Updated March 23, 2012

Antti Kamppi, Joni-Matti Määttä, Lauri Matilainen, Erno Salminen, Timo D. Hämäläinen

Introduction to IP-XACT with Kactus2 + extensions

Outline

- Motivation
- Standard IP-XACT concepts and Kactus2 examples
  - IP-XACT design process and metadata objects
  - IP-XACT connections
  - Design hierarchy
  - Addressing
  - Design configurations
- Introduction to Kactus2 extensions for SW, HW/SW mappings and communication
- Kactus2 IP-XACT extensions and examples
  - Summary of standard and extended IP-XACT objects and elements inside objects
  - Communication interface and definition
  - SW component and design
  - System component and design

Metadata based design

Motivation

The Challenges

- **Design of IP (HW, SW) for reusability and portability**
  - Reuse is not the primary constraint due to tight project deadlines and/or performance
  - Too big overhead in time and effort to make it reusable
- **Errors in design data access and transfers**
  - Missing, outdated, informal documentation: Understanding the IP takes much time between people
  - The same information is typed in several times
  - Lots of manual inspections for correctness
  - Much time to search for correct versions, files and file dependencies
- **Platform and component dependency**
  - Locking into vendor, IP, tool, custom tool format
  - Problems in component availability trigger laborious re-design without any other need
- **Special expertise required**
  - Much more SW engineers available than HW/FPGA engineers

ITRS roadmap 2011-2026

Table SYSD2 SOC Consumer Driver Desi

Years	2011	2012	2013	2014	2015	2016	2017	2018
SoC-CP Total Logic Size	1.00	1.32	1.79	2.32	2.96	3.77	4.70	5.85
Required % of reused design	54%	58%	62%	66%	70%	74%	78%	82%
Required Productivity for new designs (Normalized to 2011)	1.00	1.22	1.60	2.02	2.50	3.08	3.72	4.48
Required Productivity for reused designs (Normalized to productivity for new designs at 2011)	1.00	1.22	1.60	2.02	2.50	3.08	3.72	4.48

ign Productivity Trends

2019	2020	2021	2022	2023	2024	2025	2026
7.45	9.70	11.65	15.50	19.56	24.40	31.23	38.10
86%	90%	92%	94%	95%	96%	97%	98%
5.51	6.93	8.17	10.67	13.34	16.48	20.89	16.48
5.51	6.93	8.17	10.67	13.34	16.48	20.89	25.24

