave
        <spirit:bridge spirit:masterRef="M1"/>
        <spirit:bridge spirit:masterRef="M2"/>
        <spirit:bridge spirit:masterRef="M3"/>
      </spirit:slave>
    </spirit:busInterface>
    <spirit:busInterface>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
      <spirit:name>S3</spirit:name>
      <spirit:busType spirit:vendor="spiritconsortium.org"
spirit:library="my_lib" spirit:name="simpleBB"/>
      <spirit:slave>
        <spirit:bridge spirit:masterRef="M2"/>
      </spirit:slave>
    </spirit:busInterface>
  </spirit:busInterfaces>
</spirit:component>
```

The `bridge` element indicates how `addressSpace`s on Master Interfaces are mapped back on the Slave Interface `addressSpace`.

At the top level (design.xml), the interconnection between the components of the figure above would look like this.

A sample of the interconnection section (inside the design.xml file) is shown below.

```
<spirit:interconnections>
  <spirit:interconnection>
    <spirit:activeInterface
      spirit:componentRef="Master1"
      spirit:busRef="M" />
    <spirit:activeInterface
      spirit:componentRef="BusBridge "
      spirit:busRef="S1" />
  </spirit:interconnection>
>    <spirit:activeInterface
      spirit:componentRef="Master2"
      spirit:busRef="M" />
    <spirit:activeInterface
      spirit:componentRef="BusBridge "
      spirit:busRef="S2" />
  </spirit:interconnection>
>    <spirit:activeInterface
      spirit:componentRef="Master3"
      spirit:busRef="M" />
    <spirit:activeInterface
      spirit:component2Ref="BusBridge "
      spirit:busRef="S3" />
  </spirit:interconnection>
>    <spirit:activeInterface
      spirit:componentRef="BusBridge"
      spirit:busRef="M1" />
    <spirit:activeInterface
      spirit:componentRef="Slave1"
      spirit:busRef="S" />
  </spirit:interconnection>
>    <spirit:activeInterface
      spirit:componentRef="busBridge"
      spirit:busRef="M2 " />
    <spirit:activeInterface
      spirit:componentRef="Slave2"
      spirit:busRef="S" />
  </spirit:interconnection>
>    <spirit:activeInterface
      spirit:componentRef="BusBridge"
      spirit:busRef="M3" />
    <spirit:activeInterface
      spirit:componentRef="Slave3"
      spirit:busRef="S" />
  </spirit:interconnection>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
</spirit:interconnections>
```

Unlike channels, bus bridges are regular components with direct interfaces and therefore can be chained to other components direct interfaces or connected to channels mirrored interfaces.

4.9.3.5 Multi-Layer busses modelling

Multi-Layer (ML) busses have to be modelled as component bridges with direct Interfaces or as a hierarchical component. They cannot be modelled as channels because they support multiple memory maps.

A special case however has to be considered for asymmetric multi-layer busses (i.e. busses with asymmetric interfaces). They cannot be directly connected to direct slave interface because of '`onSlave`' or '`onMaster`' signals they own (e.g. in the AMBA multi layer AHB bus definition, the chipselect HSEL is defined only for the Slave Interface). Therefore a channel component has to be introduced (e.g. between the busInterface and the memory slave interface) where the chip-select would appear on the channel mirror slave interface. This is illustrated in the following figure.

Figure 14 - Asymmetric Multi-layer bus connection using channels

In case of a connection from an AHB ML bus-matrix to an AHB slave (i.e., HSEL present) a new compatible busDefinition can be provided to connect the bus matrix to the slave.

4.9.4 Memory Map

This section details how to describe a memory map in IP-XACT using the two bus representations: channels and bridges. For bridge, we have to distinguish two kinds of bridges with regard to the memory map: the transparent bridge and the opaque bridge.

It applies both to direct and mirrored connection indifferently; the only differences being that direct connections allow cascading directly bridges and channels whereas mirrored connection require them to be interleaved.

A bus implemented as a **transparent bridge** does not modify the address. It just does decoding (or demux). For example, if a master component connected to a transparent bridge writes to a memory at the address 0x1900, the transparent bridge will decode the address to select the appropriate memory (e.g. memory2 in the range 0x1000-0x1FFF), and hit that memory at address 0x1900.

A bus implemented as an **opaque bridge** can modify the address; i.e. remove the base address and just use the offset. For example, if a master component connected to an opaque bridge writes to a memory at the address 0x1900, the opaque bridge will decode the address to select the appropriate memory (e.g. memory2 in the range 0x1000-0x1FFF), and hit that memory at address 0x0900.

The different memory map modelling methods will be illustrated (i.e. channel, transparent bridge and opaque bridge) based on the same example. The example consists of a single master connected to 3 memories (of 4K each) through a bus. The first memory starts at address 0x0000, the second at address 0x1000 and the third at address 0x2000.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.9.4.1 Memory map in channel

In a channel, the memory map is defined on each Mirrored Slave using the `baseAddresses` element. This element contains the base address and the range of the addressable memory. The base address is defined using the element `remapAddress` but really describes the base address.

This memory map of channel for our example is illustrated in the following figure. The `addressSpace` (AS) element defined on the Master interface is the total address space as seen by this master.

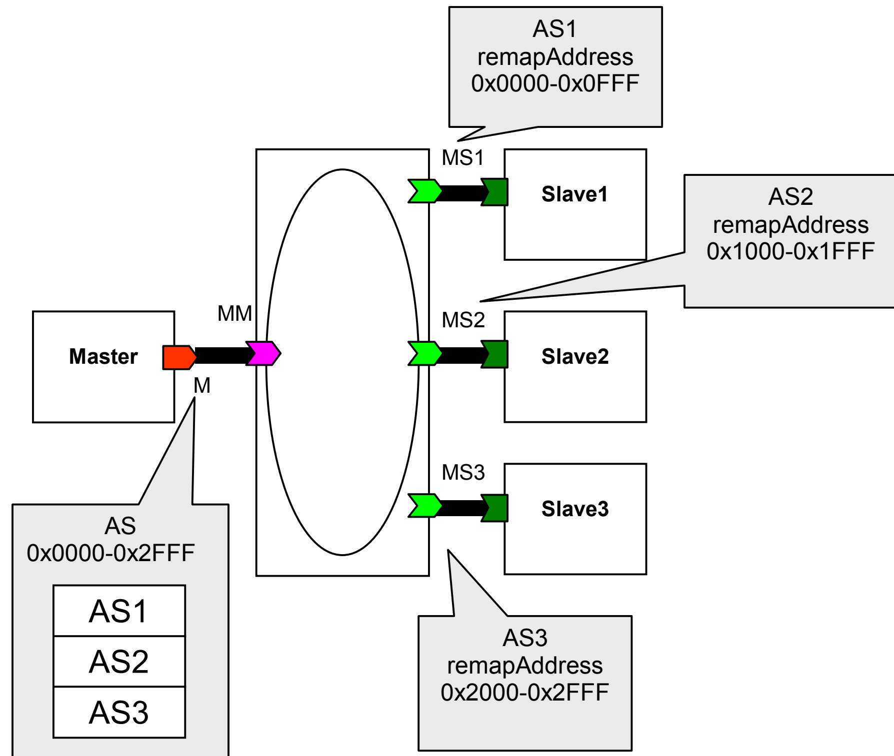


Figure 15 - Memory Map in Channel

A sample of the IP-XACT description for such a memory map is given below:

```
<spirit:component>…
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>MS1</spirit:name>…
      <spirit:mirroredSlave>
        <spirit:baseAddresses>
          <spirit:remapAddress>0x0000</spirit:remapAddress>
          <spirit:range>0x1000</spirit:range>
        </spirit:baseAddresses>
       </spirit:mirroredSlave>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>MS2</spirit:name>…
      <spirit:mirroredSlave>
        <spirit:baseAddresses>
          <spirit:remapAddress>0x1000</spirit:remapAddress>
          <spirit:range>0x1000</spirit:range>
        </spirit:baseAddresses>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
      </spirit:mirroredSlave>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>MS3</spirit:name>…
      <spirit:mirroredSlave>
        <spirit:baseAddresses>
          <spirit:remapAddress>0x2000</spirit:remapAddress>
           <spirit:range>0x1000</spirit:range>
        </spirit:baseAddresses>
      </spirit:mirroredSlave>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>MM</spirit:name>…
      <spirit:mirroredMaster/>
    </spirit:busInterface>
  </spirit:busInterfaces>
  <spirit:channels>
    <spirit:channel>
      <spirit:busInterfaceRef>MS1</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS2</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS3</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MM</spirit:busInterfaceRef>
    </spirit:channel>
  </spirit:channels>
</spirit:component>
```

4.9.4.2 Memory map in transparent bridge

A bridge is transparent by default (no specific attribute defined on the bridge element). In transparent bridges, there is no `memoryMap` section. The address space specified on each master interface is static, i.e. the base address of each memory is directly defined under the master interface.

On each master interface, the address space is referenced with the `addressSpaceRef` element, using the `addressSpaceRef` attribute. This attribute references to the `addressSpace` section directly defined under the component. The `addressSpace` element contains a name (referenced from the Masters) and a `range` element.

In addition, each master interface specifies a base address under the `addressSpaceRef` element.

This memory map in the transparent bridge for our example is illustrated in the Figure 16. The address space (AS) defined on the Master interface is the total address space as seen by this master.

 © Copyright The SPIRIT Consortium 2006. All rights reserved.

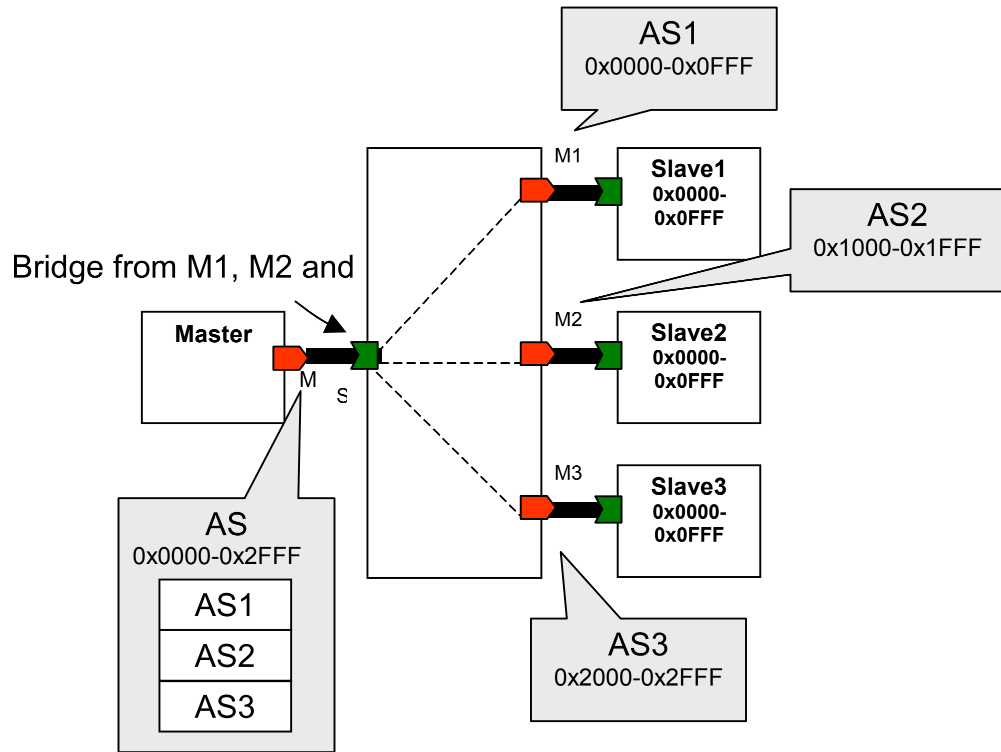


Figure 16 – Memory Map in a Transparent Bridge

```
<spirit:component>…
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>M1</spirit:name>…
      <spirit:master>
        <spirit:addressSpaceRef spirit:addressSpaceRef="AS1">
          <spirit:baseAddress>0x0000</spirit:baseAddress>
        </spirit:addressSpaceRef>
      </spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>M2</spirit:name>…
      <spirit:master>
        <spirit:addressSpaceRef spirit:addressSpaceRef="AS2">
          <spirit:baseAddress>0x1000</spirit:baseAddress>
        </spirit:addressSpaceRef>
      </spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>M3</spirit:name>…
      <spirit:master>
        <spirit:addressSpaceRef spirit:addressSpaceRef="AS3">
          <spirit:baseAddress>0x2000</spirit:baseAddress>
        </spirit:addressSpaceRef>
      </spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>S</spirit:name>…
      <spirit:slave>
        <spirit:bridge spirit:masterRef="M1"/>
```

 © Copyright The SPIRIT Consortium 2006. All rights reserved.

```
            <spirit:bridge spirit:masterRef="M2"/>
            <spirit:bridge spirit:masterRef="M3"/>
          </spirit:slave>
        </spirit:busInterface>
      </spirit:busInterfaces>
      <spirit:addressSpaces>
        <spirit:addressSpace>
          <spirit:name>AS1</spirit:name>
          <spirit:range>0x1000</spirit:range>
        </spirit:addressSpace>
        <spirit:addressSpace>
          <spirit:name>AS2</spirit:name>
          <spirit:range>0x1000</spirit:range>
        </spirit:addressSpace>
        <spirit:addressSpace>
          <spirit:name>AS3</spirit:name>
          <spirit:range>0x1000</spirit:range>
        </spirit:addressSpace>
      </spirit:addressSpaces>
    </spirit:component>
```

4.9.4.3 Memory map in opaque bridge

An opaque bridge is characterized by the `opaque` attribute. In opaque bridges, the address space (AS) is referenced from each master interface in `addressSpaceRef` element, using the `addressSpaceRef` attribute. This attribute references to the `addressSpace` section directly defined under the component. The `addressSpace` element contains a name (referenced from the Masters) and a `range` element.

The memory map (MM) is referenced from each slave interface in the `memoryMapRef` element, using the `memoryMapRef` attribute. This attribute references to the `memoryMap` section directly defined under the component. The `memoryMap` element contains a name (referenced from the Slaves) and a `subspaceMap` element for each master interface defining the base address.

The reference links between master and slave interfaces is described in the figure below.

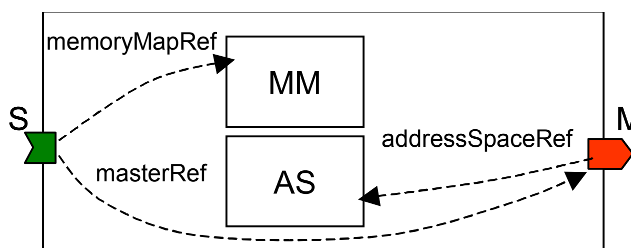


Figure 17 - Memory Map and Address Space links in a Bridge

This memory map in opaque bridge for our example is illustrated in the following figure. The address space (AS) that will be seen from the Master component Master interface will the total of all the address spaces defined on the bus bridge master interfaces (i.e. 12K; from 0x0000 to 0x2FFF).

A sample of the IP-XACT description for such as `memoryMap` is given below:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

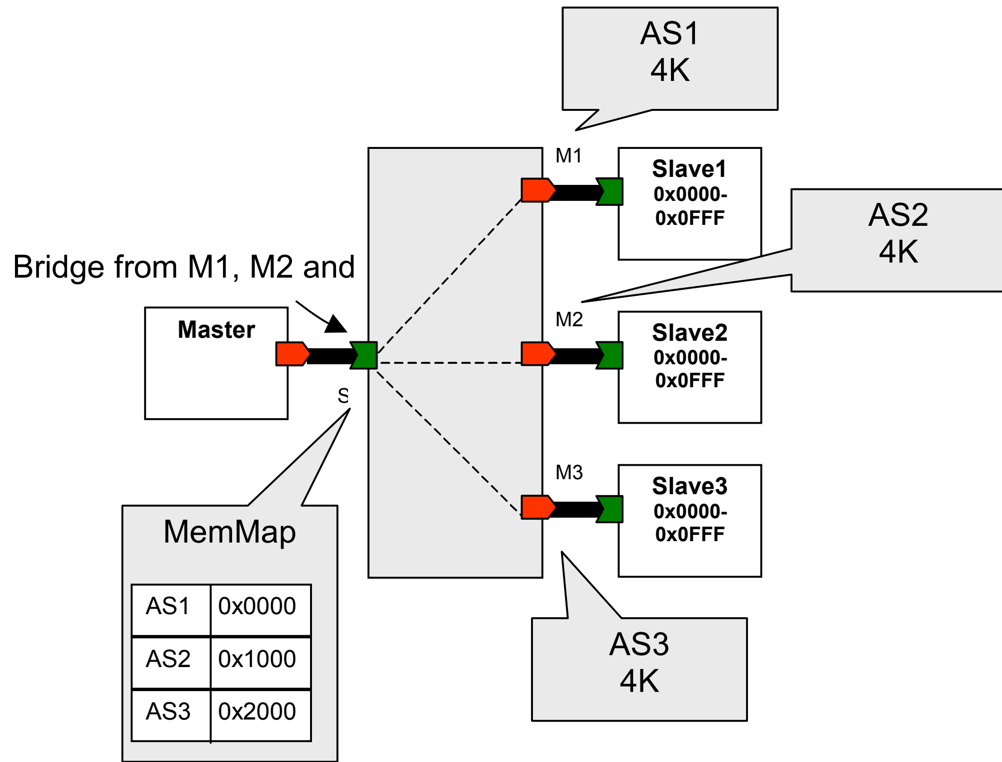


Figure 18 - Memory Map in a opaque Bridge

```
<spirit:component>…
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>M1</spirit:name>…
      <spirit:master>
        <spirit:addressSpaceRef spirit:addressSpaceRef="AS1"/>
      </spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>M2</spirit:name>…
      <spirit:master>
        <spirit:addressSpaceRef spirit:addressSpaceRef="AS2"/>
      </spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>M3</spirit:name>…
      <spirit:master>
        <spirit:addressSpaceRef spirit:addressSpaceRef="AS3"/>
      </spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>S</spirit:name>…
      <spirit:slave>
        <spirit:memoryMapRef spirit:memoryMapRef="memMap"/>
        <spirit:bridge spirit:masterRef="M1" spirit:opaque="true"/>
        <spirit:bridge spirit:masterRef="M2" spirit:opaque="true"/>
        <spirit:bridge spirit:masterRef="M3" spirit:opaque="true"/>
      </spirit:slave>
    </spirit:busInterface>
  </spirit:busInterfaces>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
    <spirit:addressSpaces>
      <spirit:addressSpace>
        <spirit:name>AS1</spirit:name>
        <spirit:range>4K</spirit:range>
      </spirit:addressSpace>
      <spirit:addressSpace>
        <spirit:name>AS2</spirit:name>
        <spirit:range>4K</spirit:range>
      </spirit:addressSpace>
      <spirit:addressSpace>
        <spirit:name>AS3</spirit:name>
        <spirit:range>4K</spirit:range>
      </spirit:addressSpace>
    </spirit:addressSpaces>
    <spirit:memoryMaps>
      <spirit:memoryMap>
        <spirit:name>memMap</spirit:name>
        <spirit:subspaceMap spirit:masterRef="M1">
          <spirit:baseAddress>0x0000</spirit:baseAddress>
        </spirit:subspaceMap>
        <spirit:subspaceMap spirit:masterRef="M2">
          <spirit:baseAddress>0x1000</spirit:baseAddress>
        </spirit:subspaceMap>
        <spirit:subspaceMap spirit:masterRef="M3">
          <spirit:baseAddress>0x2000</spirit:baseAddress>
        </spirit:subspaceMap>
      </spirit:memoryMap>
    </spirit:memoryMaps>
  </spirit:component>
```

4.9.5 Remapping

Remapping can have different meanings.

- In a simple case, it means a logical address change for a slave; with no structural change inside the bus (e.g. no decoding change or no crossbar reconfiguration).
- In a more general case, it means a reconfiguration of the mapping (e.g. change the address decoding depending on the state).

IP-XACT covers all cases.

Like in basic memory map, we have to distinguish two ways to model remapping:

- When using a channel
- When using opaque bridge

Note that remapping in transparent bridge is not meaningful since no address decoding (or demux) is done and therefore the Schema does not allow defining remapping addresses using transparent bridge.

4.9.5.1 Defining conditional states

In IP-XACT, the definition of the legal set of remapping states is separated from the definition of the remapping addresses. The conditional remap states are defined with the `remapStates` element, which is a simple list of `remapState` elements defined

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

under the component. Each remap state is conditioned by a remap signal specified with `remapSignal` element.

Here is an example of how the states might be defined, where a single boolean signal is used to distinguish the two states.

```
<spirit:component>
  <spirit:remapStates>
    <spirit:remapState name="boot">
      <spirit:remapSignal spirit:id="doRemap">true
      </spirit:remapSignal>
    </spirit:remapState>
    <spirit:remapState name="normal">
      <spirit:remapSignal spirit:id="doRemap">false
      </spirit:remapSignal>
    </spirit:remapState>
  </spirit:remapStates >
</spirit:component>
```

4.9.5.2 Defining remap addresses

The list of remap addresses (base address for remapping) is specified under the `component memoryMap` element.

Here is an example of how a remappable memory might be defined.