INTERNATIONAL
TECHNOLOGY ROADMAP
FOR
SEMICONDUCTORS

2011 EDITION

SYSTEM DRIVERS

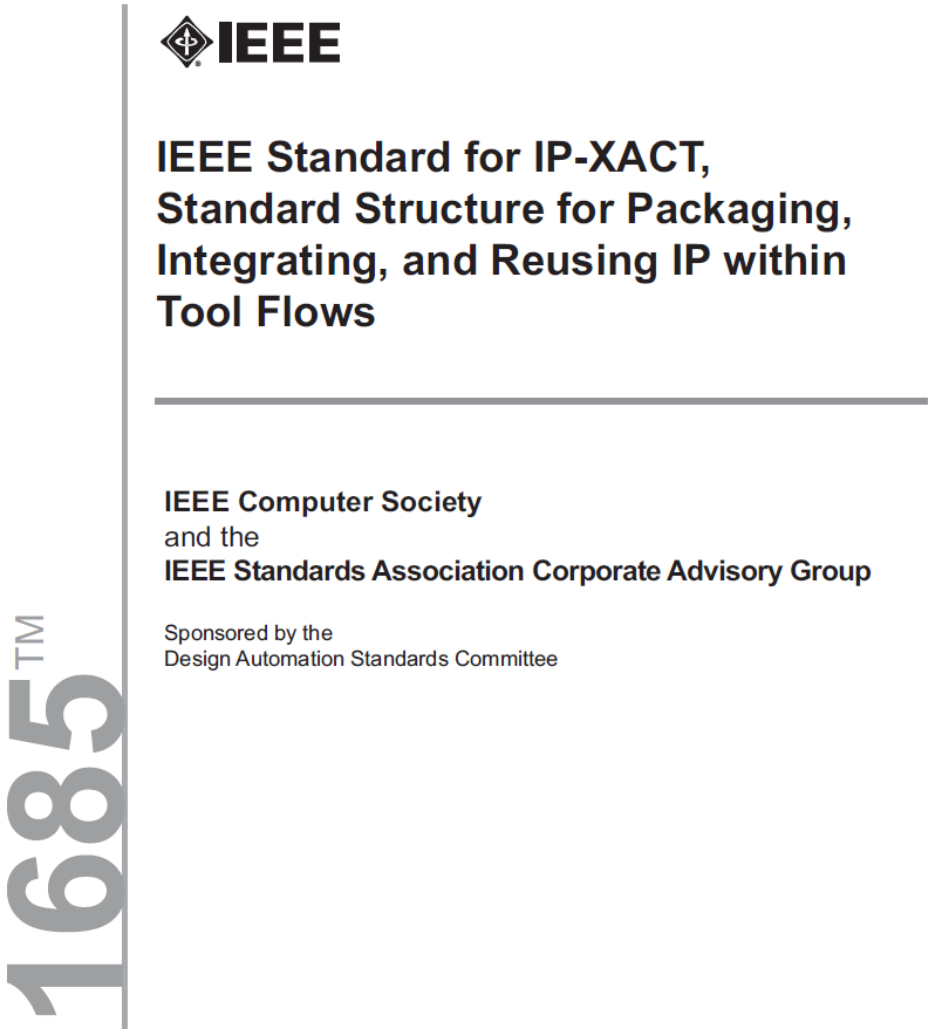
TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Computer Systems

The Solutions

- **Make designing for reuse fun**
  - Not an extra effort but elementary part of the design approach
  - Do reusability at once, not as a separate part
  - Easy to follow design flow & tools, also for SW engineers
- **Use metadata**
  - Replace informal documentation by machine readable metadata
  - Vendor, tool, abstraction independency
- **Open formats and tools**
  - Advanced tools are too expensive for SMEs, but an SME can invest its share of time to contribute open tools
- **Incremental adoption of metadata**
  - Any new method should be gradually deployed without disturbing existing design flow
  - Often too expensive modify all legacy IP, so allow both new and old exist at the same time
  - Metadata does not require any new IP content creation tools

IP-XACT

- IP-XACT 1.5 approved as standard in 2009
- Defines metadata format (XML) for describing IPs, designs, configurations, tools and automation scripts



IP-XACT Goals & Benefits Simplified

- ”The model”: **Vendor, implementation language, abstraction level and tool independent description of IP-blocks and systems**
- **Standardized integration and configuration flow independent** of vendors
- **Standard tool interfaces**
- **Metadata is “machine readable” information about the IP itself**
  - “Electronic databook”