```
<spirit:component>
…
  <spirit:memoryMaps>
    <spirit:memoryMap>
      <spirit:memoryRemap state="boot">
        <spirit:addressBlock>
          <spirit:baseAddress spirit:format="long">0x0000
          </spirit:baseAddress>
          <spirit:range spirit:format="long">4096</spirit:range>
          <spirit:usage>memory</spirit:usage>
          <spirit:access>read-only</spirit:access>
        </spirit:addressBlock>
      </spirit:memoryRemap >
    </spirit:memoryMap>
  </spirit:memoryMaps>
…
</spirit:component>
```

The following rules apply to define the `memoryRemap` element:

- Each memory Remap is associated with a `state` attribute
- `State` attribute is mandatory
- `State` values must be unique, otherwise illegal
- Any `memoryMap` element without state is assigned a DEFAULT attribute

4.9.5.3 Linking remap States to Memory Map

With such a separation of memory map and states definition, the combination of possible mapping cases between addresses and states becomes difficult to interpret. Therefore some semantic rules are needed.

1) If there are duplicate `state` attributes in different `memoryRemap` tags in the same `memoryMap` section, then only the first occurrence will be recognized.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

In other words, the `state` attribute values of `memoryRemap` should be unique within a `memoryMap` section.

2) If a `component` has no `remapStates` tag specified, then the `memoryMap` is assumed to be in the 'default' state.
3) If a `component` has `remapStates` specified but no `memoryRemap`, then the first `state` listed is synonymous with the 'default' `state`, and will match the `memoryMap` tag with no `state` attribute.

4.9.5.4 Slave interface Remapping

In regular components (including bus Bridges), the list of remap addresses (`baseAddress` for remapping) is referenced through the `memoryMapRef` element under the component slave interface. This referenced memory map can contain a list of `memoryRemap` elements defining the base address for each state.

The Slave `busInterface` element can include reference to multiple `memoryMap` and `state` elements.

4.9.5.5 Mirror slave interface Remapping

A remapping can be defined in the channel's Mirror Slave interfaces, by specifying the list of remap addresses (`baseAddress` for remapping) together with a state identifier. The idea is to leave the slave `memoryMap` unchanged, but to condition the base addresses values in the mirror slave interface. Inside the slave `memoryMap`, the subspace `baseAddress` is relative to the overall `memoryMap's baseAddress`. The `remapAddress` defined in the mirror slave applies to the complete `memoryMap`, not to elements within the `memoryMap`.

Here is an example of how a remappable memory slave might be defined in a channel interface:

```
<spirit:busInterface spirit:id="busInterface_0" spirit:resolve="user">
…
  <spirit:mirroredSlave>
    <spirit:baseAddresses>
      <spirit:remapAddress spirit:state="boot">0x0000
      </spirit:remapAddress>
      <spirit:remapAddress spirit:state="normal">0x1000
      </spirit:remapAddress>
    </spirit:baseAddresses >
  </spirit:mirroredSlave >
  …
</spirit:busInterface>
```

The remap `state` attribute on a `remapAddress` element on a mirrored slave interface is not mandatory.

Note that the `baseAddress` element can be mapped multiple times according to the states and may appear multiple times in the same `state` attributes.

4.9.5.6 Example of memory remapping with a channel

Let's take again our example of a single master connected to 3 memories (of 4K each) through a bus channel. This bus supports two memory map states: init mode and user mode.

In init mode, the three memories are respectively mapped at addresses 0x0000-0x0FFF, 0x1000-0x1FFF and 0x2000-0x2FFF. If the master writes at address 0x1900, it will hit the second memory.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

In user mode, activated by a doRemap signal, the second memory base address and address range is changed from 0x1000, 4K to 0x3000, 4K. Now, if the master is writing at the same address 0x1900, it will get an error because there is no addressable slave at this address. The new address space after remapping is shown in Figure 20.

A sample of the IP-XACT description for such as remapping of slave2 is given below. The XML code is very similar to the simple memory map in channel example. The only changes here are in the `busInterface` element of the second Mirrored slave (MS2) and the addition of the `remapStates` section.

```
<spirit:component>…
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>MS1</spirit:name>…
      <spirit:mirroredSlave>
        <spirit:baseAddresses>
          <spirit:remapAddress>0x0000</spirit:remapAddress>
          <spirit:range>0x1000</spirit:range>
        </spirit:baseAddresses>
      </spirit:mirroredSlave>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>MS2</spirit:name>…
      <spirit:mirroredSlave>
        <spirit:baseAddresses>
          <spirit:remapAddress spirit:state="init">0x1000
          </spirit:remapAddress>
          <spirit:remapAddress spirit:state="user">0x3000
          </spirit:remapAddress>
          <spirit:range>0x1000</spirit:range>
        </spirit:baseAddresses>
      </spirit:mirroredSlave>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>MS3</spirit:name>…
      <spirit:mirroredSlave>
        <spirit:baseAddresses>
          <spirit:remapAddress>0x2000</spirit:remapAddress>
          <spirit:range>0x1000</spirit:range>
        </spirit:baseAddresses>
      </spirit:mirroredSlave>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>MM</spirit:name>..
      <spirit:mirroredMaster/>
    </spirit:busInterface>
  </spirit:busInterfaces>
  <spirit:channels>
    <spirit:channel>
      <spirit:busInterfaceRef>MS1</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS2</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MS3</spirit:busInterfaceRef>
      <spirit:busInterfaceRef>MM</spirit:busInterfaceRef>
    </spirit:channel>
  </spirit:channels>
  <spirit:remapStates>
    <spirit:remapState spirit:name="init">
      <spirit:remapSignal spirit:id="doRemap">true
      </spirit:remapSignal>
    </spirit:remapState>
```

© Copyright The SPIRIT Consortium 2006. All rights reserved.

```
        <spirit:remapState spirit:name="user">
          <spirit:remapSignal spirit:id="doRemap">false
          </spirit:remapSignal>
        </spirit:remapState>
      </spirit:remapStates>
    </spirit:component>
```

4.9.5.7 Example of memory remapping with a bus bridge

Let's take again our example of a single master connected to 3 memories (of 4K each) through a bus bridge. The master writes at address 0x1900 and thus hits the second memory (slave2). The following figure shows the address space as seen from each master interface.

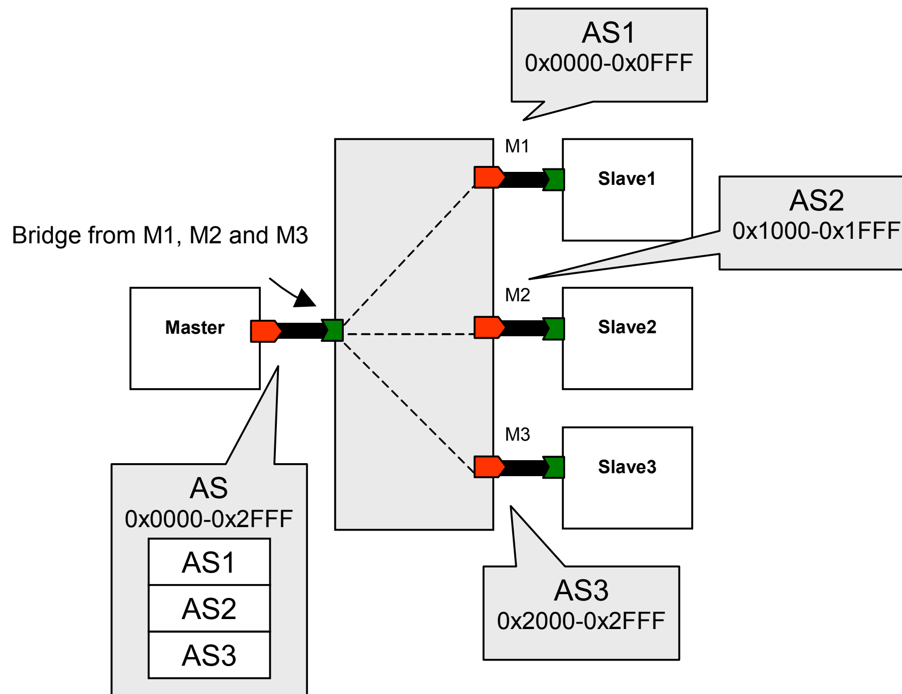


Figure 19 - Before Remapping

After remapping (e.g. activated by a `doRemap` signal), the second memory base address and address range is changed from 0x1000, 4K to 0x3000, 4K. Now, if the master is writing at the same address 0x1900, it will get an error because there is no addressable slave at this address. The new address space after remapping is shown in the following figure.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

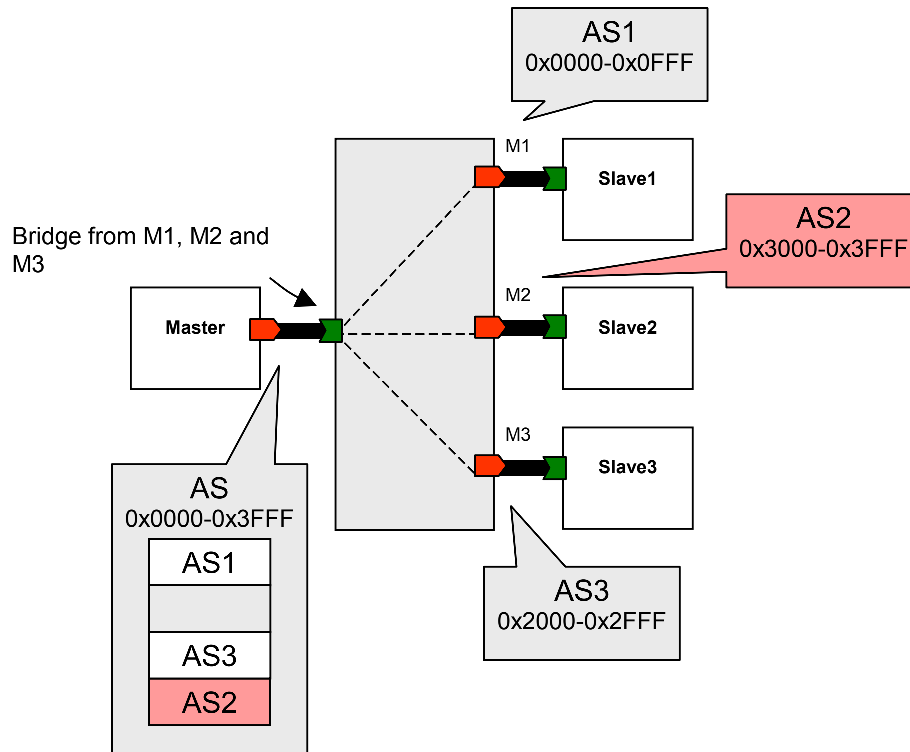


Figure 20 - After Remapping

A sample of the IP-XACT description for such as remapping of slave2 is given below. The `busInterfaces` section and the `addressSpace` section are exactly the same as for the simple memory map in opaque bridge example. The `remapState` and `memoryMap` sections are only shown here.

```
<spirit:component>…
   <spirit:busInterfaces>…
   </spirit:busInterfaces>
   <spirit:addressSpaces>…
   </spirit:addressSpaces>
   <spirit:remapStates>
    <spirit:remapState spirit:name="init">
      <spirit:remapSignal spirit:id="doRemap">true
      </spirit:remapSignal>
    </spirit:remapState>
    <spirit:remapState spirit:name="user">
      <spirit:remapSignal spirit:id="doRemap">false
      </spirit:remapSignal>
    </spirit:remapState>
   </spirit:remapStates>
   <spirit:memoryMaps>
    <spirit:memoryMap>
      <spirit:name>memMap</spirit:name>
      <spirit:memoryRemap spirit:state="init">
        <spirit:subspaceMap spirit:masterRef="M1">
          <spirit:baseAddress>0x0000</spirit:baseAddress>
        </spirit:subspaceMap>
        <spirit:subspaceMap spirit:masterRef="M2">
          <spirit:baseAddress>0x1000</spirit:baseAddress>
        </spirit:subspaceMap>
        <spirit:subspaceMap spirit:masterRef="M3">
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
          <spirit:baseAddress>0x2000</spirit:baseAddress>
        </spirit:subspaceMap>
      </spirit:memoryRemap>
      <spirit:memoryRemap spirit:state="user">
        <spirit:subspaceMap spirit:masterRef="M1">
          <spirit:baseAddress>0x0000</spirit:baseAddress>
        </spirit:subspaceMap>
        <spirit:subspaceMap spirit:masterRef="M3">
          <spirit:baseAddress>0x2000</spirit:baseAddress>
        </spirit:subspaceMap>
        <spirit:subspaceMap spirit:masterRef="M2">
          <spirit:baseAddress>0x3000</spirit:baseAddress>
        </spirit:subspaceMap>
      </spirit:memoryRemap>
    </spirit:memoryMap>
  </spirit:memoryMaps>
</spirit:component>
```

4.9.6 Signal Connections

4.9.6.1 signalMap between component physical signal and busInterface logical signal

The connection between component physical signals and the bus interface logical signals is defined in the signal map of the component busInterface. Each signal defined in the bus interface can be assigned `left` and `right` elements to represent bit slices of a vector. The `left` element means first boundary, the `right` element second boundary. There is no assumption that left is larger than right nor that left is MSB and right is LSB.

The `left` and `right` elements are the (bit) rank of the leftmost and rightmost bits of the signal.

For example, to map a component 16-bit address signal (addr) to an 8-bit width address busInterface logical signal (addrL), one would write:

In the component signal declaration:

```
<spirit:signal>
   <spirit:name>addr</spirit:name>
   <spirit:direction>out</spirit:direction>
   <spirit:left>15</spirit:left>
   <spirit:right>0</spirit:right>
</spirit:signal>
```

In the busDefinition:

```
<spirit:signal>
  <spirit:logicalName>addrL</spirit:logicalName>
  <spirit:onMaster>
    <spirit:direction>out</spirit:direction>
    <spirit:bitWidth>8</spirit:left>
  </spirit:onMaster>
</spirit:signal>
```

In the component busInterface signalMap:

```
<spirit:signalMap>
  <spirit:signalName>
    <spirit:componentSignalName>addrL</spirit:componentSignalName>
    <spirit:busSignalName>addr</spirit:busSignalName>
```

 © Copyright The SPIRIT Consortium 2006. All rights reserved.

```
      <spirit:left>15</spirit:left>
      <spirit:right>8</spirit:right>
    </spirit:signalName>
  </spirit:signalMap>
```

Meaning that:

```
addr[15]  → addrL[7]
addr[14]  → addrL [6]
addr[13]  → addrL [5]
...
addr[7]   → addrL [0]
```

The *busDef* has an optional *bitWidth* element in it and this must match the width specified in the *signalMap* element if present, but if the component physical signal vector has exactly the same width as the logical signal, then it is not needed to specify the *left* and *right* elements in the busInterface signalMap.

4.9.6.2 Connection between vector signals of two busInterfaces

When connecting two busInterfaces in a design (i.e. their VLNV match), this can be straightforward if the two component physical signals have the same width and same left and right values, but more complicated in other cases. Here follow some specific connection rules to cover the different connection scenario.

When connecting two signal vectors (v1 and v2) from respectively a component busInterface I1 and I2, we have to distinguish the following cases:

(1) If the two vectors connected have the same width, and the same left and right values (e.g. v1 has left=3 right=0 and signal v2 has left=3 right=0) then do a direct bit to bit connection

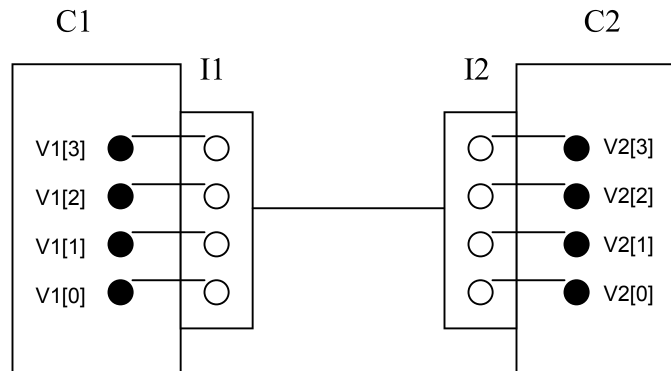


Figure 21 - Vector connection of equal width and indices

(2) If the two vectors connected have the same width, but the left and right values are opposite ways round(for example component signal v1 has left=3 right=0 and signal v2 has left=0 right=3), then reverse the connection; i.e. connect the bits one by one from left down to right (e.g. vectors v1[3] to v2[0], v1[2] to v2[1], v1[1] to v2[2] and v1[0] to v2[3])

© Copyright The SPIRIT Consortium 2006. All rights reserved.

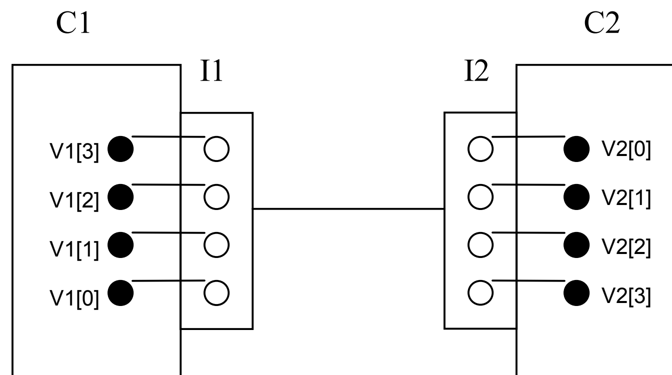


Figure 22 - Vector connection of equal width and non-equal indices

(3) If the two vectors connected have a different width but either the same left or right values (e.g. vectors v1[5:0] connecting to v2[7:0] connected through a busInterface of 4 bits with left=3 and right=0 on both sides), then do a direct bit to bit connection from left down to right.

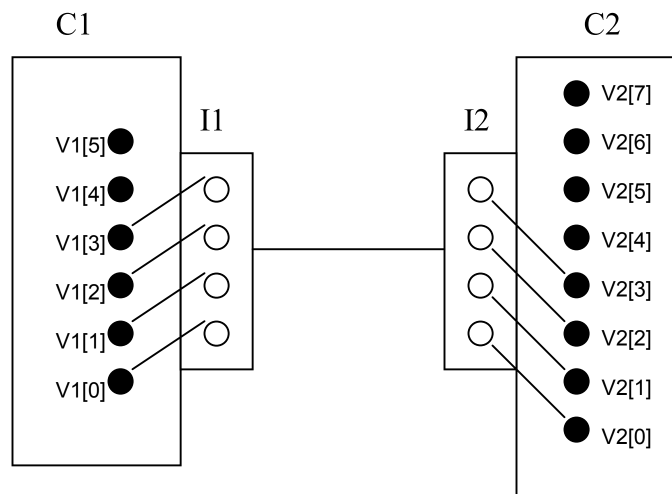


Figure 23 - Vector connection of non-equal width and common indices

(4) If the two vectors connected have the different a width, with different left and right values (for example component signal v1 has left=5 right=2 and busInterface signal v2 has left=4 right=1), then connect the bits one by one from left down to right (e.g. vectors v1[5:2] connecting to v2[4:1]. If they are also opposite ways round, then reverse right and left on the master or mirrored slave before connecting the bits.)

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

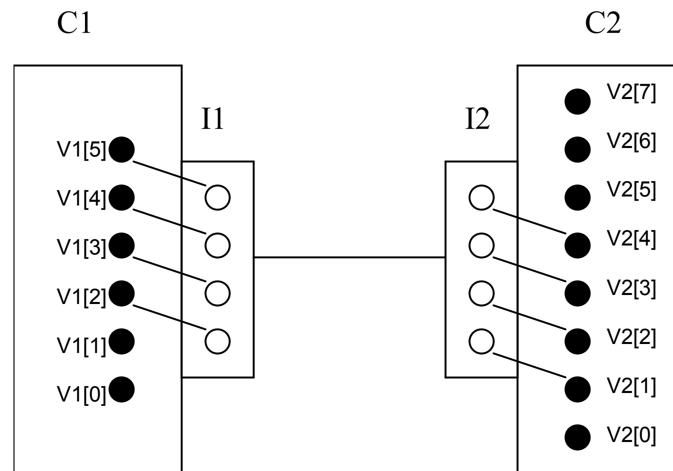


Figure 24 - Vector connection with non-equal width and non-equal indices

4.9.6.3 Connection of Ad-hoc signals

The name "ad-hoc" is used for signals that do not belong to bus interfaces.

These signals can be, for example, wires that are internal to a component, or clock signals (when not defined as System Interface).

Such signals, which are not part of a bus interface, have to have an explicit definition (e.g. to specify if they are exported, specify their names, direction…). In addition, these signal definitions should be consistent with the left and right element definitions for both the interface signal to component signal.

Limitation: ad-hoc connections only are possible at the design level (not in hierarchical components).

For ad-hoc wires, IP-XACT requires that the sizes of each pin (`left - right` + 1) are exactly the same and that bits are connected from left to right with no exceptions. In the `pinReference` element, left and right do not define the size of the pin, just the portion that is connected. An example follows:

```
<spirit:adHocConnection>
  </spirit:pinReference componentRef="U1" signalRef="A" left="8"
right="1">
  </spirit:pinReference componentRef="U2" signalRef="B" left="7"
right="0">
</spirit:adHocConnection>
```

It would imply that:

```
U1/A[8] → U2/B[7]
U1/A[7] → U2/B[6]
U1/A[6] → U2/B[5]
U1/A[5] → U2/B[4]
U1/A[4] → U2/B[3]
U1/A[3] → U2/B[2]
U1/A[2] → U2/B[1]
U1/A[1] → U2/B[0]
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

4.9.7 Clock and Reset Handling

USAGE NOTE: IP-XACT does not dictate the mechanisms for describing Reset and Clock pins in relation to bus definitions. For example, Reset and Clock are part of several bus-interface standard definitions (e.g., AMBA). To that end, consistency with the specification implies that Reset and Clock would be included in the busDefinition. However, Reset and Clock are generally routed separately from buses (e.g., A bus master does not generally drive the clock of a bus slave). Some tools will need to extract these pins from those associated with a bus.