Kactus2 in brief

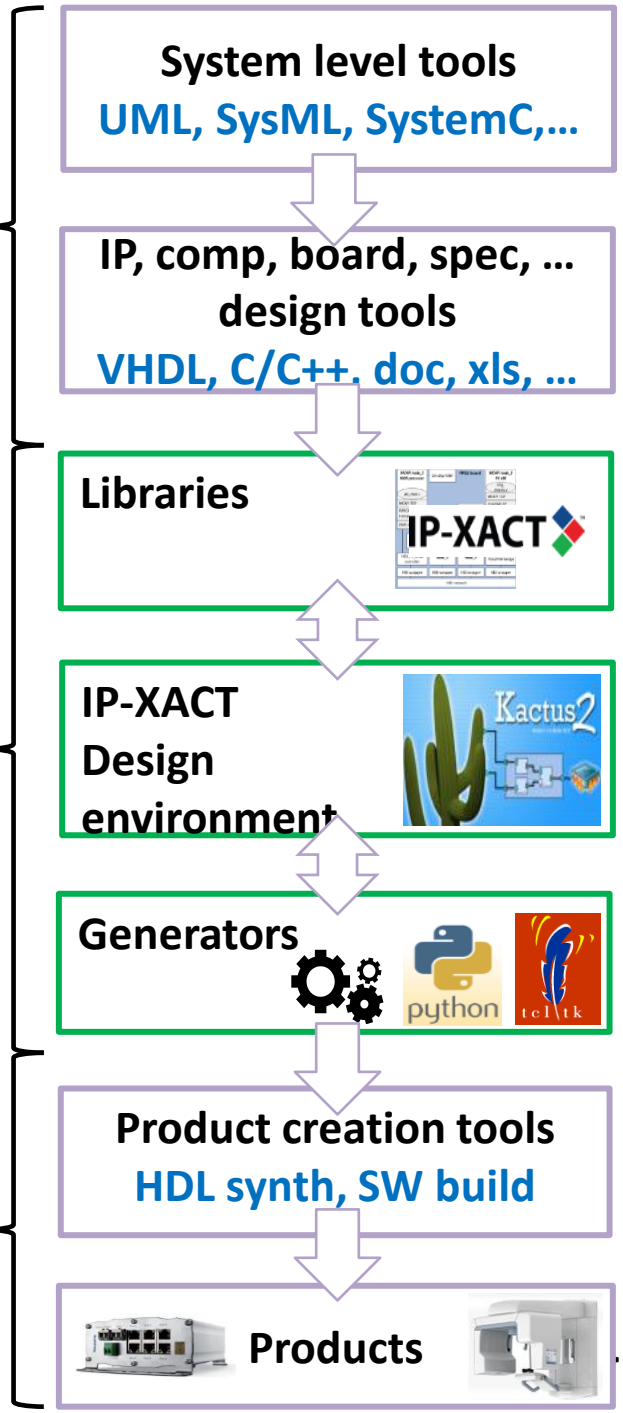
- Purpose: For schetching, packing, integrating and generating **both HW and SW** for embedded products at **several hierarchy levels**
- Goal: **Easiest to use** metadata based tool
- Objective: **Standard compliant**
  - IP-XACT/IEEE1685 + extensions
- Details: see http://funbase.cs.tut.fi

Meta-data wrt. traditional design flow

Product specification and design tools

IP-XACT XML metadata based management of design information

Implementation and production tools



Metadata based design flow

Typical design flow

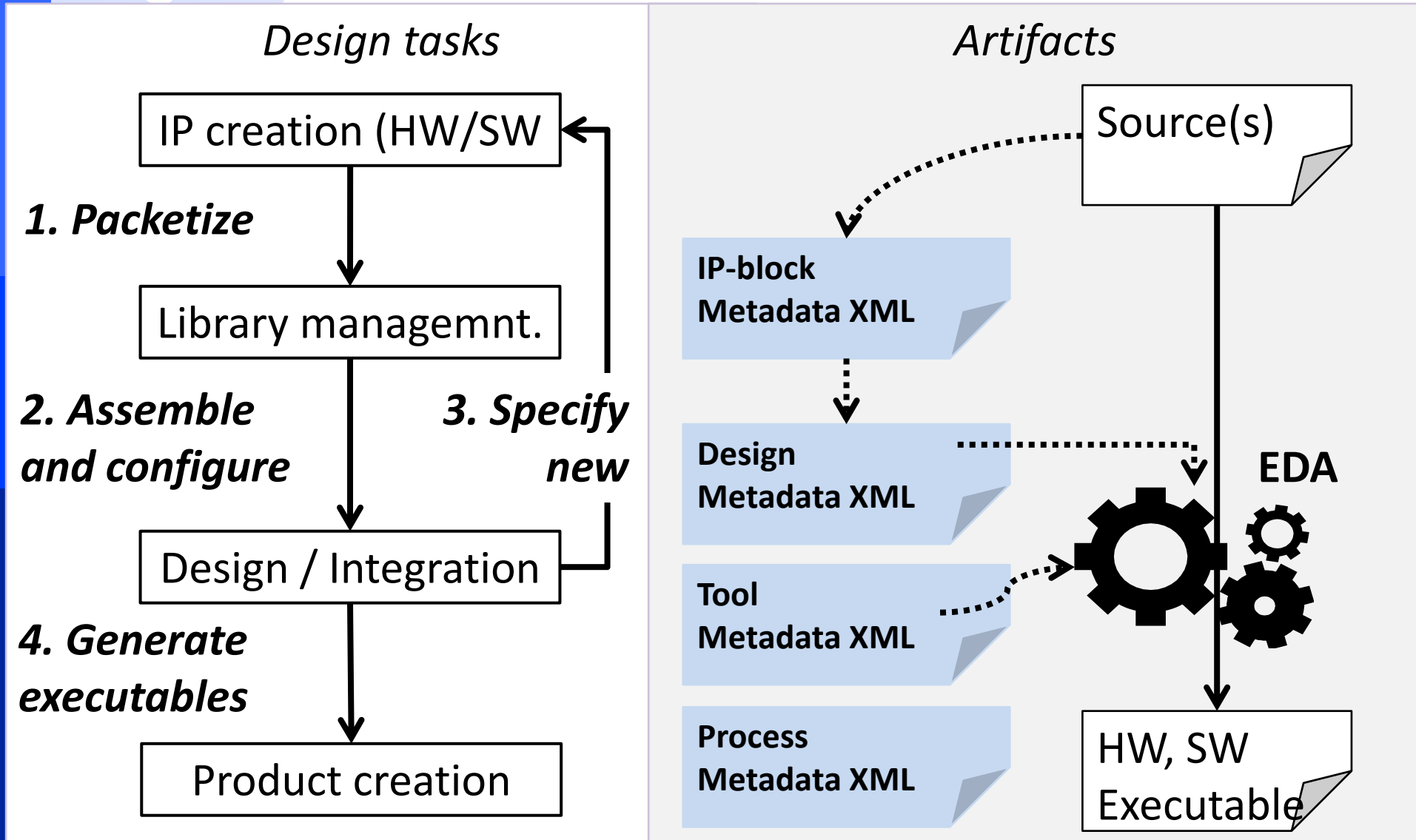
IP-XACT / IEEE1685

IP-XACT metadata objects

IP-XACT-based SoC design

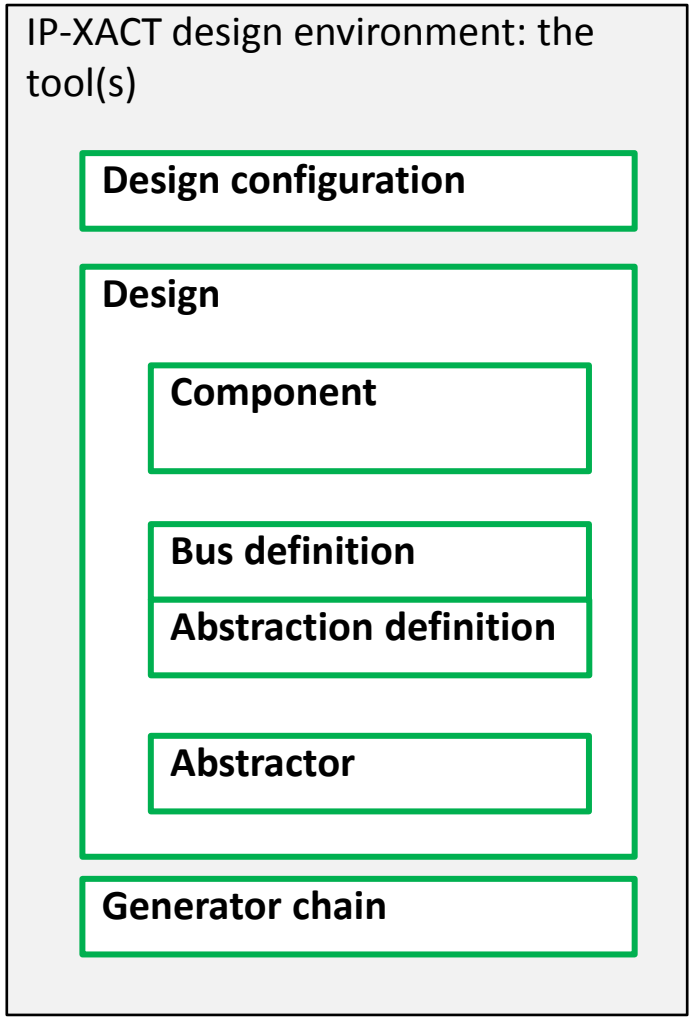
- Input is **IP-XACT component**
  - The sources are encapsulated and separated from the IP XML description
  - E.g. HDL source code is embedded via links to the source file in XML file
- IP-blocks are assembled together in a **IP-XACT design**
  - Structural, tool and vendor independent description of the system
  - IP-XACT design tools handle its objects, not IP source files
  - Compare: Mentor Graphics HDL designer is a visual VHDL design tool
- **Generators** are specific tools that configure or generate required components
  - IP-blocks and the design itself may have generic parameters, which are configured using generators that are typically scripts
- Output is **IP-XACT Design**
  - XML description of the complete system
  - The final configured IP-XACT design can be seen as “instructions” how to create the real system