The provider of a bus standard dictates the style of busDefinition to be used for IP supporting those protocols. These must be published and freely available, and clear about whether Reset and Clock pins are included in the busDefinition.

This section compares the three ways IP-XACT can handle clocks and resets - for simplicity only clock are mentioned hereafter, but the discussion applies to both - with respect to the following three main features

1. Misconnection detection. IP-XACT only allows interconnection of bus interfaces with compatible types, which can be used to prevent connecting clocks to non-clock signals.
2. Description of the relation of the clock with other signals. Several applications – e.g. verification, timing – require identifying the reference clock of signals, which can be provided by one of the approach listed hereafter.
3. Clock distribution control. Traditionally, in HDL, clocks have been handled separately from data to control explicitly the clock distribution. IP-XACT allows to combine clock with data and to make clock distribution implicit.

4.9.7.1 Clock Outside a Bus Interface

A clock, like any signal, can be handled as a `signal` element that does not belong to any bus interface.

The main features of the approach are:

1. No protection against connecting a clock to a non-clock signal. An `adhocConnection` element can connect a signal clock to any other signal even if is not a clock.
2. No possibility to describe of the relation the clock with other signals.
3. Explicit control on the clock distribution. The clock distribution is done through `adhocConnection` element connection the clock output signal to the clock inputs.

This approach should only be used with clocks that do not relate to signals in a bus interface either because it does not or because the bus interface has not been created (e.g. because the corresponding bus definition does not exist).

4.9.7.2 Clock in Dedicated Bus Interface

A clock can be handled as a clock bus interface that contains one signal i.e. the clock.

The main features of the approach are:

1. Protection against connecting a clock to a non-clock – the bus interface types would not match.
2. No possibility to describe the relation of the clock with other signals.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

3. Explicit control on the clock distribution. The clock distribution is done through `interConnection` element between the clock inputs and outputs.

With this approach, the following additional features need being taken into account:

- A clock tree may have to drive clocks with incompatible bus types. At this time, IP-XACT does not provide a standard clock bus definitions and it up to the IP packager to provide/choose this definition, which can lead to clock trees driving clock bus interfaces of different types. These incompatibilities must be handled when assembling the design.
- A clock tree drives several clock input from a single output whereas SRIRIT interconnections only allow a bus interface to "drive" one other interface. This can be handled either with a clock-broadcasting component or by creating as many "output" bus interface, as needed sharing the same "output" signal.
- Simulations can be very sensitive to delta cycles in a clock tree. Care should be taken to avoid/handle the delta cycle that could be introduced by a clock-broadcasting element.

Two variations of this approach are possible:

1. A dedicated clock bus type could be used for the interface. This is likely to be appropriate when the clock is, for example, a system clock not associated with any particular bus type.
2. The bus type of the interface could match that of some data bus in the system. This may be appropriate when the clock is associated with a particular bus type, but is routed differently from the data signals for that bus. In this case the clocking bus interface would normally be a system interface.

4.9.7.3 Clock in Regular Bus Interface

A clock signal can be handled as one of the signal of a bus interface.

The main features of this approach are:

1. Protection against connecting a clock to a non-clock – the bus interface types would not match.
2. Description of the relation of the clock to the other signals of the bus interface.
3. Implicit control on the clock distribution. The clock distribution is done through `interconnection` element with the signals of the bus interface.

With this approach, the following additional features need being taken into account:

- The same clock can clock several bus interfaces. The clock can be shared by more than one bus interface.
- A clock distribution policy must be defined The clock being distributed along the data signal, a policy must be defined to guarantee that a clock can provided to the bus interface that need them e.g. that clock output must be provided on each mirroredMaster, mirroredSlave to drive the clock input of each Master/Slave.
- Complex clocking scheme – e.g. extensive clock gating for low power – may require extensive support from the clock distribution IP.

4.9.8 Bus interface parameter declaration

The section of IP-XACT XML hereafter illustrates how to declare, in a bus definition, the names and legal values that parameters on bus interfaces of that type can take.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
<spirit:busDefParameters>
  <spirit:busDefParameter name="BDparam1" spirit:minimum="0"
     spirit:consistent="true"/>
  <spirit:busDefParameter name="BDparam2"
     spirit:choiceRef="BDchoiceRef">
    <spirit:defaultValue>False</spirit:defaultValue>
    <interfaceType>master</interfaceType>
    <interfaceType>mirroredMaster</interfaceType>
  <spirit:busDefParameter>
</spirit:busDefParameters>
```

Each bus interface parameter is declared with a `spirit:busDefParameter` element specifying:

- The parameter name
- The legal values – using the same constraints attributes as regular SPIRIT parameters -,
- Whether the value of the parameter must be consistent – equal – on interconnected bus interfaces.
- The default value.
- The type(s) of bus interface(s) on which the parameter can be found – all by default -.

4.9.9 Bus interface parameter

The section of SPIRIT XML hereafter illustrates how to specify, in a bus interface, the values of the parameters of the bus interface.

```
<spirit:busInterfaceParameters>
  <spirit:busInterfaceParameter
     name="BDparam1">1</spirit:busInterfaceParameter>
  <spirit:busInterfaceParameter name="BDparam2"
     spirit:resolve="user"
     spirit:id="BDparam2ID">True</spirit:busInterfaceParameter>
</spirit:busInterfaceParameters>
```

Each bus interface parameter value is defined separately with a `spirit:busInterfaceParameter` element specifying:

- The parameter name – it must be one the name declared in the busDefParameters of the bus definition matching the busType of the interface -.
- The parameter value – it must comply with the constraints defined in the busDefParameters of the bus definition matching the busType of the interface -. It can be configured with IP-XACT configuration mechanism.

4.10 Reference Bus Definitions

The SPIRIT Consortium has prepared a set of BusDefs for several common busses. It is expected, over time, that those standards groups and manufacturers who define

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

busses will include IP-XACT XML BusDefs in their set of deliverables. Until that time, and to cover existing useful busses, a set of BusDefs for common busses has been created.

Having a set of reference BusDefs means that many vendors defining IP using these busses can interconnect. The SPIRIT Consortium posts these for use by their members, with no warrantee of suitability, but in the hope that these will be useful. The SPIRIT Consortium will from time to time, update these files and if a Standards body wishes to take over the work of definition, will transfer that work to that body.

4.10.1 The difference between an external bus and an internal/digital interface

While the current use of IP-XACT schema may be viewed as describing single chip implementations, the schemas works equally well at the package and board level. Often a PHY component exists which interconnects the internal and external bus. Some standards define both of these interfaces, some define only the internal and some define only the external. A common point of confusion is to use an external bus standard as an interface on an internal component. This is legal if the component caries the full PHY implementation, but often will make the component very technology/implementation dependant. Some examples of busdef families that include both interfaces are listed below:

4.10.1.1 Example: Ethernet Interfaces:

An Ethernet “bus” would not be only described as a single wire, but in a system that includes Ethernet busses, it may also include, for example:

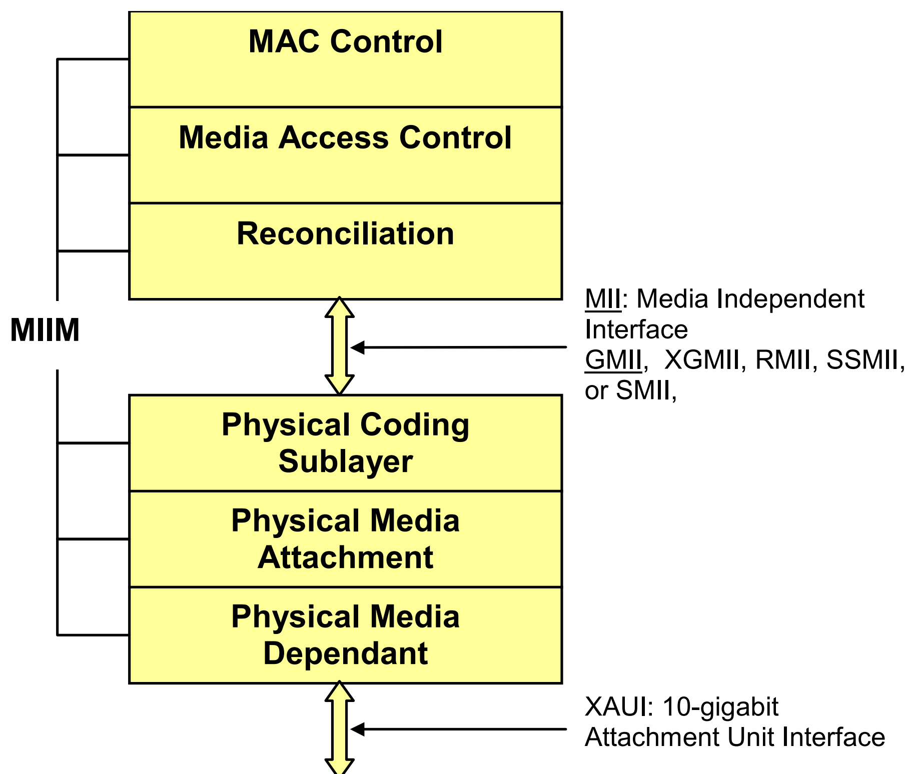


Figure 25 - Ethernet Interface Examples

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

XAUI: 10-gigabit Attachment Unit Interface

MII: Media Independent Interface

GMII: Gigabit Media Independent Interface

XGMII: 10-gigabit media-independent interface

RMII: Reduced MII, 7-pin interface

SSMII: Source Synchronous MII

SMII: Serial Media Independent Interface, The Serial Media Independent Interface [SMII] provides an interface to Ethernet MAC. The SMII provides the same interface as the Media Independent Interface [MII] but with a reduced pinout. The reduction in signals is achieved by multiplexing data and control information to a signal transmit signal and a single receive signal.

4.10.1.2 Example: I^{2}C Bus

The I^{2}C “eye-squared-see” bus is a two-wire bus with a clock and data line. The standard described bus is the two-wire bus. The SPIRIT Consortium has defined an additional, related bus which is the internal digital interface. This reference BusSpec contains three pins for each external pin: for SDA (the data line) the internal pins are defined as input, output, and enable as SDA_I, SDA_O, and SDA_E, in a similar manner for the clock bus: SCL the internal pins are defined again for the functions of input, output, and enable as SCL_I, SCL_O, and SCL_E

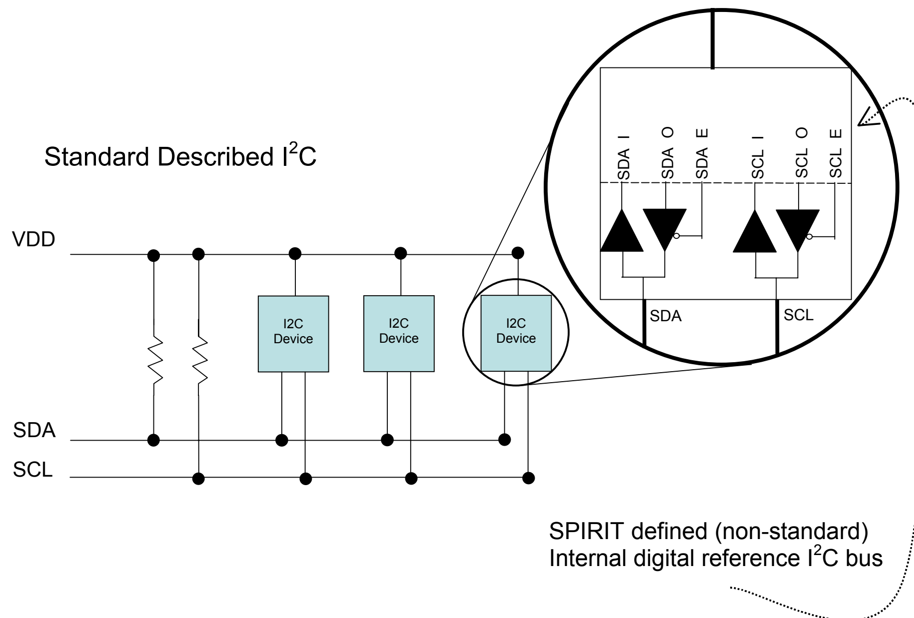


Figure 26 - I2C Interface Example

4.10.2 Location of reference BusDefs

The reference BusDefs are available from the public area of the spiritconsortium.org web site.

4.10.2.1 Reference BusDef template

A BusDef template with comments and examples is located with the reference BusDefs also available from the public area of the spiritconsortium.org web site.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

5 IP-XACT GENERATORS

IP-XACT defines a tool integration API that provides a standard method for linking tools into an IP framework, enabling a more flexible, optimized development environment. IP-XACT-enabled tools are able to interpret, configure, integrate and manipulate IP blocks that comply with the IP meta-data description.

The API allows the querying of XML IP meta-data that has been imported into the design-environment. Queries may be for the existence of IP, the structure of IP, or features offered by that IP such as configurability and interface protocol support. This API is also used for the import and export of meta-data when an IP block is extracted from, or imported back into, the IP management system.

Another role for the API is to interface to generators and tool plug-ins, allowing the execution of these scripts and code-elements against the SoC meta-description. The API enables the registration of new generators / plug-ins, export of SoC meta-data and update of that data following generator or plug-in execution, and handling of generator / plug-in error conditions which relate to the meta-data description.

5.1 Generator registration

Generators can be registered with the DE in components and generatorChains. When registering a generator extra information can be supplied to allow the DE to optimise the amount of information that it needs to write out before the generator can be invoked. This extra information includes:

Read-only flag: If this is set then the DE need not worry about the generator changing the database. This allows the DE to invoke the generator in a separate thread, if this is possible.

Hierarchy flag: This flag allows the DE to trim the number of levels of hierarchy that it needs to write out for the generator to work correctly. Currently the flag indicates that only the top-level information is required or all hierarchical levels are required.

Instance flag: This flag indicates to the DE that the generator requires the name of at least one instance to work upon. A list of instances can also be supplied. It is the DE's responsibility to collect this list of instances before invoking the generator.

Subset flags: There are some flags that allow further trimming of the information that the DE must provide. The *design file* flag indicates that the generator knows that all the information it requires is contained within the design file so the DE can just make this available. The *component definitions* flag indicates that the generator requires only the component definition files to work upon. Finally the *bus definitions* flag indicates the same for bus definitions.

Generator name element collisions are allowed and assumed to refer to the same generator.

5.2 Tight Integration

In IP-XACT terminology a tight integration of an API means the direct interfacing to generators and XML meta-data within the Design environment. An API can manipulate values of elements, attributes and parameters for IP-XACT compliant XML and save any modified data in a persistent way.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

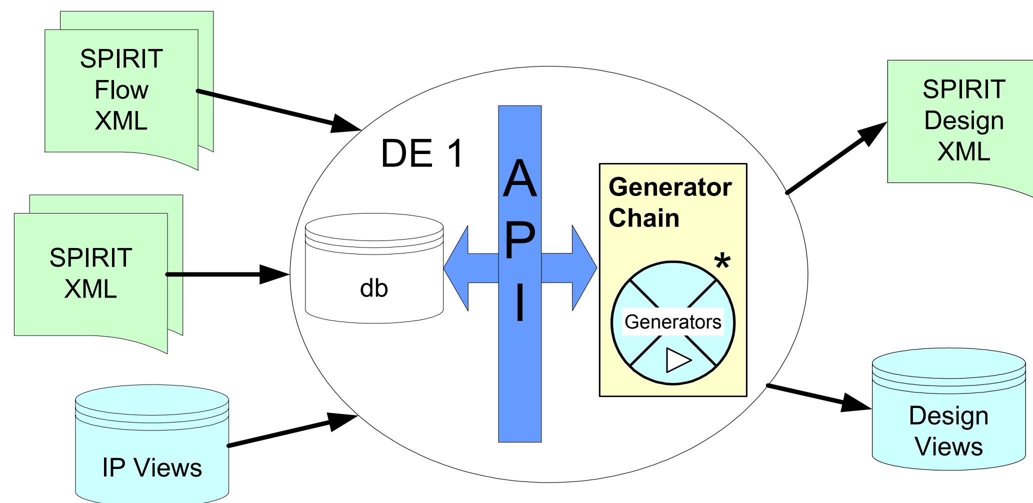


Figure 27 - Example of Tight Integration Flow

The design environment reads the XML input files in and the internal Database representation is accessed via an API. This API is also used to supply parameters for generators and execute them. The results of generators can be used to update the DB until the design and all its configurable parameters are finally saved to an XML file. More information on the tight generator interface is available in a separate document from the spiritconsortium.org web site.

http://www.spiritconsortium.org/releases/tgi/index.html

5.3 Loose Integration

5.3.1 Definition

In IP-XACT terminology a loose integration of an API means the indirect interfacing to generators and XML meta-data outside of the Design environment. An API can still manipulate values of elements, attributes and parameters for IP-XACT compliant XML and save any modified data in a persistent way but this time via explicitly generated XML and transformation data that another design environment can understand.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

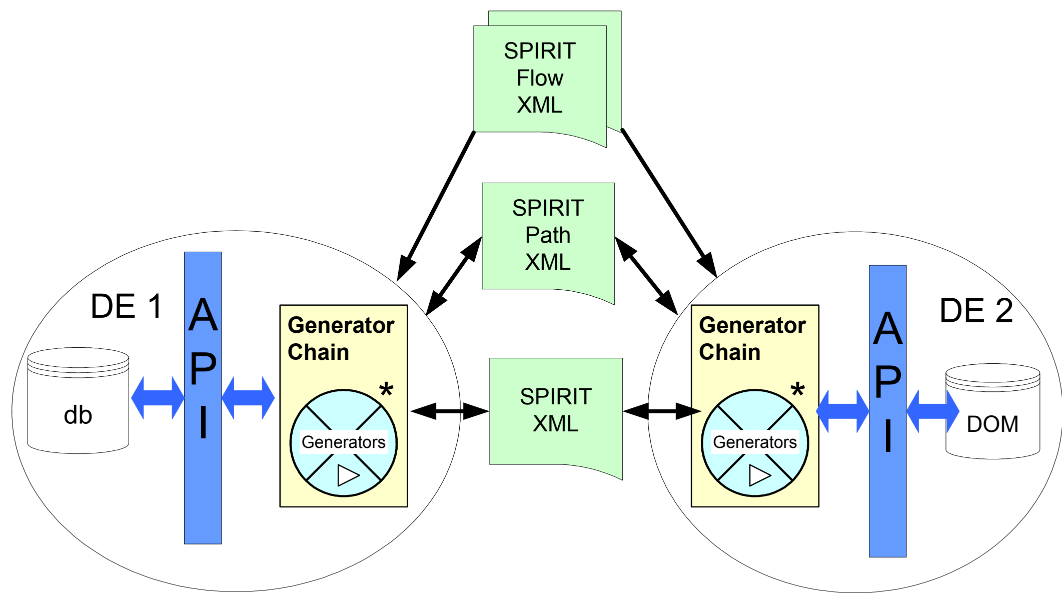


Figure 28 - Example of Loose Integration flow

Here the XML files need to be exported and imported as a result of running generators. The API is external and is not bound by choice of language or condition that is characteristic of the tightly integrated solution.

Note that the loose generator interface is deprecated in IP-XACT 1.2, and will be removed in a future version of the IP-XACT specification.

5.3.2 Typical DE flow

This section illustrates a typical DE flow with call to a generator using the *loose generator interface* (LGI). The diagram below captures the essence of a loose generator call.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

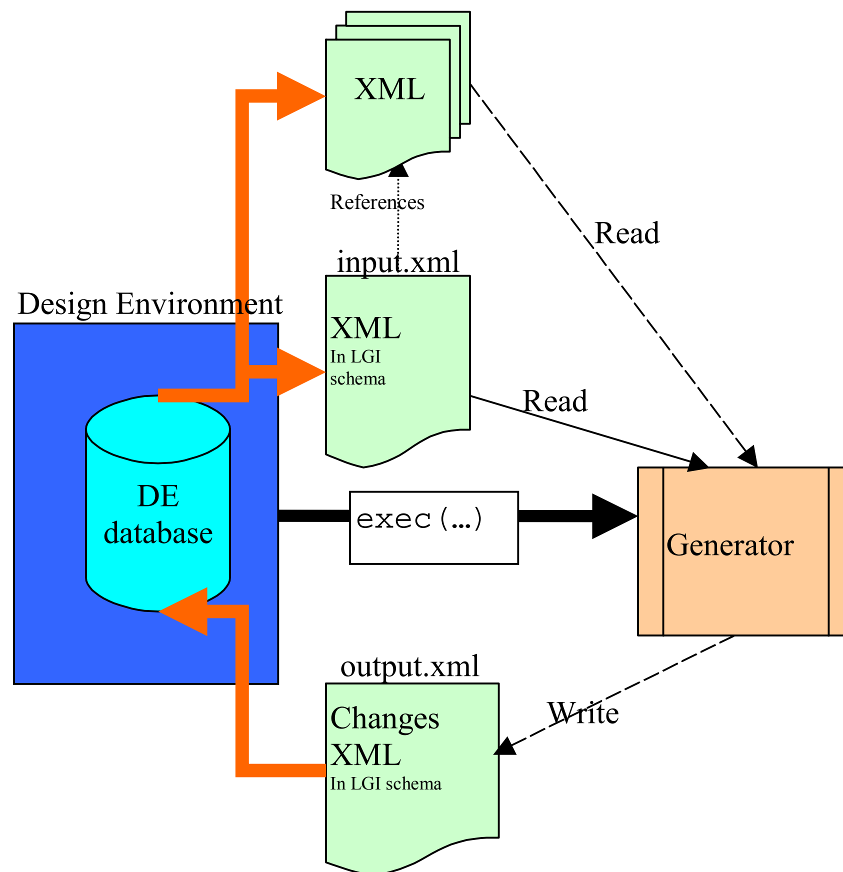


Figure 29 - Typical DE Flow

5.3.2.1 Design environment flow

- Somewhere a generator is invoked, probably from a generator chain or maybe directly by the user pressing a 'build' button in the DE.
- The DE examines what requirements the generator has defined and writes out files including the bus definitions, the current component definitions and the design file.
- The DE puts above file paths into `input.xml` according to the LGI schema.
- The DE prepares return file, 'output.xml'. For example, it may be that this file still exists from a previous generator run, so must be deleted.
- The DE then calls the underlying OS's `exec("generator_name", "input.xml", "output.xml")` function call and the generator flow then starts (described below). Depending on the generator the DE may also have to capture the generator's stdout/stderr streams and wait until the generator process completes.
- The DE then examines the generator return status and can take action at this point to handle generator failure.
- The DE reads the `output.xml` file if the generator returned a successful exit status and the generator has registered itself as being one that can modify the database, i.e. not a read-only generator. The DE then executes any database modification elements in that file.

5.3.2.2 Generator flow

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Generator reads `input.xml` and then any other files that it might need, all of which should be referenced by this `input.xml` file.

The generator should then parse the XML and create an internal database, if required. This database should follow the same IP-XACT semantic interpretation as the DE otherwise there will be differences between the DE's interpretation of the design and that of the generator. The generator can then act upon this database and write out any required changes to `output.xml`.

The generator then exits with appropriate success/fail return value.

5.3.2.3 Generator input files

The DE is responsible for interpreting the generator's requirements and writing out enough information for the generator to process. This is typically a two-stage process. The first stage is to write out the files that describe the parts of the design that the generator needs, component definitions, bus definitions, etc. The second is to write a file that references these files, the generator invocation file.

There are some things that the DE must take note of when this generator invocation file is written. Firstly, in order to link component definitions to their instances each component definition file element has an instanceRef attribute and this should be filled in with the instance name of the instance in the design that this file defines. The DE may place these files wherever it chooses, as long as they are accessible to the generator on the local filesystem. However, there is a catch in that files referenced in these component definitions must also be available to the generator on the local filesystem. So the DE must ensure that the versions of component definition files it writes have these file references fully expanded. The DE is also responsible for capturing the values of any parameters associated with the generator and writing these to the invocation file.

5.3.2.4 Generator changes

The generator may write out changes that it wishes to make to the design in a limited fashion. The limits are imposed by the looseGeneratorChanges schema. These limits are imposed since it reduces the burden on the DE when it comes to making changes to its (master) database.

The generator may add/remove/replace components, add/remove/replace interconnections, add/remove/replace configuration settings and add/remove vendor specific data.

Component changes are the most complex. Removal of an instance just requires the instance name. Adding or replacing an instance requires the instance name, the component definition file and, if this is a hierarchical component, then the generator must also supply the sub component definitions. Also the configuration of an instance can be supplied.

Note: to change the configuration associated with an instance, the generator must replace the instance with *itself* and supply the new configuration information.

5.3.2.5 Rules for interpreting changes

- Adding components: The instance name should be unique; the DE will reject the change if not. The component file must be supplied. If the component is hierarchical then subcomponents must be supplied. Note that components can only refer to existing bus definitions since there is no way to add new bus definitions from a generator.
- Removing components: Only the instance name need be supplied. The interconnections must be deleted by the generator if not required.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Interconnections that do not have a bus interface at both ends are not allowed.

- Replacing components: There are two expected uses for replace: 1) to change the configuration of an existing component and 2) to allow the transformation of a component.
- The difference is whether or not a new component definition file is supplied. If not then the generator is trying to modify the existing component's configurable information and this requires little effort on the DE's part to handle.
- If there is a new component definition supplied then the DE has to work a bit harder to maintain the database. The component's configuration info will be supplied by the generator and so the DE must use this. The DE should maintain the connectivity with existing interconnections where they refer to identically named interfaces on the old and the new component. The generator must delete interconnections that now refer to interfaces that no longer exist, since interconnections that do not have a bus interface at both ends are not allowed.
- Adding interconnections: New interconnections must refer to existing bus interfaces at both ends. The DE can reject the change if this is not true. Note that it is not an error for the generator to define a new interconnection where one already exists between the two interfaces.
- Removing interconnections: The generator must ensure that the referenced interconnection exists; it is an error to specify an interconnection that doesn’t exist.
- Replacing interconnections: This is equivalent to a *remove* followed by an *add* operation.
- Adding configuration settings: New settings must refer to existing configurable elements or instance and view pairs.
- Removing configuration settings: The configuration setting with the given referenceId and value must exist in the database to be deleted. If the configuration change refers to an instance’s view then that instance must exist.
- Replacing configuration settings: This is equivalent to a *remove* followed by an *add* operation.
- The rules for adding, removing and replacing vendor-defined data will depend on the type of data. The DE will know what data it can and cannot handle in this area.

5.3.2.6 Error handling

Error handling is DE specific, some DE's may apply all changes in an atomic manner, others may apply the changes until the first failure and then stop, and other DEs may apply all changes despite previous failures. It is recommended that DEs do not allow the database to become so corrupted by failure that manual editing is impossible. Ideally the atomic change strategy (the changes are accepted only if all changes are successful) should be the preferred approach.

5.3.2.7 Generator environment

Generators written to work with the LGI must obey certain rules. They must accept one or two command line arguments; the first one is mandatory since it defines the XML file that references any further files. The second is optional and may not be supplied to generators that have registered themselves as being read-only. This second argument is the name of the file in which generators must write any modifications that they wish to make to the DE's master database.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Generators must be written in a manner that is directory structure independent. The DE is responsible for invoking the generator and can do this in any directory it chooses, the generator cannot rely on this directory being the same between all DEs nor even two runs in the same DE.

5.3.2.8 Shared responsibilities

Both the DE and generators have a number of other rules to adhere to.

The DE must supply absolute pathnames in definition files. The generator must do similarly in any definition files it returns.

Additionally, the DE should write out component definitions as they currently appear in the DE's database. This is so that configured or transformed definitions are seen by the generator too. The generator must assume that all component and bus definition files are read only to allow the DE to write out definitions once and subsequently refer to the previously written definition thereafter.

DEs must note that a replace is not the same thing as a delete followed by an add. The DE should try to replace the component in-situ rather than deleting all the connectivity associated with that component. Generators rely on this connectivity being retained for further processing.

The order for processing changes is: component changes followed by interconnection changes followed by project setting changes and finally persistent data changes. There is also an order within each of these subsections. Both DEs and generators should follow this order. If this order is too restrictive to a generator then it must be split into as many parts as are needed so that the prescribed order of processing can be followed by the DE. This splitting may be into a generator chain.

5.3.3 Configurators

In many cases the LGI will be use to handle configuration tasks through configurators, validators or hooks. In these cases the generator should be marked, where possible read-only and the registration flags should be set to allow the DE to minimise the amount of data it needs to make available to the generator through the LGI.

5.4 Generator chain

In the current IP-XACT methodology a design flow can be represented as a generator chain that links an ordered sequence of named tasks. Each named task can be represented as a single generator or itself be the name of a generator chain. In this way design flow hierarchies can be constructed and executed from within a given design environment. The design environment is responsible for understanding the semantics of the specified chain described in the XML schema.

The current IP-XACT schema defines the generator group definition and elements in the `generator.xsd` file.

In the current terminology:

- A generator chain is a sequential list of ordered generator groups
- A generator group is a named generator that contains a sequential list of generator invocations
- A generator invocation is a method of running an application at a defined phase in the generator group with a given number of parameters
- A phase is a number that defines when a generator invocation occurs in a sequential ascending order.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

- A number of parameters support the ability to influence the behavior of the generator invocation

With these definitions the names of generators should reflect what they are trying to achieve. While the generator group names have string values and are therefore generic, one of The SPIRIT consortium goals is to allow plug-and-play-like architectures for generators. To help facilitate this, the following recommendations have been made to both give examples of generator naming and guidelines for generator usage.

5.4.1 Generator Naming Convention

The recommendation from The SPIRIT Consortium is to classify different parts of a generator name into <owner>_<target>_<context>_<action>. These recommendations should help readability of an IP supplier's generator chain or an EDA vendor's tool generator chain.

Table 1 Example of IP-XACT Generator Naming Conventions

Owner	Target	Context	Action
SPIRITGEN ARMGEN PSGEN STGEN SNPSGEN CDSGEN MENGEN	SIM SYN FPGA CONFIG	HW SW HS	INIT CHECK CREATE COMPILE BUILD

Below is a description of action name part:

- INIT (setup, possible environment creation, directory structure, required init files, things to do before HW/SW views created)
- CHECK (perform some kind of validation check, parameter ranges, boundary conditions, all required user input done)
- CREATE (produce the configured HW or SW depending on context)
- COMPILE (perform the act of compiling on objects created)
- BUILD (hierarchical usage of INIT, CREATE, COMPILE, ELABORATE, VERIFY)
- ELABORATE (stitch compiled objects together in some way)
- VERIFY (run an environment to test the target, for example a simulator)

The example below defines a new generator chain called **SPIRITGEN_SIM_HS_CHAIN** intended to specify a sequence of named simulation tasks for both HW and SW compilation (HS).

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:generatorChain
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2"
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2/index.xsd">
    <spirit:vendor>spiritconsortium.org</spirit:vendor>
    <spirit:library>DesignFlow</spirit:library>
    <spirit:name>flowBuildChain</spirit:name>
    <spirit:version>1.0</spirit:version>
```

© Copyright The SPIRIT Consortium 2006. All rights reserved.

```
        <spirit:fileGeneratorSelector>
            <spirit:groupSelector>
                <spirit:name>SPIRITGEN_SIM_HS_CHAIN</spirit:name>
            </spirit:groupSelector>
        </spirit:fileGeneratorSelector>
    </spirit:generatorChain>
```

The following diagram shows the calling sequence and names of the generator chain in terms of lower-level groups.

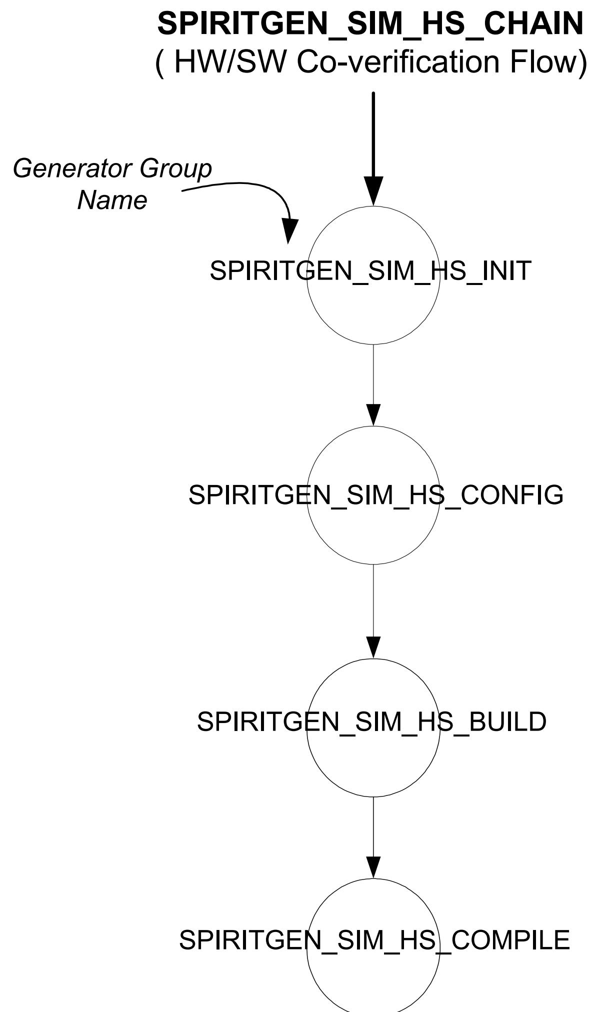


Figure 30 - Example of major Generator group names

The XML file below shows how the above flow would be encapsulated.

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:generatorChain
xmlns:xs=http://www.w3.org/2001/XMLSchema
xmlns:spirit=http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2/index.xsd">
    <spirit:vendor>spiritconsortium.org</spirit:vendor>
    <spirit:library>buildChain</spirit:library>
    <spirit:name>CompleteBuild</spirit:name>
    <spirit:version>1.0</spirit:version>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
        <spirit:fileGeneratorSelector>
            <spirit:groupSelector>
                <spirit:name>SPIRITGEN_SIM_HS_INIT</spirit:name>
            </spirit:groupSelector>
        </spirit:fileGeneratorSelector>
        <spirit:fileGeneratorSelector>
            <spirit:groupSelector>
                <spirit:name>SPIRITGEN_SIM_HS_CONFIG</spirit:name>
            </spirit:groupSelector>
        </spirit:fileGeneratorSelector>
        <spirit:fileGeneratorSelector>
            <spirit:groupSelector>
                <spirit:name>SPIRITGEN_SIM_HS_BUILD</spirit:name>
            </spirit:groupSelector>
        </spirit:fileGeneratorSelector>
        <spirit:fileGeneratorSelector>
            <spirit:groupSelector>
                <spirit:name>SPIRITGEN_SIM_HS_COMPILE</spirit:name>
            </spirit:groupSelector>
        </spirit:fileGeneratorSelector>
        <spirit:chainGroup>SPIRITGEN_SIM_HS_CHAIN</spirit:chainGroup>
    </spirit:generatorChain>
```

5.4.2 Phase Numbers

Phase numbers are intended to define the sequence in which generators should be fired.

A phase number is defined as a non-negative floating-point number that is used to sequence when a generator is run. By building up a series of generators and phase numbers specific sequences of named task invocations can be built up to influence when a design environment should fire a specific generator. Generators can be attached to high-level chains, specific components or specific buses.

There may be multiple generators with the same phase number. In this case, the order should not matter with respect to other generators at the same phase. If no phase number is given then the design environment has license to decide on its position.

Generators can be attached to both components and buses by using the same generator group name. In this case the sequence in which each generator will be invoked depends on the associated phase number. It is up to the design environment to process the generator chains, groups and phase numbers to construct the sequence of generator invocations.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

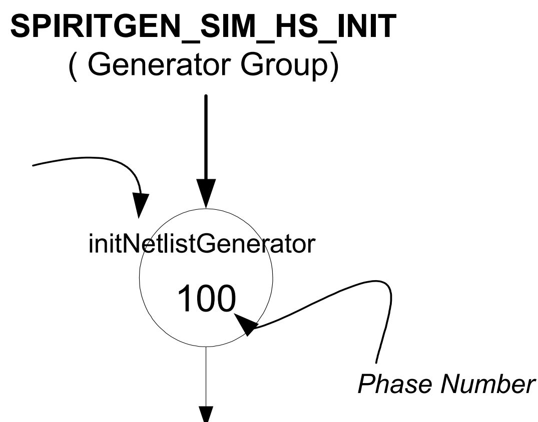


Figure 31 - Loose Generator Example with Phase Number

The XML below shows how a call to such a generator might be defined. Note the definition of the component and bus generator usage of the same group name **SPIRITGEN_SIM_HS_INIT**. This means that if generators are associated with either components or buses using the same generator group name then the DE should invoke them in a sequence defined by the phase numbers.

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:generatorChain
xmlns:spirit=http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.2/index.xsd">
   <spirit:vendor>spiritconsortium.org</spirit:vendor>
   <spirit:library>buildChain</spirit:library>
   <spirit:name>commonInit</spirit:name>
   <spirit:version>r1.0</spirit:version>
   <spirit:generator>
      <spirit:name>initNetlistGenerator</spirit:name>
      <spirit:phase>100</spirit:phase>
      <spirit:accessType>
         <spirit:readOnly>true</spirit:readOnly>
         <spirit:hierarchical>true</spirit:hierarchical>
         <spirit:instanceRequired>true</spirit:instanceRequired>
      </spirit:accessType>
<spirit:looseGeneratorExe>/user/spirit/generators/setupNetlist
</spirit:looseGeneratorExe>
   </spirit:generator>
   <spirit:componentGeneratorSelector>
      <spirit:groupSelector>
         <spirit:name>SPIRITGEN_SIM_HS_INIT</spirit:name>
      </spirit:groupSelector>
   </spirit:componentGeneratorSelector>
   <spirit:busGeneratorSelector>
      <spirit:groupSelector>
         <spirit:name>SPIRITGEN_SIM_HS_INIT</spirit:name>
      </spirit:groupSelector>
   </spirit:busGeneratorSelector>
   <spirit:chainGroup>SPIRITGEN_SIM_HS_INIT</spirit:chainGroup>
</spirit:generatorChain>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

6 IP-XACT SEMANTIC RULES

For an IP-XACT document, or set of IP-XACT documents, to be valid they must, in addition to conforming to the IP-XACT schema, obey certain semantic rules. While many of these are described informally in other sections of this document this chapter defines them formally. Tools generating IP-XACT documents must ensure that these rules are obeyed. Tools reading IP-XACT documents should report any breaches of these rules to the user.

Most of the semantic rules listed below can be checked purely by examining a set of IP-XACT documents. A few, listed at the end of this chapter, need some external knowledge, so cannot be checked this way. In the tables that follow “single document check” indicates that a rule can be checked purely by examining a single IP-XACT document. Rules for which “single document check” is “No” require the examination of the relationships between IP-XACT documents.

6.1 Cross References and VLNVs

Rule Number	Rule	Single Document Check	Notes
1.	Every IP-XACT document shall have a unique VLNV.	No	
2.	Any VLNV in an IP-XACT document used to reference another IP-XACT document shall precisely match the identifying VLNV of an existing IP-XACT document	No	In the schema such references always use the attribute group `versionedIdentifier`
3.	The VLNV in an `extends` element in a bus definition shall be a reference to a bus definition	No	
4.	The VLNV in a `busType` element in a bus interface shall be a reference to a bus definition	No	
5.	The VLNV in a `designRef` element in a design configuration must be a reference to a design	No	
6.	The VLNV in a `pmdRef` element in a design configuration or loose generator changes shall be a reference to a pmd	No	
7.	The VLNV in a `generatorChainRef` element in a design configuration or loose generator changes shall be a reference to a generator chain	No	
8.	The VLNV in an `owner` sub-element of a `fileSet` in a component shall be a reference to a component	No	

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
9.	The VLNV in a `fileName` sub-element of `fileGeneratorSelector` in a generator chain shall be a reference to a generator chain	No	
10.	The VLNV in a `busDefinitionFile` element in a loose generator invocation shall be a reference to a bus definition. In addition the URI in this element shall reference the same bus definition	No	
11.	The VLNV in a `componentRef` element in a pmd or design shall be a reference to a component	No	

6.2 Interconnections

Definitions:

Compatibility of busDefinitions:

- A busDefinition A is an **extension** of busDefinition B if A contains an extension element that references either B or an extension of B.
- A busDefinition is **compatible** with itself.
- If A is an extension of B then A and B are **compatible**
- No other pairs of busDefinitions are **compatible**.
- A set of busDefinitions { A, B, C, ...} is **compatible** if every possible pair of busDefinitions from the set ({ A, B }, { A, C }, { B, C } ...) is compatible

Direction of a bus interface:

- Whether the bus interface is a master, slave, system, mirroredMaster, mirroredSlave, mirroredSystem, or monitor interface.

Rule Number	Rule	Single Document Check	Notes
12.	In the attributes of an `activeInterface` or `monitorInterface` element the value of the `busRef` attribute shall be the name of a `busInterface` in the component description referenced by the VLNV of the component instance named in `componentRef` attribute.	No	

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
13.	In the sub-elements of an `interconnection` element the bus interfaces referenced by the two `activeInterface` subelements shall be compatible. I.e. the VLNVs of the `busType` elements within the two `busInterface` elements shall reference compatible busDefinitions.	No	
14.	A particular component/bus interface combination may appear in, at most, one `interconnection` element in a design	Yes	
15.	An `interconnection` element may only connect a master interface to a slave interface or a mirrored master interface	No	
16.	An `interconnection` element may only connect a mirrored master interface to a master interface	No	
17.	An `interconnection` element may only connect a slave interface to a master interface or a mirrored slave interface	No	
18.	An `interconnection` element may only connect a mirrored slave interface to a slave interface	No	
19.	An `interconnection` element may only connect a direct system interface to a mirrored system interfaces	No	
20.	An `interconnection` element may only connect a mirrored system interface to a direct system interface	No	
21.	In a direct master to slave connection the value of `bitsInLAU` in the master's address space shall match the value of `bitsInLAU` in the slave's memory map	No	
22.	In a direct master to slave connection the range of the master's address space shall be greater or equal to the range of the slave's memory map	No	If the slave's memory map is defined in terms of memory banks or subspace maps then calculating its range may be complex
23.	In a direct master to slave connection the busDefinitions referenced by the `busInterfaces` shall have a `directConnection` element with value "true" (the default value).	No	

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
24.	In a connection between a system interface and a mirrored system interface the values of the `group` elements of the two bus interfaces shall be identical	No	

6.3 Channels and bridges

Rule Number	Rule	Single Document Check	Notes
25.	Within a `channel` element all the `busInterfaceRef` elements shall refer to compatible bus types. I.e. the VLNVs of the `busType` elements within the `busInterface` elements shall reference compatible busDefinitions.	No	
26.	All bus interfaces referenced by a channel shall be mirrored interfaces	Yes	
27.	A channel can be connected to no more mirrored master busInterfaces than the least value of `maxMasters` in the channel and in the busDefinitions referenced by the connected busInterfaces (whether these interfaces are mirrored master or mirrored slave interfaces)	No	
28.	A channel can be connected to no more mirrored slave bus interfaces than the least value of `maxSlaves` in the channel and in the bus definitions referenced by the connected bus interfaces (whether these interfaces are mirrored master or mirrored slave interfaces)	No	
29.	Each bus interface on a component may connect to at most one channel of that channel component	Yes	
30.	The interface referenced by `masterRef` sub-element of a `bridge` element shall be a master.	Yes	

6.4 Monitor interfaces and interconnections

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
31.	An `interconnection` element is not allowed to reference a monitor interface	No	
32.	The `activeInterface` sub-element of a `monitorInterconnection` element shall reference a master, slave, system, mirroredMaster, mirroredSlave, or mirroredSystem interface.	No	
33.	The `monitorInterface` sub-elements of a `monitorInterconnection` element shall reference a monitor bus interface	No	
34.	In a `monitorInterconnection` element the value of the `interfaceType` attributes of the monitor interfaces shall match the direction of the active interface	No	This means all the active interfaces must have the same direction.
35.	A monitor interface may only be connected to a system or mirroredSystem interface if it has a group sub-element, and the value of this element matches the value of the group sub-element of the system or mirroredSystem interface	No	
36.	A particular component/busInterfaceName combination may only appear in one `monitorInterconnection` element	No	This applies to both monitor and active interfaces; however a single `monitorInterconnection` element can connect an active interface to many monitor interfaces. The same active interface can also appear in at most one `interconnection` element

6.5 Configurable elements

Rule Number	Rule	Single Document Check	Notes

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
37.	A configurable element shall have a `dependency` attribute if and only if it has a `resolve` attribute with the value "`dependent`"	Yes	
38.	The value of a `dependency` attribute shall be an XPATH expression. This XPATH expression may only reference the containing document	Yes	
39.	The XPATH expression in a `dependency` attribute may not reference configurable elements with a `resolve` attribute value of either "`dependent`" or "`generated`"	Yes	
40.	Any parameters used within dependent parameter's XPATH id() calls shall exist.	Yes	
41.	All references to elements in dependency XPATH expressions shall be by id. Dependency XPATH expressions shall not use either absolute or relative document navigation to reference other elements	Yes	The purpose of this rule is to allow XPATH expressions to remain valid through schema or design changes. DEs reading IP-XACT documents should, if possible, treat breaches of this rule as minor errors, and should attempt to interpret any XPATH expressions in the document.
42.	An *id* attribute is required in any element with a *resolve* attribute value of *"user"*,	Yes	
43.	`ConfigurableElement` elements may only reference configurable elements that exist in the component referenced by the enclosing `componentInstance` object; specifically the value of the *referenceId* attribute of the `configurableElement` element shall match the value of the *id* attribute of some configurable element of the component.	No	Uniqueness of *id* values within a component is guaranteed by the schema.
44.	`configurableElement` elements may only reference configurable elements with a `resolve` attribute value of "`user`" or "`generated`"	No	

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
45.	If a `configurablElement` element references an element with a formatType attribute value of *"float"* or *"long"* and a *minimum* attribute, then the value of the `configurableElementValue` element shall be greater or equal to the specified value of the `minimum` attribute.	No	
46.	If a `configurablElement` element references an element with a formatType attribute value of *"*`float`*"* or *"*`long`*"* and a *maximum* attribute, then the value of the `configurableElementValue` sub-element shall be less than or equal to the specified value of the `maximum` attribute	No	
47.	If an element has a `formatType` attribute with a value of "`choice`" then it shall also have a `choiceRef` attribute	Yes	
48.	If a `configurableElement` element references an element with a *choiceRef* attribute then the value for `configurableElementValue` sub-element shall be one of values listed in the `choice` element referenced by the `choiceRef` attribute.	No	

6.6 Signals

Rule Number	Rule	Single Document Check	Notes
49.	The value of any `busSignalName` sub-element in a `busInterface` element shall match the value of a `logicalName` element of the bus definition referenced by the `busInterface` element.	No	

6.7 Registers

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
50.	Register offsets in `addressOffset` elements shall not cause registers to overlap.	Yes	
51.	Bit offsets in register fields shall not cause register fields to overlap.	Yes	
52.	The total size of register list within a memory range shall not exceed the size of the range.	Yes	
53.	The total size of bit fields within a register shall not exceed the size of the register	Yes	

6.8 Memory maps

Rule Number	Rule	Single Document Check	Notes
54.	The width of an address block either directly or indirectly (via banks) included in a memory map shall be a multiple of the memory map's bitsInLau	Yes	
55.	Neither a parallel bank, nor banks within a parallel bank, may contain subspace maps.	Yes	
56.	If a parallel bank contains a serial bank, then the widths of all address blocks and sub-banks of that serial bank shall have identical widths	Yes	I.e. The serial bank has a fixed, well-defined width. This is required for sensible addressing of the locations in a parallel bank

6.9 Addressing

See also section 6.13 for a description of how addresses are interpreted in IP-XACT.

Definitions:

Address signal:

- A signal, in a bus definition, is an **address signal** if it has an isAddress sub-element.

Addressable bus interface:

- A bus interface shall be **addressable** if any of its connected signals are address signals.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Hierarchical bus interface:

- A **hierarchical bus interface** is a bus interface of a component that is listed as a hierarchical connection in a design of that component.

Note: a bus interface is hierarchical if **any view** of the component contains a design that names it as a hierarchical connection.

Rule Number	Rule	Single Document Check	Notes
57.	A non-hierarchical addressable master bus interface shall have an `addressSpaceRef` sub-element	No	Since there are potentially useful applications of IP-XACT that do not require addressing information, failure to obey this rule should be treated as a warning rather than an error
58.	A non-hierarchical addressable slave bus interface shall have either a `memoryMapRef` sub-element, or one or more `bridge` sub-elements referencing addressable master bus interfaces.	No	Since there are potentially useful applications of IP-XACT that do not require addressing information, failure to obey this rule should be treated as a warning rather than an error

6.10 Hierarchy

Definitions:

Hierarchical child bus interface:

- Bus interface *IC* of component *CC* is a hierarchical child of bus interface *IP* of component *CP* if and only if *CP* contains a hierarchical view the design file of which contains a hierarchical connection with interface name *IP*, component ref *CC* and interface ref *IC*.

Note: A hierarchical child bus interface may or may not be a hierarchical bus interface itself.

Hierarchical descendant bus interface:

- Bus interface *DC* is a **hierarchical descendant** of bus interface *AC* if and only if *DC* is either a hierarchical child of *AC* or *DC* is a hierarchical child of a hierarchical descendant of *DC*.

Hierarchical family of bus interfaces:

- A **hierarchical family** of bus interfaces a set of bus interfaces consisting of a hierarchical bus interface together with all its hierarchical descendants.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Hierarchical component:

- A component is **hierarchical** if any of its views reference an IP-XACT design.

Hierarchical child component:

- A **hierarchical child** of a component *C* is any component referenced in a design of *C.*

Hierarchical descendent component:

- A **hierarchical descendent** of a component is any hierarchical child of that component, or any hierarchical child of any hierarchical descendent of the component.

Hierarchical family of components:

- A component together with all its hierarchical descendents.

Behavioural properties of a memory location**:**

- The **behavioral properties** of a bit in memory are defined to be:
  - Its access rights
  - Its volatility
  - Whether it has a defined reset value, and if so this value.
  - The width of the memory area containing it:
    - For bits not within parallel banks this is the width of the containing address block.
    - For bits within parallel banks this is the width of the top level parallel bank containing it.
  - The effective least addressable unit (i.e. value of bitsInLau) of its containing memory map
    - Note that bridges between the memory location and the bus interface from which it is observed may modify a location's effective least addressable unit from that which is defined in the memory map.
  - The endianess of its containing address block.
  - The usage of its containing address block
  - Its dependencies:
    - Two bits have the same dependencies if they depend on the same values of the same bits at the same address. Since different memory maps may vary in how they name registers and fields (and even in how they split the address spaces into registers and fields) it is possible for two dependencies to match even if they use different register and field names.

Rule Number	Rule	Single Document Check	Notes
59.	All members of a hierarchical family of bus interfaces shall reference the same busDefinition in their `busType` sub-elements	No	

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
60.	All members of a hierarchical family of bus interfaces shall have the same direction (master, slave, system etc.)	No	
61.	If any member of a hierarchical family of bus interfaces has a connection sub-element with a value other than "explicit" then they all shall have connection sub-elements with identical values	No	The value "explicit" is the default
62.	If any member of a hierarchical family of bus interfaces has an index sub-element then all members shall have identical index sub-elements	No	
63.	If any member of a hierarchical family of bus interfaces has a bitSteering sub-element then all members shall have identical bitSteering sub-elements	No	
64.	If any member of a hierarchical family of bus interfaces has signalMap sub-element then they all shall.	No	
65.	All the signalMaps of a hierarchical family of bus interfaces shall reference the same set of bus signals. I.e. if one contains a signal with busSignalName element with value *s* then they all shall all contain a signal with busSignalName element with value *s*	No	An effect of this, together with 57 and 58, is that, if a hierarchical bus interface is addressable then its non-hierarchical descendents (i.e. the leaves of the tree) must be, and hence shall contain addressing information
66.	In a hierarchical family of bus interfaces all signals in the signalMaps referencing the same bus signal shall have the same left and right values.	No	
67.	In a hierarchical family of bus interfaces the componentSignalName of all signals in the signalMap referencing the same bus signal shall reference signals with the same direction	No	

© Copyright The SPIRIT Consortium 2006. All rights reserved.

Rule Number	Rule	Single Document Check	Notes
68.	In a hierarchical family of bus interfaces the component signals referenced by the componentSignalName of all signals in the signal maps referencing the same bus signal shall, if they have default values, have identical default values.	No	I.e. it is legal for only some of the descriptions of a signal to have default values, but those that have must have identical default values.
69.	In a hierarchical family of bus interfaces the componentSignalName of all signals in the signalMap referencing the same bus signal shall reference signals with identical clockDriver sub-elements	No	
70.	In a hierarchical family of bus interfaces the componentSignalName of all signals in the signalMap referencing the same bus signal shall reference signals with identical singleShotDriver sub-elements	No	
71.	In a hierarchical family of bus interfaces the componentSignalName of all signals in the signalMap referencing the same bus signal shall reference signals with identical signalConstraintSets sub-elements	No	

6.11 Hierarchy and Memory maps

Rule Number	Rule	Single Document Check	Notes
72.	In a hierarchical family of slave or mirrored master bus interface all bus interfaces that define addressing information (either directly or via bridges and channels) shall define the same set of addresses to be visible	No	I.e. If one member of the family defines an address as a valid address accessible through that bus interface then all members of the family that define addressing information must define that same address as a valid address accessible through that bus interface.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Rule Number	Rule	Single Document Check	Notes
73.	If, in any member of a hierarchical family of slave or mirrored master bus interfaces, an address resolves to reference a location outside the containing hierarchical family of components then that address shall reference the same location (i.e. the same address on the same bus) in every member of the hierarchical family that defines addressing information.	No	I.e. if *C* is a hierarchical component, and either the IP-XACT description of *C* itself, or some design of C, tells us that accessing address *a* of *C* on bus interface *I* results in an access to address *b* of some other bus interface *J* of *C*; then all designs of *C* that tell us about addressing on *I* must tell us the same about this address.
74.	If any bit address (i.e. address plus bit offset) is resolved to a bit within an address block by any member of a hierarchical family of slave bus interfaces then all members of that family with addressing information shall resolve that bit address to a bit with identical behavioural properties.	No	If an address resolves to a location within the hierarchical family of components then, normally, its only observable features from outside the hierarchical family are its behavioural properties (but see the rule 75)
75.	If, in the addressing information of any member of a hierarchical family of bus interfaces, any two addresses resolve to the same location, then this shall be true for all members of the hierarchical family of bus interfaces that have addressing information	No	i.e. aliasing of addresses shall be preserved. Note that aliasing is observable from outside the hierarchical family.

6.12 Rules requiring external knowledge

Rule Number	Rule	Single Document Check	Notes

 © Copyright The SPIRIT Consortium 2006. All rights reserved.

Rule Number	Rule	Single Document Check	Notes
76.	The `name` sub-element of a `file` element can contain environment variables in the form of ${ENV_VAR} which are meaningful to the host operating system and which, when expanded, should result in a string which is a valid URI	Yes	
77.	In VLNVs the vendor name should be specified as the top level internet domain name for that organization. The domain should be ordered with the top level domain name at the end (as in HTTP URLs) Example Mentor: mentor.com; ARM: arm.com etc.	Yes	This is to guarantee uniqueness of vendor names.
78.	The envIdentifier of a view shall be a text string consisting of three fields delimited by colons. The first two fields shall be a language name, which shall be one of the languages available for fileTypes, and a tool name. The tool name may be either generic (e.g. "*Simulation", "*Synthesis") or a specific tool name such as "DesignCompiler", "VCS", "NCSim", "ModelSim". The third field will an be arbitrary vendor specific text string	Yes	Tool vendors shall publish a list of valid tool names. The SPIRIT Consortium website includes a list of these names registered by members of The SPIRIT consortium.

6.13 PMD Files

Rule Number	Rule	Single Document Check	Notes
79.	The `styleSheet` element must reference a valid XSLT file	No	
80.	The execution of the `styleSheet` should produce a valid IP-XACT XML file	No	
81.	The `parameters` referenced in the `styleSheet` shall be defined in the PMD XML file	No	Extra parameters in the PMD file is OK

6.14 Addressing formulas

This section describes how the address of a location in an IP-XACT component is transformed by connections between components in IP-XACT. It also describes which bits of an address space are visible across such connections.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

***Note** The material in this section is preliminary and awaiting further review.*

6.14.1 Overview

IP-XACT component descriptions include addressing information for each addressable bus interface of a component. If this addressing information in IP-XACT is to be useful it is essential that it should be unambiguous. In particular, IP-XACT must precisely define what location is accessed when a component attempts an access to an address on a master bus interface.

Within an IP-XACT component each addressable slave bus interface has associated with it either a memory map (which may be shared with other bus interfaces) or a set of transparent bridges to master bus interfaces. A memory map may contain both address blocks, which describe the local memory and device registers of the component, and subspace maps, which describe how addresses are transformed by the component's opaque bridges.

In IP-XACT every memory map has a least addressable unit (LAU). This describes how that memory map is addressed. For example, if the memory map is byte addressed the LAU is 8 bits. Within a memory map any bit may be uniquely addressed by an address which is a multiple of the LAU plus a bit offset less than the LAU.

A number of factors control whether a location in a memory map is visible at a master port within a design, and if so at what address. These are:

1 Whether there is a path (via interconnections, channels and bridges) from the master bus interface to a slave bus interface using that memory map. For the remainder of this document it is assumed that such a path exists.
2 The address and bit offset of the location in the memory map.
3 The base addresses and bit offsets defined in:
   3.1 The master bus interface
   3.2 The mirrored slave interfaces of channels
   3.3 The master bus interfaces of intervening bridges
   3.4 The subspace maps of intervening opaque bridges.
4 The widths of:
   4.1 The bus interfaces
   4.2 The master bus interface's address space
   4.3 The address block containing the location
5 Whether any bus interface supports bit steering, and if so which.
6 The range of:
   6.1 The master bus interface's address space.
   6.2 The address space of the master bus interfaces of any intervening bridges.
   6.3 Mirrored slave interfaces.
7 The endianess of:
   7.1 The address block containing the location
   7.2 The master bus interface's address space
   7.3 The address space of the master bus interface of any intervening bridges.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

***Note**: The current version of this section does not discuss endianess. This will be added in a later version.*

The following sections describe in detail the rules for calculating the visibility and addresses of locations. In these sections I talk entirely in terms of bit addresses at various points in a design. This is based on every point in a design having a well-defined local `bitsInLau`. The bit address of a bit in memory is `address*localBitsInLau+bitOffset`.

6.14.1.1 Scope of addressing formulas in IP-XACT 1.2

IP-XACT 1.2 only defines addressing if:

1. All address blocks and address spaces relevant to calculating the transformations of a particular address have the same endianness.
2. There is only one (multi-bit) data signal on any interconnection. The addressing formulas in this document would work equally well with one input and one output signal, and could be applied to that case. If, however, the input and output data signals are different widths, and bit steering is not enabled throughout, then this could result in different addressing for reads and writes. IP-XACT does not, at present, contain sufficient information to fully define the addressing on bus interfaces with multiple input or output signals.

6.14.1.2 Simplifying assumptions

To simplify the following description it is assumed that all multi-bit signals are described with `left ≤ right`. If a signal has `right < left` then `left` and `right` can be swapped.

6.14.2 Breaking down the path

The path an address block to a master component (which may be a processor or some other device that generates bus accesses) accessing that address block may be broken down into a number of steps. If there are no bridges between the master component and the address block are:

1. Connection from the address block to "just outside" the slave bus interface
2. Connection from "just outside" the slave bus interface to "just outside" the mirrored slave bus interface.
3. Connection through a channel from "just outside" the mirrored slave bus interface to "just outside" the mirrored master bus interface
4. Connection from "just outside" the mirrored master bus interface to "just outside" the master bus interface.
5. Connection from "just outside" the master bus interface to master component's address space.

If there is no channel then steps 2, 3, and 4, are replaced by:

6. Connection from "just outside" the slave bus interface to "just outside" the master bus interface

Bridges replace step 5 with:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

7. Connection across a bridge from "just outside" the master bus to “just outside” the slave bus interface.

The whole sequence is then repeated from step 2:

The figure below illustrates all these steps

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

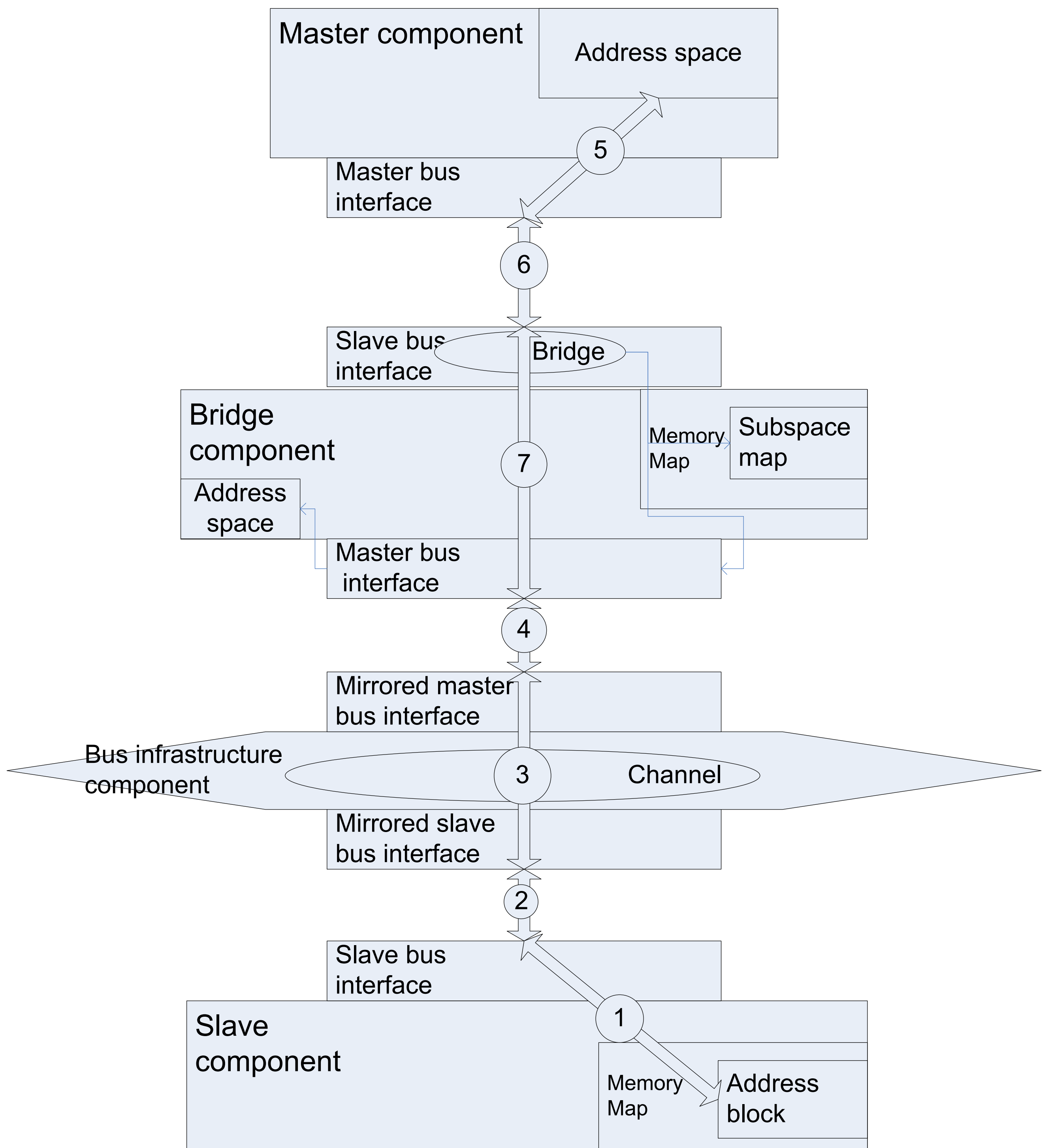


Figure 32- Connection Steps

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

6.14.3 Connection from "just outside" bus interface A to "just outside" bus interface B.

In the steps that connect two bus interfaces (i.e. steps 2, 4, and 6), whether slave to master, slave to mirrored-slave or mirrored-master to master, addresses, and bit visibility, are modified in a consistent way:

1. If the data signal (i.e. signal with `isData` set) is connected, with the same widths, at the two ends then addresses and bit visibility are unchanged.
   1.1. This is true whatever left and right values are used in the signal map in bus interface definition.
2. If the data signal has different widths at the two ends, then it is defined to be the highest numbered bits within the signal that are lost from the data at the narrow end of the connection (or added, with default signal bit values if defined)
   2.1. This is true whatever left and right values are used in the signal map in bus interface definition.
   2.2. The effect of this is that, for bits that are transmitted between the two ends:
      2.2.1. The bit number relative within the signal is unchanged
      2.2.2. The bus word address (i.e. bit_address ÷ width) is unchanged.
   2.3. Mathematically a bit is only transmitted between the two ends if:
      2.3.1. `bit_address(wide_end) mod width(wide_end) < width(narrow_end)`
   2.4. For bits that are transmitted between the two ends the addressing formula is:
      2.4.1. `bit_address(narrow_end) = (bit_address(wide_end) ÷ width(wide_end)) * bit_address(narrow_end) + (bit_address(wide_end) mod width(wide_end))`

This is illustrated in the following diagram:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Figure 33 - Connection across Two Bus Interfaces

6.14.4 Connection from an address block to “just outside” the associated slave bus interface

This step (step 1) converts the bit address from a relative bit address within the address block to a bus address. How this is done depends on whether the bus interface has `bitSteering` set.

1. With `bitSteering`
   1.1. All bits in the address block are visible “just outside” the associated slave bus interface, as consecutive bits (unless the address block is within a parallel bank).
   1.2. The bit address on the bus is:
      1.2.1. `relative_bit_address_in_address_block`
      `address_block.baseAddress*memory_map.bitsInLau +`
      `address_block.bitOffset`
2. Without `bitSteering`:
   2.1. The address block is seen as a series of rows each `address_block.width` wide; the bits within each row are numbered from 0 to `address_block.width - 1.`

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

2.2. If the data signal has a range from `left` to `right` in its component signal description then it connects to bit columns `0` to `right-left+1` inclusive of the address block.

2.3. The `baseAddress` and `bitOffset` of the address block are translated into a row number, and bit number within the row, of the first bit in the address block.

2.4. This can be expressed by the following formulas:

2.4.1. `bit_address_in_memory_map = relative_bit_address_in_address_block + address_block.baseAddress*memory_map.bitsInLau + address_block.bitOffset`

2.4.2. `row_in_memory_map = bit_address_in_memory_map ÷ address_block.width`

2.4.3. `bit_pos_in_row = bit_address_in_memory_map mod address_block.width`

2.4.4. A bit is visible on the bus if `bit_pos_in_row ≤ right – left + 1`

2.4.5. On the bus the bit address is: `row_in_memory_map*(right-left+1) + bit_pos_in_row`

© Copyright The SPIRIT Consortium 2006. All rights reserved.

The following diagram illustrates this:

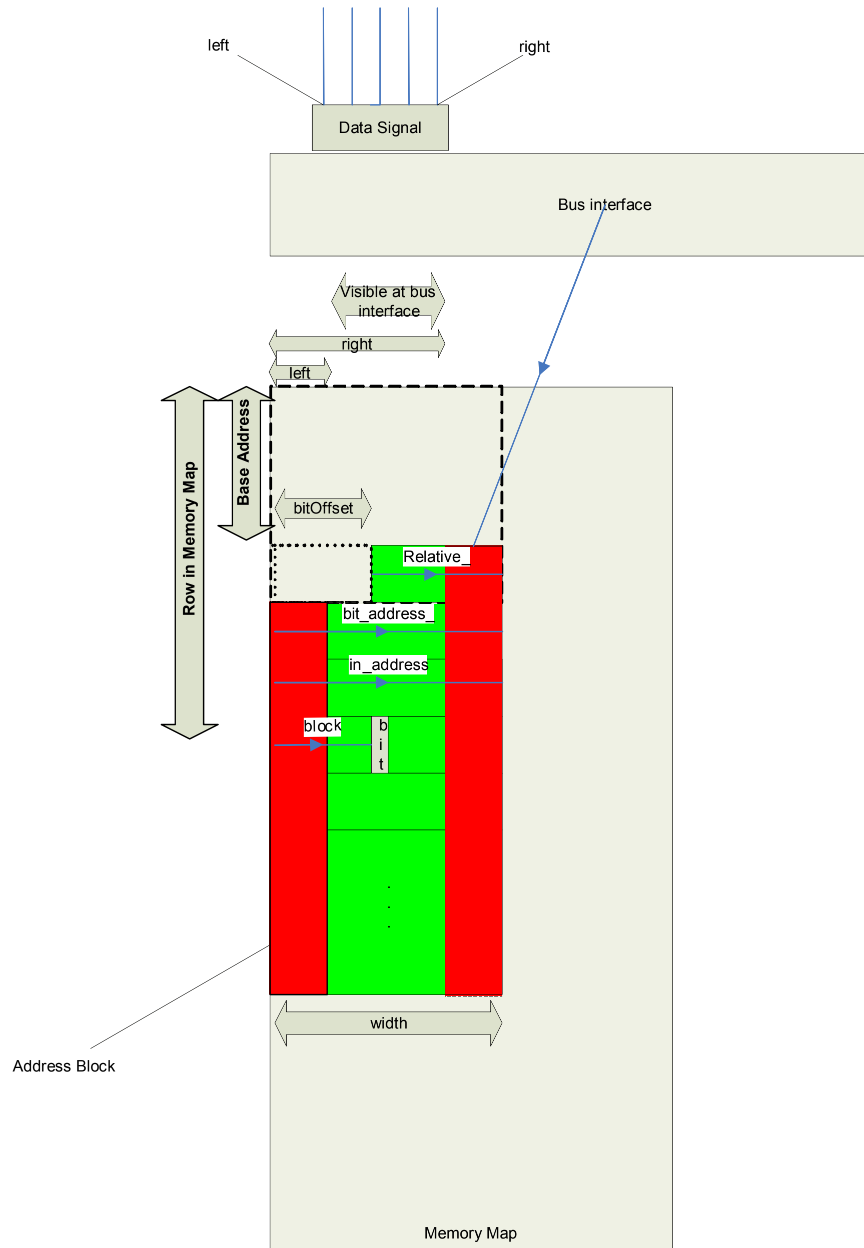


Figure 34 - Bus Interface to Address Block Connection

Note: A consequence of this definition is that the conversion of the base address into a row is dependent on the width of the address block. This can give some non-intuitive results. For example with a byte addressable memory map (i.e. bitsInLau =

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

8) an address block that starts at 0, is 16 bits wide, and has a range of 0x2000 does not overlap with an address block that starts at 0x1000, and is 8 bits wide. This is because the first block contains 0x2000 * 8/16 = 0x1000 rows, and the second block starts at row 0x1000*8/8 = 0x1000.

3. `Parallel banks`

3.1. If the address block is within a parallel bank, then, whether or not bit steering is enabled, `relative_bit_address_in_address_block` should be replaced by `relative_bit_address_in_bank` in the above calculations. Also, `address_block.width` should be replaced by `bank_width`. If the address block containing the location is the ith element of n then this can be calculated as follows:

3.1.1. $$\mathtt{bank_width} = \sum_{j=0}^{n-1} \mathtt{width(element(j))}$$

3.1.2. $$\mathtt{block_bit_offset} = \sum_{j=0}^{i-1} \mathtt{width(element(j))}$$

3.1.3. $$\mathtt{bit_row} = \mathtt{relative_bit_address_in_address_block} \div \mathtt{width(address_block)}$$

3.1.4. $$\mathtt{bit_offset} = \mathtt{relative_bit_address_in_address_block}\ \mathtt{mod}\ \mathtt{address_block.width}$$

3.1.5. $$\mathtt{relative_bit_address_in_bank} = \mathtt{bit_row} * \mathtt{bank_width} + \mathtt{bit_offset} + \mathtt{block_bit_offset}$$

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

The following diagram illustrates this:

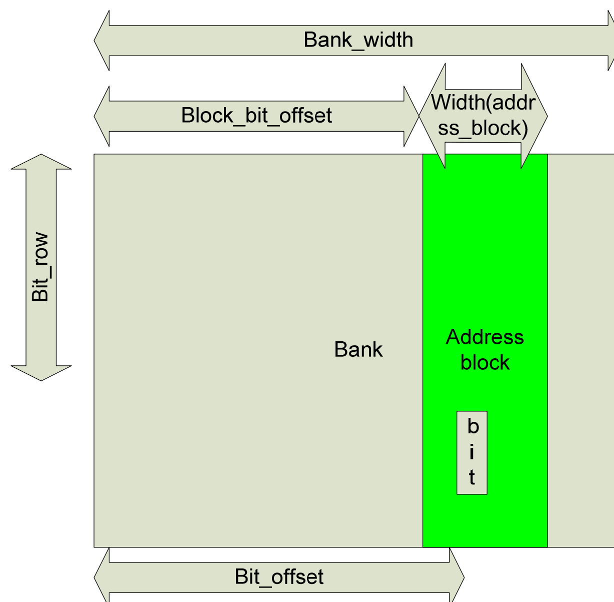


Figure 35 - Bank Address Example

6.14.5 Connection through a channel from "just outside" the mirrored slave bus interface to "just outside" the mirrored master bus interface

This step (step 3) converts between the addressing on the two sides of a channel. How this is done depends on whether bit steering is enabled on either of the mirrored bus interfaces. With bit steering set on either or both bus interfaces the visibility of bits is only modified by the range of any remap element in the mirrored slave, and the addressing of bits is only modified by any remap base address in the mirrored slave. If bit steering is not set on either interface then, in addition, the component signal `left` and `right` values are used to describe the range of bit lanes on the bus that connect to each component. For example, if a component has an 8 bit wide bus interface, and is connected to a 32 bit wide bus, then `left = 0` and `right = 7` means that it is connected to the 1st byte lane, whereas `left = 8` and `right = 15` indicates that it is connected to the 2nd byte lane.

Details:

1. With `bitSteering:`
   1.1. `bit_address(mirrored_master) = bit_address(mirrored_slave) + mirrored_slave.baseAddress.remapAddress * slave.memory_map.bitsInLau`
      1.1.1. Note that the remap address is measured in units of the slave's least addressable unit.

© Copyright The SPIRIT Consortium 2006. All rights reserved.

1.1.2. If there is no remap address for the current remap state then it defaults to 0.

1.2. A bit is visible at the mirrored master if `bit_address(mirrored_slave) ≤ mirrored_slave.base_address.range * slave.memory_map.bitsInLau`

1.2.1. If there is no base address the range is treated as infinite.

2. Without `bitSteering`:

2.1. `rebased_bit_address = bit_address(mirrored_slave) + mirrored_slave.baseAddress.remapAddress * slave.memory_map.bitsInLau`

2.2. `mslave_row = rebased_bit_address ÷ ( right(mirrored_slave) - left(mirrored_slave) + 1 )`

2.3. `mslave_bit_offset = (rebased_bit_address mod ( right(mirrored_slave) - left(mirrored_slave) + 1 )) + left(mirrored_slave)`

2.4. A bit is visible just outside the mirrored master if:

2.4.1. `bit_address(mirrored_slave) ≤ mirrored_slave.base_address.range * slave.memory_map.bitsInLau`

2.4.2. and `left(mirrored_master) ≤ mslave_bit_offset ≤ right(mirrored_master)`

2.5. If a bit is visible its bit address just outside the mirrored master is: `bit_address(mirrored_master) = mslave_row * ( right(mirrored_master) – left(mirrored_master) + 1) + mslave_bit_offset – left(mirrored_master)`

6.14.6 Connection from "just outside" the master bus interface to master component's address space

This step (step 5) converts from the master component’s addressing to the bus addressing at the master bus interface. If the bus interface supports bit steering then the bit address is simply offset by the bus interface’s base address. If it does not then, in addition the data signal’s component left and right values describe how the bit lanes of the signal map onto the bit lanes of the bit lanes of the master component’s address space.

In detail:

1. With `bitSteering:`

1.1. `bit_address(address_space) = bit_address(outside_master) + master.addressSpaceRef.baseAddress * master.addressSpaceRef->bitsInLau + master.addressSpaceRef.bitOffset`

1.1.1. Both the base address and the bit offset default to 0.

1.2. A bit is visible at the mirrored master if `bit_address(outside_master) ≤ master.addressSpaceRef->range * master.addressSpaceRef->bitsInLau`

1.2.1. If there is no base address the range is treated as infinite.

2. Without `bitSteering`:

2.1. `master_row = bit_address(outside_master) ÷ ( right(master) - left(master) + 1 )`

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

2.2. `master_bit_offset = (bit_address(outside_master) mod (right(master) - left(master) + 1 )) + left(master)`

2.3. A bit is visible to the master component if:

2.3.1. `bit_address(outside_master) ≤ master.addressSpaceRef->range * master.addressSpaceRef->bitsInLau`

2.3.2. and `master_bit_offset ≤ master.addressSpaceRef -> width.`

2.4. If a bit is visible:
`relative_bit_address = master_row * (master.addressSpaceRef -> width) + mslave_bit_offset`

2.5. `bit_address(address_space) = relative_bit_address + master.addressSpaceRef.baseAddress * master.addressSpaceRef->bitsInLau + master.addressSpaceRef.bitOffset`

The bit address will normally be seen by the master component as an address in units of bitsInLau and a bit offset. These values are:

3. `address(address_space) = bit_address(address_space) ÷ master.addressSpaceRef->bitsInLau`
4. `offset(address_space) = bit_address(address_space) mod master.addressSpaceRef->bitsInLau`

6.14.7 Connection across a bridge from "just outside" the master bus to “just outside” the slave bus interface.

In a connection across a bridge (step 7) the address is modified by the base address of the master and, for opaque bridges, by the base address and bit offset of the subspace map. A bridge always connects the full-connected width of the master bus interface to the full-connected width of the slave bus interface, whether or not the ports support bit steering.

1. An address is visible on just outside the slave bus interface if
`bit_address(bridge_master) < bridge_master.addressSpaceRef->range * bridge_master.addressSpaceRef->bitsInLau`
2. The address at the slave can be calculated as:
```
bit_address(bridge_slave) =
bit_address(bridge_master) +
bridge_master.addressSpaceRef.baseAddress *
bridge_master.addressSpaceRef->bitsInLau +
bridge_master.addressSpaceRef.bitOffset +
subspace_map.baseAddress *
bridge_slave.memoryMapRef->bitsInLau +
subspace_map.bitOffset
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

7 BACKWARD COMPATIBILITY

Some IP-XACT 1.2 changes are not backward compatible with IP-XACT 1.1. These are documented in the IP-XACT 1.2 release notes. As part of the IP-XACT 1.2 deliverables, XSLT transforms are supplied that transforms IP-XACT 1.1 XML to IP-XACT 1.2 XML. Design Environments that are compliant with IP-XACT 1.2 are expected to incorporate these transforms or equivalent to be able to automatically read and process IP-XACT 1.1 XML. This provides forward compatablity, such that deliverables for 1.1 will be able to be used in 1.2 Design Enviroments. Note that the LGI (Loose Generator Interface) is not expected to be supported beyond 1.2.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

8 VERIFICATION SUPPORT IN IP-XACT

IP-XACT v1.2 introduces extensions to IP-XACT v1.1 for verification. These extensions document and support test bench creation for a design. The main features that will be highlighted in this chapter (some of them already presented in previous sections and thus summarized here, in context):

Monitor bus interface type and interconnection of verification IP with monitor bus interfaces,

White box interface, documenting probing or driving points within a component or design, and

Sequence files, data level verification IP.

For verification components (VIP), a new bus interface type called "Monitor" was introduced in IP-XACT v1.2 and presented in Chapter 4.9.1.3. As well, that chapter presented the update to the IP-XACT schema for specification of interconnections in a design involving monitor bus interfaces on a VIP and regular bus interfaces on design IP components. The white box interface (WBI) was introduced in Chapter 4.6.2. The WBI does not reflect a physical interface on an IP component; rather, the WBI specifies internal points in the IP to be probed or driven by verification tools and/or test benches that can do so. The interconnection syntax of whitebox ports at the boundary of the test bench is design environment dependant. Design Environments must export and import the XML in standard form to identify whitebox interface points and connect whitebox interfaces to the testbench environment. The specification of sequence files, providing data for verification IP and tools in test benches, is introduced in this chapter.

8.1 Monitor Bus Interface & Interconnection

IP-XACT v1.2 extends the bus interface definition with the monitor type. A verification IP is defined as a component but, when connected to an IP component in a design, it must not appear as a component that is an active part of that design. When making interconnections between a monitor bus interface on a VIP and any active interface on a component IP, the capacity of the active bus interface (i.e., count of masters or slaves) is not impacted, since the VIP is neither a master nor a slave component.

Some of the following repeats material (with slight modifications) from Chapter 4.9 from more general discussions of bus interfaces and component interconnections.

8.1.1 Monitor interfaces

A **monitor interface** is an interface used in verification that that is neither an active master, slave nor system interface. This allows specialized (or non-standard) connections to a bus that will not count as a connected interface. Monitor interfaces are used to connect verification IP used to monitor an interface of type master, slave or system and do not count as a connected interface in the design environment. A monitor interface is identified by the monitor tag in the interface definition, with an attribute to specify the type of active interface being monitored (master, slave, system).

A monitor interface is declared in the component XML as shown in the example below, repeated from Chapter 4.9.2.5, for monitoring a master bus interface:

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

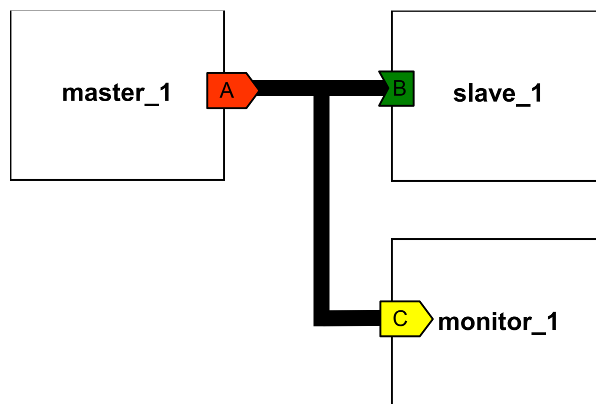


Figure 36 - Interconnection with a Monitor

```
<spirit:component>
  …
  <spirit:name>MyMonitor<spirit:name>
  …
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>C</spirit:name>
      <spirit:monitor spirit:interfaceType="master" />
    </spirit:busInterface>
  </spirit:busInterfaces>
</spirit:component>
```

In the complete component definition for a VIP component, there will be a signalMap (associating logical to physical pins) defined within the bus interface section, following the declaration of the monitor type. All of the physical pin signals on the VIP are inputs for bus interfaces defined as the monitor type.

8.1.2 Monitor Interface connection

With a monitor interface it is possible to have multiple connections between an interface of a design IP and an interface of verification IP. If we consider the figure above, this figure represents the connection between a master interface (A) on Master_1 and a slave interface (B) on Slave_1. The monitor (Monitor_1) is used to check, e.g., the bus protocol. Monitor_1 has a monitor interface (C).

Two interconnections are used to enable this:

- Master_1(A) → Slave_1(B) – a standard interconnection
- Master_1(A) → Monitor_1(C) – a monitor interconnection

A sample of the interconnection section (in design.xml) for the figure above is given below:

```
<spirit:interconnections>
   <spirit:interconnection>
     <spirit:activeInterface
         spirit:componentRef="Master_1"
         spirit:busRef="A" />
    <spirit:activeInterface>
         spirit:componentRef="Slave_1"
         spirit:busRef="B" />
   </spirit:interconnection>
   <spirit:monitorInterconnection>
     <spirit:activeInterface
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
            spirit:component1Ref="Master_1"
            spirit:busInterface1Ref="A" />
        <spirit:monitorInterface
            spirit:componentRef="Monitor_1"
            spirit:busRef="C" />
      </spirit:interconnection>
   </spirit:interconnections>
```

The presence of the monitor, C, does not count against the number of slaves present on the master interface. This is important where capacity limits are specified by a maxSlaves value in the bus definition used for these bus interfaces.

8.2 White Box Interface

Verification IP such as monitors will have pseudo-physical bus interfaces to connect with bus interface signals under test while not being an actual part of the design but as part of a test bench. Other verification tools may require access to component IP in a design, at a level deeper than the interfaces defined for the component. This can be seen in situations where internal registers or flags must be monitored, or internal nodes are driven with particular signals. IP-XACT v1.2 defines a new, non-physical interface type for verification under these requirements.

Internal elements of a component can be accessed in the DE by other components, in particular verification components, through whitebox interfaces. These are listed in the <spirit: whiteboxElements> section of the component. Each whitebox element has a name, a direction, and a string description.

Example: A status flag that is not visible at the interface of a component, but may be needed for debug, or for probing by a testbench component or an assertion.

```
<spirit:whiteboxElements>
      <spirit:whiteboxElement>
        <spirit:name>Name</spirit:name>
        <spirit:whiteboxType>register</spirit:whiteboxType>
        <spirit:driveable>false</spirit:driveable>
        <spirit:description>
           error in most recent transfer
        </spirit:description>
      </spirit:whiteboxElement>
    </spirit:whiteboxElements>
```

Connections to whitebox elements are not specified in the IP-XACT design file. The whiteboxElement section informs the DE what internal elements are accessible, and the DE itself has to make any connections. The details of how the whitebox element is implemented, and the path to it, are contained in the <spirit: view> section of <spirit: model>. The same whitebox element may be implemented differently in different views.

8.3 Describing Verification Sequences

Verification sequences are a means of defining a stream of input data to be driven into a design under test. This information may be represented in a number of diverse ways and may be at many different levels of abstraction. Some of the more common ones are listed below.

- Print on change data stream to be applied to a pin set, such as VCD
- List of READ/WRITE actions to be applied by a bus functional model (BFM) (dependant on BFM design)

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

- C program to be run on an embedded processor
- Constraints on random data items
- Hierarchical tree of constrained random sequences
- ….. many others

The IP-XACT schema is capable representing these sequences such that they can be:

- Identified and listed for user selection
- Associated with the design IP that they may be designed for
- Associated with the Verification IP that is required to ‘play’ them

It is assumed that all sequences either resided within a single file or are split across multiple files. In addition a single file may contain multiple sequences.

8.3.1 Representing the sequence

The Sequence should be represented using the ‘filesets’ schema element under a component definition, and each fileset should be used to represent a logical set of sequences that should be grouped together. Within the fileset the following fields should be used:

- *spirit:file* : This is a list of files in which the sequence is contained; within the file element the following fields are relevant.
- *spirit:logicalName*: May be used to describe a logical name for the set of sequences
- *spirit:exportedName*: Define all of the names of the sequences defined in this file. Where subtyping is used this may be represented in the exported name i.e. “BIG ERROR usb_init_sequence”
- *spirit:swFunction*: If the fileset contains parameterizable sequences then the parameterization is defined using swFunction, each parameterizable sequence contained in the fileset should have its own swFunction definition. Within the swFunction the following elements are relevant
- *spirit:entryPoint*: This is used to represent the name of the sequence to be parameterized, as defined in *spirit:file→ spirit:exportedName*.
- *spirit:argument*: This is a list of the parameters that may be applied to the sequence.

8.3.2 Associating the sequence with Design IP

If a sequence is associated with a specific Design IP, then it should be physically located within the filesets of the design IP’s component definition.

8.3.3 Associating the sequence with Verification IP

If a sequence is a general purpose sequence that is applicable to many design IPs, and it is delivered with the Verification IP, then it may be defined in the fileset associated with the Verification IP.

Sequences that are delivered with the Design IP and are located in the design IP’s component declaration also need to be associated with the Verification IP that is required to play it. This is achieved by using a tight generator associated with the verification IP that will search the design database for all sequences and identify ones that it is capable of playing. It will do this identification by interrogating the *spirit:file→spirit:exportedName* field to see if the defined sequence is of a type it can play.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

9 APPENDIX: USE CASE EXAMPLES

This section intends to give some modeling guidelines based on the IP-XACT platform example (Leon, Amba) to help write a Design Environment and IP-XACT IP models/libraries. When not applicable, some simple examples will be given to illustrate a specific point.

9.1 Packaging of a component

9.1.1 Introduction

In this section we will describe how to package in IP-XACT XML the Leon UART IP from the RTL point of view. It will mainly show the usage of the following elements:

- `busInterfaces`
- `memoryMaps`
- `model`
- `choices`
- `fileSets`
- `componentConstraintSets`
- `otherClockDrivers`

In order to specify:

- The IP signals list
- The bus interfaces list
- The memory map
- The connections between the IP signals and the interfaces signals
- The different views on which the design environment can work on
- The hardware parameters mapping VHDL generics
- The VHDL source files list
- The timing constraints

9.1.2 Describing the bus interfaces

For the UART IP, we mainly want to describe:

1. The different busses connecting the IP
2. Since it is a slave, the reference to the memory map of that slave
3. The connection between the IP signals and the interfaces signals

The Uart is connected to an AMBA APB bus and an external bus getting the serial data. Only the first one has an interface and is listed as busInterface inside the XML:

```
<spirit:busInterfaces>
   <spirit:busInterface>
     <spirit:name>ambaAPB</spirit:name>
     …
   </spirit:busInterface>
</spirit:busInterfaces>
```

For the external bus, its pins are being set as exportable in order to be able to connect them at top level (see below in section 9.1.4 to see how this is done).

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

How the Amba APB bus interface is specified will be examined next. The first thing specified is the bus definition defined with its VLNV. The bus definition for the APB bus is coming from the ARM:

```
<spirit:busType
    spirit:vendor="amba.com"
    spirit:library="AMBA2"
    spirit:name="APB"
    spirit:version="r0p0"/>
```

In the slave section, we specify a reference to the memory map of this slave interface:

```
<spirit:slave>
  <spirit:memoryMapRef spirit:memoryMapRef="ambaAPB" />
</spirit:slave>
```

Finally, we have to specify how the connections are made between UART signals and APB bus signals. On the IP side, we have the following APB signals:

```
psel    : in  Std_ULogic;
penable : in  Std_ULogic;
paddr   : in  Std_Logic_Vector(31 downto 0);
pwrite  : in  Std_Ulogic;
pwdata  : in  Std_Logic_Vector(31 downto 0);
prdata  : out Std_Logic_Vector(31 downto 0);
```

In XML, the bus signal name is specified using the busSignalName element while the component signal name is specified using the componentSignalName element. The connection is then specified as:

```
<spirit:busInterface>
…
  <spirit:signalMap>
     <spirit:signalName>
       <spirit:componentSignalName>psel</spirit:componentSignalName>
       <spirit:busSignalName>PSELx</spirit:busSignalName>
     </spirit:signalName>
     <spirit:signalName>
       <spirit:componentSignalName>penable</spirit:componentSignalName>
       <spirit:busSignalName>PENABLE</spirit:busSignalName>
     </spirit:signalName>
     <spirit:signalName>
       <spirit:componentSignalName>paddr</spirit:componentSignalName>
       <spirit:busSignalName>PADDR</spirit:busSignalName>
     </spirit:signalName>
     <spirit:signalName>
       <spirit:componentSignalName>pwrite</spirit:componentSignalName>
       <spirit:busSignalName>PWRITE</spirit:busSignalName>
     </spirit:signalName>
     <spirit:signalName>
       <spirit:componentSignalName>pwdata</spirit:componentSignalName>
       <spirit:busSignalName>PWDATA</spirit:busSignalName>
     </spirit:signalName>
     <spirit:signalName>
       <spirit:componentSignalName>prdata</spirit:componentSignalName>
       <spirit:busSignalName>PRDATA</spirit:busSignalName>
     </spirit:signalName>
   </spirit:signalMap>
</spirit:busInterface>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

The full description of the `busInterfaces` element can be found in section 9.1.9. All the bus interfaces are described together with a full description of all the registers.

9.1.3 Describing the Memory Map

The available registers and their offset are specified in the following table.

Offset	Register Name
0x0	Data Register
0x4	Status Register
0x8	Control Register
0xC	Scalar Register

Table 2 : Leon Uart register map

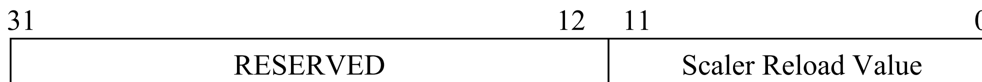
This memory map can be described in xml as:

```
<spirit:memoryMaps>
  <spirit:memoryMap>
    <spirit:name>ambaAPB</spirit:name>
    <spirit:addressBlock>
      …
      <spirit:register>
        <spirit:name>data</spirit:name>
 <spirit:addressOffset>0x0</spirit:addressOffset>
        <spirit:size>32</spirit:size>
        <spirit:access>read-write</spirit:access>
        <spirit:description>
Data read/write register
</spirit:description>
      </spirit:register>
      <spirit:register>
        <spirit:name>status</spirit:name>
 <spirit:addressOffset>0x4</spirit:addressOffset>
        <spirit:size>32</spirit:size>
        <spirit:access>read-only</spirit:access>
        <spirit:description>
Status register
</spirit:description>
      </spirit:register>
      <spirit:register>
        <spirit:name>control</spirit:name>
 <spirit:addressOffset>0x8</spirit:addressOffset>
        <spirit:size>32</spirit:size>
        <spirit:access>read-write</spirit:access>
        <spirit:description>
Control Register
</spirit:description>
      </spirit:register>
      <spirit:register>
        <spirit:name>scalerReload</spirit:name>
 <spirit:addressOffset>0xc</spirit:addressOffset>
        <spirit:size>32</spirit:size>
        <spirit:access>read-write</spirit:access>
        <spirit:description>
Scaler Reload Register
</spirit:description>
      </spirit:register>
    </spirit:addressBlock>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
  </spirit:memoryMap>
</spirit:memoryMaps>
```

Furthermore, each register can be divided in each of its field. For example, the scalar reload register is divided into two fields:



In the XML, the `register` element for the reload register is updated to reflect the different fields as follow:

```
<spirit:register>
  <spirit:name>scalerReload</spirit:name>
  <spirit:addressOffset>0xc</spirit:addressOffset>
  <spirit:size>32</spirit:size>
  <spirit:access>read-write</spirit:access>
  <spirit:field>
    <spirit:name>scalerReloadValue</spirit:name>
    <spirit:bitOffset>0</spirit:bitOffset>
    <spirit:bitWidth>12</spirit:bitWidth>
    <spirit:access>read-write</spirit:access>
    <spirit:description>Scaler reload value</spirit:description>
  </spirit:field>
  <spirit:field>
    <spirit:name>reserved</spirit:name>
    <spirit:bitOffset>12</spirit:bitOffset>
    <spirit:bitWidth>21</spirit:bitWidth>
    <spirit:access>read-write</spirit:access>
    <spirit:description>Reserved</spirit:description>
  </spirit:field>
  <spirit:description>Scaler Reload register</spirit:description>
</spirit:register>
```

9.1.4 Describing the hardware model

The hardware model is quite simple in the case of the Leon IP components since the only provided files are the VHDL source. The main elements we want to describe are:

1. The views a design environment (DE) can work on
2. The signals list
3. The hardware parameters mapping VHDL generics

Since the only provided files are the VHDL sources, we only define one view referencing the fileset containing these VHDL source files. Four environment identifiers are also defined to specify to the DE that three simulators can be used to simulate the VHDL: Mentor ModelSim, Cadence NcSim and Synopsys VCS, and one tool to synthesize the IP: Synopsys DesignCompiler:

```
<spirit:envIdentifier>:modelsim.mentor.com:</spirit:envIdentifier>
<spirit:envIdentifier>:ncsim.cadence.com:</spirit:envIdentifier>
<spirit:envIdentifier>:vcs.synopsys.com:</spirit:envIdentifier>
<spirit:envIdentifier>designcompiler.synopsys.com</spirit:envIdeentifier>
```

Furthermore, we define the model name to specify the top level entity and architecture to be used in a configuration file:

```
<spirit:modelName>leon2_Uart(struct)</spirit:modelName>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Using this parameter, the VHDL configuration looks like:

```
for uart_1 : leon2_Uart
      use entity leon2_uart.leon2_Uart(struct)
end for;
```

Finally, we define two references. The first one specifies the files that are used inside the view:

```
<spirit:fileSetRef>fs-vhdlSource</spirit:fileSetRef>
```

The second one specifies the timing constraints valid for this view:

```
<spirit:constraintSetRef>normal</spirit:constraintSetRef>
```

The complete `views` element looks like:

```
<spirit:views>
  <spirit:view>
    <spirit:name>vhdlsource</spirit:name>
    <spirit:envIdentifier>:modelsim.mentor.com:</spirit:envIdentifier>
    <spirit:envIdentifier>:ncsim.cadence.com:</spirit:envIdentifier>
    <spirit:envIdentifier>:vcs.synopsys.com:</spirit:envIdentifier>

<spirit:envIdentifier>:designcompiler.synopsys.com:</spirit:envIdentifier
>
    <spirit:language>vhdl</spirit:language>
    <spirit:modelName>leon2_Uart(struct)</spirit:modelName>
    <spirit:fileSetRef>fs-vhdlSource</spirit:fileSetRef>
    <spirit:constraintSetRef>normal</spirit:constraintSetRef>
  </spirit:view>
</spirit:views>
```

From the signals side, the VHDL entity of the uart gives us the list of signals:

```
entity leon2_Uart is
  generic ( EXTBAUD  : boolean );
  port (
    rst     : in  std_logic;
    clk     : in  clk_type;
    psel    : in  Std_Ulogic;
    penable : in  Std_Ulogic;
    paddr   : in  Std_Logic_Vector(31 downto 0);
    pwrite  : in  Std_Ulogic;
    pwdata  : in  Std_Logic_Vector(31 downto 0);
    prdata  : out Std_Logic_Vector(31 downto 0);
    rxd     : in std_logic;
    ctsn    : in std_logic;
    scaler  : in std_logic_vector(7 downto 0);
    rxen    : out std_logic;
    txen    : out std_logic;
    flow    : out std_logic;
    irq     : out std_logic;
    rtsn    : out std_logic;
    txd     : out std_logic  );
end leon2_Uart;
```

In XML, this list of signals is included in the `signals` element (see paragraph 9.1.9 for the complete list):

```
<spirit:signals>
  <spirit:signal>
    <spirit:name>clk</spirit:name>
```

© Copyright The SPIRIT Consortium 2006. All rights reserved.

```
      <spirit:direction>in</spirit:direction>
      <spirit:clockDriver spirit:clockName="clk">
        <spirit:clockPeriod spirit:id="ClockPeriod" spirit:prompt="Clock
  Pulse Period:" spirit:resolve="user">8</spirit:clockPeriod>
  <spirit:clockPulseOffset spirit:id="ClockPulseOffset"
  spirit:prompt="Clock Pulse Offset:"
  spirit:resolve="user">4</spirit:clockPulseOffset>
        <spirit:clockPulseValue spirit:id="ClockPulseValue"
  spirit:prompt="Clock Pulse Value:"
  spirit:resolve="user">1</spirit:clockPulseValue>
  <spirit:clockPulseDuration spirit:id="ClockPulseDuration"
  spirit:prompt="Clock Pulse Duration:"
  spirit:resolve="user">4</spirit:clockPulseDuration>
      </spirit:clockDriver>
      <spirit:export spirit:configGroups="export" spirit:id="sig_clk"
                     spirit:prompt="clk">false</spirit:export>
    </spirit:signal>

    <spirit:signal>
      <spirit:name>rst</spirit:name>
      <spirit:direction>in</spirit:direction>
      <spirit:export spirit:configGroups="export" spirit:id="sig_rst"
                     spirit:prompt="rst">false</spirit:export>
      <spirit:signalConstraintSets>
        <spirit:signalConstraints>
          <spirit:timingConstraint spirit:clockName="virtual_clk">
            <spirit:percentOfPeriod>50</spirit:percentOfPeriod>
          </spirit:timingConstraint>
        </spirit:signalConstraints>
      </spirit:signalConstraintSets>
    </spirit:signal>
    …
    <spirit:signal>
      <spirit:name>ctsn</spirit:name>
      <spirit:direction>in</spirit:direction>
      <spirit:defaultValue>
        <spirit:value spirit:format="long"
          spirit:id="sigdefVal_ctsn">0</spirit:value>
      </spirit:defaultValue>
      <spirit:export spirit:configGroups="export"
          spirit:format="bool"
          spirit:id="sig_ctsn" spirit:prompt="ctsn"
          spirit:resolve="user">false</spirit:export>
      <spirit:signalConstraintSets>
        <spirit:signalConstraints>
          <spirit:timingConstraint spirit:clockName="virtual_clk">
     <spirit:percentOfPeriod>75</spirit:percentOfPeriod>
          </spirit:timingConstraint>
        </spirit:signalConstraints>
      </spirit:signalConstraintSets>
    </spirit:signal>
  </spirit:signals>
```

The `defaultValue` element specified for a signal in a component contains two subelements `value` and `strength`. The component specification of a default value, for a signal, can override the default value specified in a bus definition. In either case the default value is only used when no other signal is connected to this signal by the design. The `value` and `strength` elements work together to produce the results shown in the table below.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Table 3 Value and strength signal decoding

value↓\strength→	strong (default)	weak
0	Forced to 0	Weakly pulled low
1	Forced to 1	Weakly pulled high
Unspecified (default)	Forced to unknown	Tristate

As mentioned before, the user if needed can export the external bus. This is specified in the signal list by putting the attribute `spirit:resolve="user"` in the `export` element of each exportable signal.

Finally, the entity of the UART includes a generic to specify how the baud rate is generated: either internally with an internal clock divider register or externally. This has to be specified by the user in order to generate the correct generic map. The generic is specified in the XML thanks to the `modelParameter` element:

```
<spirit:modelParameter spirit:choiceRef="EXTBAUDChoice"
    spirit:choiceStyle="combo"
    spirit:configGroups="requiredConfig"
    spirit:dataType="boolean"
    spirit:format="choice"
    spirit:id="EXTBAUD"
    spirit:name="EXTBAUD"
    spirit:prompt="Set baud rate externally:"
    spirit:resolve="user">false</spirit:modelParameter>
```

If the value specified by the user is true, it will appear in the generic map when the component is instantiated:

```
uart_1 : leon2_Uart
  generic map (
    EXTBAUD => true
  )
```

The full description of the `model` element can be found in paragraph 6.1.7.

9.1.5 Describing the configuration choices

In the UART IP, the `choices` element will give the different choices for the hardware parameters. In the `modelParameter` element we described in the previous paragraph, there is an attribute `spirit:choiceRef="EXTBAUDChoice"` referencing the choice id specified in this section (EXTBAUDChoice). Since the hardware parameter is a boolean, there are only two choices:

```
<spirit:choices>
   <spirit:choice>
     <spirit:name>EXTBAUDChoice</spirit:name>
     <spirit:enumeration spirit:text="false">false</spirit:enumeration>
     <spirit:enumeration spirit:text="true">true</spirit:enumeration>
   </spirit:choice>
</spirit:choices>
```

9.1.6 Describing the file sets

The source code of the IP is provided as a set of VHDL file. Several files are common to all Leon IP components and have been placed in a common directory in the library

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

root directory (the relative path from the xml file is ../../common). The description of the UART (uart.vhd) itself is placed in a directory, hdlsrc, under the directory containing the XML description of the IP. A wrapper was also added around the IP description in order to unbundle the AMBA signals that were packaged as records (leon2_Uart.vhd). The directory structure for the UART is the following:

```
Common
Uart
     |-1.00
        |-uart.xml
        |-      hdlsrc
           |- leon2_Uart.vhd
           |- uart.vhd
```

The compilation order of all the files, including the common one, is the following:

- ../../common/target.vhd
- ../../common/device.vhd
- ../../common/config.vhd
- ../../common/sparcv8.vhd
- ../../common/iface.vhd
- ../../common/amba.vhd
- ../../common/ambacomp.vhd
- ../../common/macro.vhd
- ../../common/tech_generic.vhd
- ../../common/tech_atc25.vhd
- ../../common/tech_atc35.vhd
- ../../common/tech_fs90.vhd
- ../../common/tech_umc18.vhd
- ../../common/tech_virtex.vhd
- ../../common/tech_tsmc25.vhd
- ../../common/tech_proasic.vhd
- ../../common/tech_axcel.vhd
- ../../common/multlib.vhd
- ../../common/tech_map.vhd
- hdlsrc/uart.vhd
- hdlsrc/leon2_Uart.vhd

The directory structure and the compilation order is reflected inside the XML file since all the `file` inside a `fileSet` have to be placed in the same order as the compilation order:

```
<spirit:fileSet spirit:fileSetId="fs-vhdlSource">
  <spirit:file>
    <spirit:name>../../common/target.vhd</spirit:name>
    <spirit:fileType>vhdlSource</spirit:fileType>
  </spirit:file>
  <spirit:file>
    <spirit:name>../../common/device.vhd</spirit:name>
    <spirit:fileType>vhdlSource</spirit:fileType>
  </spirit:file>
  …
  <spirit:file>
    <spirit:name>hdlsrc/uart.vhd</spirit:name>
    <spirit:fileType>vhdlSource</spirit:fileType>
  </spirit:file>
  <spirit:file>
    <spirit:name>hdlsrc/leon2_Uart.vhd</spirit:name>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
    <spirit:fileType>vhdlSource</spirit:fileType>
    <spirit:logicalName>leon2_uart</spirit:logicalName>
  </spirit:file>
   </spirit:fileSet>
```

Each fileset has a `fileSetId` attribute in order to be able to reference them inside the `model`. In the case of the UART, only one fileset is available i.e. the VHDL source code.

9.1.7 Description of timing constraints

The UART only contains one set of timing constraints reference inside the view:

```
<spirit:componentConstraintSets>
  <spirit:componentConstraints spirit:constraintSetId="normal">
    <spirit:designRuleConstraints>
      <spirit:maxCap>0.4</spirit:maxCap>
      <spirit:maxTransition>
        <spirit:riseDelay spirit:units="ps">700</spirit:riseDelay>
        <spirit:fallDelay spirit:units="ps">700</spirit:fallDelay>
      </spirit:maxTransition>
      <spirit:maxFanout>2</spirit:maxFanout>
    </spirit:designRuleConstraints>
  </spirit:componentConstraints>
</spirit:componentConstraintSets>
```

For more explanations regarding timing constraints, see section 9.2.

9.1.8 Other Clock Drivers

All signal timing constraints are related to a virtual clock that is specified inside the following section (period is 8 ns, going from 0 to 1 after 4 ns and staying high for 4 ns):

```
<spirit:otherClockDrivers>
  <spirit:clockDriver spirit:clockName="virtual_clock">
    <spirit:clockPeriod spirit:id="VirtualClockPeriod"
      spirit:prompt="Virtual Clock Period:"
      spirit:resolve="user">8</spirit:clockPeriod>
    <spirit:clockPulseOffset spirit:id="VirtualClockPulseOffset"
      spirit:prompt="Virtual Clock Pulse Offset:"
      spirit:resolve="user">4</spirit:clockPulseOffset>
    <spirit:clockPulseValue spirit:id="VirtualClockPulseValue"
      spirit:prompt="Virtual Clock Pulse Value:"
      spirit:resolve="user">1</spirit:clockPulseValue>
    <spirit:clockPulseDuration
      spirit:id="VirtualClockPulseDuration"
      spirit:prompt="Virtual Clock Pulse Duration:"
      spirit:resolve="user">4</spirit:clockPulseDuration>
  </spirit:clockDriver>
</spirit:otherClockDrivers>
```

9.1.9 Example Source Code

The complete source code of this example is packaged within the uart.xml Leon IP. Several other examples of component packaging are provided within the Leon IPs:

- Ahbbus.xml: example of an AHB bus fabric packaging with 2 masters and 2 slaves
- Ahbstat.xml: ahb bus status registers packaging

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

- Apbbus.xml: example of an APB bus fabric with 4 slaves
- Apbmst.xml: AHB/APB bridge
- Dma.xml: direct memory access controller
- Irqcrtl.xml: interrupt controller
- Leon2Proc.xml: packaging example of one configuration of the Leon2 processor
- Timers.xml: packaging of 2 timers and 1 watchdog

9.2 Implementation Constraints

This constraint schema is not intended to model all possible constraints. It is specifically targeted at the domain of reusable intellectual property (IP) components. As such, the constraints that are supported are limited to those that can be specified in a technology portable fashion and which can be reasonably specified with little knowledge of the target environment. There are four types of constraints that can be specified, as described below. Constraints can be defined in either the component or bus definition. Constraints in the component override those defined in the bus definition.

9.2.1 Timing Constraints

A timing constraint defines an external timing requirement that must be honored for a component to function properly. This constraint applies to top-level ports on a component and indicates how much time is required for paths through the top-level port, external to the component. Each constraint is relative to a particular clock and the constraints can be different for rising and falling edge transitions at the top-level port. References clocks must exist within the component definition. They can be defined via the clockDriver element of a signal, or using the otherClocks element within the component. Example timing constraints are shown below.

```
<spirit:timingConstraint spirit:clockName="clk1">
  <spirit:percentOfPeriod>50</spirit:percentOfPeriod>
</spirit:timingConstraint>
<spirit:timingConstraint spirit:clockName="clk2"
    spirit:clockEdge="fall" spirit:delayType="max">
  <spirit:percentOfPeriod>25</spirit:percentOfPeriod>
</spirit:timingConstraint>
<spirit:timingConstraint spirit:clockName="clk3">
  <spirit:delay spirit:units="ps">300</spirit:delay>
</spirit:timingConstraint>
```

9.2.2 External Load/Drive Constraints

These constraints are available to provide a description of what is connected externally to a top-level port of the component. Specification of these constraints allows for more accurate timing analysis within the component as a stand-alone entity. Drive constraints can be specified on input ports only, and load constraints on output ports only.

A drive constraint can be used to specify the type of cell that will be driving the input port or to specify an explicit drive strength (resistance) associated with the input port. The specification of a driving cell can be done in a technology portable way by specifying either a cell class or function, and corresponding cell strength. A load constraint can be used to specify the type of cell or cells that will be acting as a load on an output port. The load specification can be defined as a user specified number

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

of cells of a particular type, or as an explicit capacitance value. Example constraints are shown below.

```
<spirit:driveConstraint>
  <spirit:cellSpecification>
    <spirit:cellFunction
       spirit:cellStrength="low">nand2</spirit:cellFunction>
  </spirit:cellSpecification>
</spirit:driveConstraint>
<spirit:driveConstraint>
  <spirit:resistance
       spirit:units="kohm">1000</spirit:resistance>
</spirit:driveConstraint>
<spirit:loadConstraint>
  <spirit:cellSpecification>
    <spirit:cellClass
        spirit:cellStrength="high">sequential</spirit:cellClass>
  </spirit:cellSpecification>
  <spirit:count>3</spirit:count>
</spirit:loadConstraint>
```

9.2.3 Point to Point Timing Requirements

These constraints are available to define exceptions to the standard timing requirements of a component. They are available to indicate false paths, multi-cycle paths, and special point-to-point timing requirements. Each of these constraints is defined using a path specifier and some additional information that is specific to the constraint type. A path specifier is a list of path start points, path end points, and path through points that define the path or paths to which the constraint applies. If a particular path specification point might match more than one design element, an attribute can be added to indicate the expected object type (clock, port, pin, cell). Example constraints are shown below.

```
<spirit:falsePath>
  <spirit:pathSpecifier>
    <spirit:from>data1</spirit:from>
  </spirit:pathSpecifier>
</spirit:falsePath>

<spirit:falsePath>
  <spirit:pathSpecifier>
    <spirit:from spirit:pathElement="clock">clk1</spirit:from>
    <spirit:to>clk2</spirit:to>
  </spirit:pathSpecifier>
</spirit:falsePath>

<spirit:falsePath>
  <spirit:pathSpecifier>
    <spirit:from>Ain</spirit:from>
    <spirit:through>U1/Z</spirit:through>
    <spirit:through>U1/U2/Out</spirit:through>
    <spirit:to>Bout</spirit:to>
  </spirit:pathSpecifier>
</spirit:falsePath>

<spirit:multiCyclePath>
  <spirit:pathSpecifier>
    <spirit:from>data2</spirit:from>
    <spirit:to>some_output</spirit:to>
  </spirit:pathSpecifier>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
  <spirit:cycles>2</spirit:cycles>
</spirit:multiCyclePath>

<spirit:timedPath>
  <spirit:pathSpecifier>
    <spirit:to>Xout</spirit:to>
    </spirit:pathSpecifier>
  </spirit:pathSpecifier>
  <spirit:delay spirit:units="ps">500</spirit:delay>
</spirit:timedPath>
```

9.2.4 Design Rule Constraints

These constraints are available for specifying technology constraints that must be honored for proper circuit operation. These constraints can be specified on individual top-level ports or at the component level, implying that they are applicable across the entire design. Constraints are available for specifying minimum and maximum capacitance requirements, minimum and maximum signal transition times, and maximum fan-out requirements. Constraints specified on a specific top-level port override those specified for the component, if they are more restrictive. Example constraints are shown below.

```
<spirit:designRuleConstraints>
  <spirit:maxFanout>4</spirit:maxFanout>
</spirit:designRuleConstraints>

<spirit:designRuleConstraints>
  <spirit:minCap>4</spirit:minCap>
  <spirit:maxCap spirit:units="ff">12</spirit:maxCap>
</spirit:designRuleConstraints>
```

9.3 Loose Generator Dump

The loose generator interface generates an input file for the generator describing the different instances contained in a design, the location of the dumped DOM XML file and the selected instances on which the generator will work. In this section, we will show on a simple design, the content of the generator input file. It will mainly show the usage of the following elements:

- `designFile`
- `busDefinitionFiles`
- `componentDefinitionFiles`
- `selectedInstances`

The design contains two levels of hierarchy. The top level contains the following instances:

- dummy_AHB_master_1
- dummy_APB_master_1
- irqctrl_2
- uart_3
- uart_4
- a Subcomponent_1
- while Subcomponent_1 contains:
- dummy_AHB_master_1

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

- dummy_APB_master_1
- irqctrl_1
- timers_1
- uart_1
- uart_2
- Both levels contain the following busses:
- AHB
- APB
- Interrupt
- SingleInterruptBus

There are several ways to place the dumped xml files in directories. In this example, the directory structure follows the design hierarchy:

```
DOM
|- <top_level_name>.xml
    |- busdef
    |   |- <bus1>.xml
    |   |- <bus2>.xml
    |   |- ...
    |- component
        |- <component_instance1>.xml
        |- <component_instance2>.xml
        |- <component_instance3>.xml
        |- ...
        |- <component_instance2>
            |- <subcomponent_instance1>.xml
            |- <subcomponent_instance2>.xml
        |- ...
```

This structure is being reflected in the generator input xml file in the LooseGeneratorDump directory in the Examples_Documentation_V1.1 tar file.

As specified inside `selectedInstances` element, the generator will work on uart_3 and irqctrl_2 instances.

9.4 Generator example

In this section, we will show a simple example of a generator using the input xml file described in the previous section. This generator takes the input file and generates an html file containing the documentation of the selected instances using an XSLT transform. There is no way back to the design environment.

The first part of the XSLT transform looks for the selected instances and, each time, place the instance in a variable called `SelectedInstance`:

```
<xsl:template match="/">
  <html>
  <body>
  <BODY BGCOLOR="#ffffff"></BODY>
  <xsl:for-each
select="spirit:looseGeneratorInvocation/spirit:selectedInstances/spirit:s
electedInstance">
    <xsl:variable name="SelectedInstance">
      <xsl:value-of select="."/>
    </xsl:variable>
…
  </xsl:for-each>
```

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

```
  </body>
  </html>
</xsl:template>
```

For each instance, the XSLT transform looks for the instance xml file path to get the data to be processed and displayed in html; the component name is placed inside the ComponentAttribute variable:

```
<xsl:template match="/">
  <html>
  <body>
  <BODY BGCOLOR="#ffffff"></BODY>
  <xsl:for-each
select="spirit:looseGeneratorInvocation/spirit:selectedInstances/spirit:s
electedInstance">
    <xsl:variable name="SelectedInstance">
      <xsl:value-of select="."/>
    </xsl:variable>
    <xsl:for-each
select="/spirit:looseGeneratorInvocation/spirit:componentDefinitionFiles/
spirit:componentDefinitionFile">
      <xsl:variable name="ComponentAttribute">
        <xsl:value-of select="@spirit:instanceRef"/>
      </xsl:variable>
…
    </xsl:for-each>
  </xsl:for-each>
  </body>
  </html>
</xsl:template>
```

If `ComponentAttribute` matches `SelectedInstance`, the html documentation for that instance is generated. The complete xslt transform is located in the examples tar file.

The generator itself is a simple shell script launching an xslt processor (for example, here the Apache Xalan Java processor in a UNIX style shell environment):

```
java -jar <Xalan install dir>/xalan.jar -IN \
looseGeneratorInput.xml -XSL ComponentDoc.xsl -OUT \
GeneratorDoc.htm
```

Where:

- LooseGeneratorInupt.xml is the generator input file described in section 9.3.
- ComponentDoc.xsl is the XSLT transform is described in this section.
- GeneratorDoc.htm is the HTML documentation

© Copyright The SPIRIT Consortium 2006. All rights reserved.

10 APPENDIX: DEFINITIONS AND NOTATION

10.1 Definitions

Term	Meaning
AHB	AMBA high speed bus
AMBA	Open specification on-chip backbone for interconnecting IP blocks
API	Application Programmers Interface. A method for accessing design and meta data in a procedural way involving a programming language and database for the purposes of adding value to existing and future design flows.
Architectural Rules	Generic rules that define how subsystems relate to platforms that relate to components of system design
ASIC	Application Specific Integrated Circuit
Bus	Collection of signals used to connect blocks connected to it involving both hardware and software protocols
Bus Bridge	An interface which connects 2 bus systems with different characteristics (i.e. bus type, bus speed)
Bus Interface	The interface of an IP to a bus
Channel	A special IP-XACT object that can be used to describe multi-point connections between regular components, which may require some interface adaptation.
Component	Configured IP stored in a design database
Configuration Manager	Object which creates and manages top-level meta-description of SoC design. Includes ability to annotate SoC schema with details of a specific SoC design including: IP versions, IP views, IP configuration, IP connectivity, IP constraints. Manages launch of IP generators and tool plug-ins, and meta-data updates occurring as a consequence of a launch. Manages update and retrieval of relevant IP meta-data from the IP repository. Driven by user through DE graphical users interface.
CSS	Cascading Style Sheets (CSS) used to style HTML & XML documents http://www.w3.org/Style/CSS
DE	Design Environment
Design Database	Working store for both meta-data and component info that helps create and verify systems and subsystems
Design Environment	The co-ordination of a set of tools and IP, or expressions of that IP (e.g., models) such that the system-design and implementation flows of a SoC re-use centric development flow is efficiently enabled. Co-ordination is managed through creation and maintenance of a meta-data description of the SoC

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Term	Meaning
DOM	The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page
EDA	Electronic Design Automation
EOS	Embedded Operating System
External Components	Components that do not end up on the SoC but are needed for total system verification
Generator	Combines component meta-data with architectural rules to provide a consistent system description that uses an API to generate specific design views or configurations for the purposes of supporting a number of design styles
Generator API	This API interfaces the generators and tool plug-ins to the configuration environment, allowing the execution of these scripts and code-elements against the SoC meta-description. The API enables the registration of new generators / plug-ins, export of SoC meta-data and update of that data following generator or plug-in execution, and handling of generator / plug-in error conditions which relate to the meta-data description
HDL	Hardware Description Language
IP	Intellectual property which is utilized in the context of a SoC design or design flow. Includes: specifications; design models; design implementation description; verification coordinators, stimulus generators, checkers and assertion / constraint descriptions; soft design objects (such as embedded software and real-time operating systems); design and verification flow information and scripts
IP Generators	Tools which create specific IP based upon SoC meta-data details entered into the configuration manager. Interfaces to IP repository for placing and retrieval of IP. Annotates completion details (e.g., generated IP or failure of generation of IP) back into configuration manager
IP Integrator	Integrator of configured IP and subsystems
IP Platform Architect	Creator of platform based architectures
IP Provider	Creator and supplier of IP
IP Repository	Database of IP
LAU	Least Addressable Unit of memory,
Legacy IP	IP that has no specific IP-XACT meta-data view
LGI	Loose Generator Interface
Master Interface	The bus interface that initiates a transaction (like a read or write) on a bus.
Memory Map	Organization of memory elements as seen from a Master interface.

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Term	Meaning
Meta Data	A tool interpretable way of describing the: design-history, locality, object association, configuration options, constraints against, and integration requirements of an IP object
Meta IP	Meta-data description of an IP object
Mirror Interface	Has the same (or similar) signals to its related direct bus interface, but the signal directions are reversed. So a signal that is an input on a direct bus interface would be an output in the matching mirror interface.
Monitor Interface	Is an interface used in verification that is neither a master, slave nor system interface.
Opaque Bridge	A bus interconnect that may modify the address.
Platform	Architectural (sub)system framework
Platform Consumer	User/group who builds a SoC based on a particular platform
Platform Provider	User/group that develops and delivers platforms to platform consumers
Platform Rules	Rules that define how components interface to a specific platform
PMD	Platform Meta Data defines the configurable parameters at the Design level
RTL	Register Transfer Level design
Schema	XML schemas provide a means for defining the structure, content and semantics of XML documents http://www.w3.org/XML/Schema
Schema API	This API allows the configuration manager to query the XML IP meta-data. Queries may be for the existence of IP, the structure of IP, or features offered by that IP such as configurability and interface protocol support. This API is also used for the import and export of meta-data when an IP block is extracted from, or imported back into, the IP management system
Semantic Rules	Additional rules to be applied to an XML description that cannot be expressed in the schema. Typically these are rules between elements in one of multiple XML files.
Slave Interface	The bus interface that terminates/consumes a transaction initiated by a master interface.
SLD	System Level Design
SoC	System on Chip
Style Sheets	Style sheets describe how documents are presented on screens, in print http://www.w3.org/Style
SubSystem	A configured or unconfigured set of connected components that have dependencies on other IP
System	A configured set of connected components
System Interface	Is an interface that is neither a master nor slave interface, and allows specialized (or non-standard) connections to a bus.

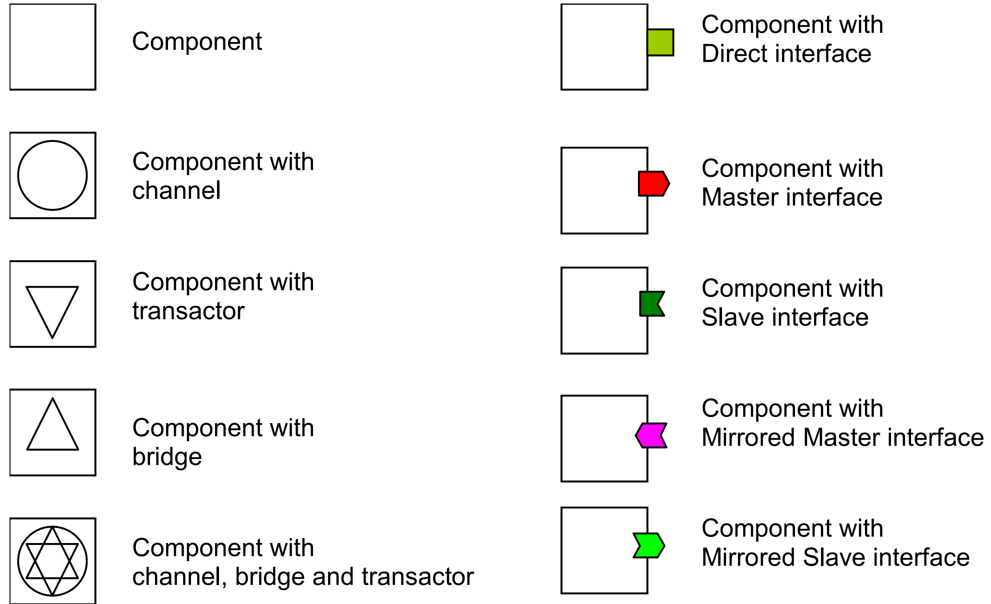
© Copyright The SPIRIT Consortium 2006.
All rights reserved.

Term	Meaning
TGI	Tight Generator Interface
Tool Plug-Ins	Tools which integrate IP based upon SoC meta-data details, and which prep IP for animation (e.g., simulation, emulation), optimization (e.g., synthesis) and verification (e.g., regression-suite generation). Annotates completion details (e.g., integrated SoC IP or failure of integration) back into configuration manager
Transparent Bridge	A bus interconnect that does not modify the address
Use Model	A process method of working with a tool
User Interface	Methods of interacting between a tool and its user.
Validation	Proving the correctness of construction of a set of components
Verification	Proving the behavior of a set of connected components
View	An implementation of a component. A component may have multiple views, each with it's own function in the design flow.
VIP	Verification IP, components included in a design for verification purposes.
VLNV	Each IP-XACT object is assigned a unique VLNV (Vendor Library Name Version) that is defined in the header of each XML file.
VSIA	Virtual Socket Interface Alliance
WBI	White Box Interface, internal points in the IP to be probed or driven by verification tools and/or test benches
XML	Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879) http://www.w3.org/XML
Xpath	An expression language used by XSLT to access or refer to parts of an XML document http://www.w3.org/TR/xpath
XSL	XSL is a language for expressing style-sheets. XSL can be used to transform XML data into HTML/CSS documents http://www.w3.org/Style/XSL
XSLT	A language for transforming XML documents into other XML documents http://www.w3.org/TR/xslt
3MD	3 levels of meta-data refers to a hierarchy of meta-data used to support platform-based SoC architectures. The lowest level defines IP parameters and constraints and is known as the IP-level. The second level is known as the platform-level that can be used to further constrain and capture platform rules for all SoC derivatives, and the third level is the chip level. for the purposes of system, production and verification tests needed to be captured for re-use and reproducibility. This 3-level hierarchy of data is known as the: 3md methodology

© Copyright The SPIRIT Consortium 2006.
All rights reserved.

10.2 Notations

The following notations are used in this document:



10.3 Last Page of Dcument