IP-XACT-based SoC Design



IP-XACT objects

- **IP-XACT Objects** are XML metadata files representing SoC components, structure, and configurations
  - top-level XML schema definitions
- **IP-XACT design environment** handles these objects
- Each object have **unique ID**: ”**VLNV**”
  - Vendor, Library, Name, Version
- Objects refer to each other, forming spanning trees of the objects
- IP-XACT defines the structure of SoC, not the actual functionality or purpose



IP-XACT objects

1. A **component** description defines an IP or interconnect structure.
2. A **bus definition** description defines the type attributes of a bus.
3. An **abstraction definition** description defines the representation attributes of a bus.
4. A **design** description defines the configuration of and interconnection between components.
5. A **design configuration** description defines additional configuration information for a generator chain or design description.
6. An **abstractor** description defines an adaptor between interfaces of two different abstractions.
7. A **generator chain** description defines the grouping and ordering of generators.

VLNV header

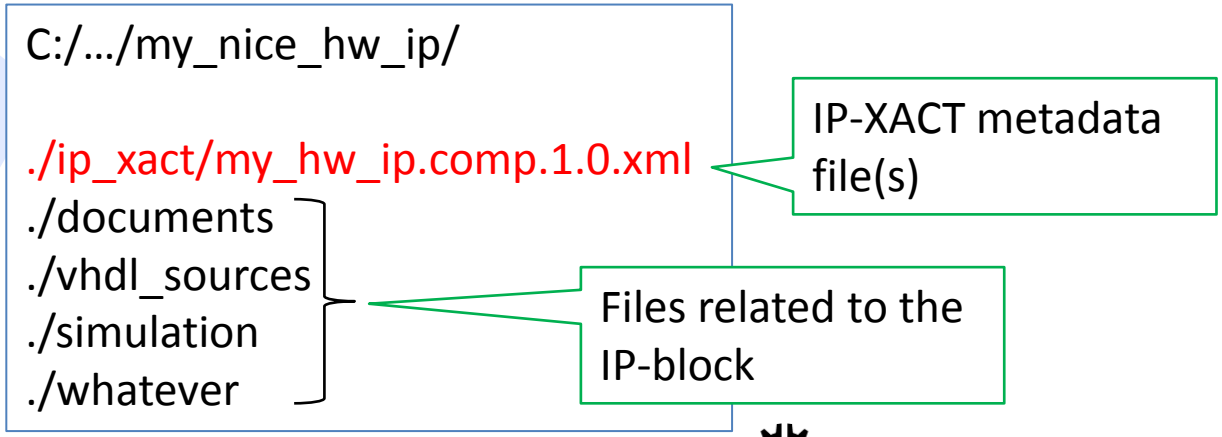
- Each IP-XACT object is identified by VLNV (Vendor Library Name Version)
- **VLNV is** used and **only used to refer between IP-XACT objects**
- VLNV is in the header of each XML file
  - Contents is not specified in standard
- VLNV must be unique for each IP-XACT object
  - E.g. IP-XACT design and component objects must have different VLNVs, since the object type is not counted when making references
  - Tools may add validation of legal references between IP-XACT objects (e.g. a design can not refer to bus definition)

IP-XACT design

```
…
<spirit:vendor>TUT.course.TKT3541</spirit:vendor>
<spirit:library>product</spirit:library>
<spirit:name>speden_spelit</spirit:name>
<spirit:version>1.0</spirit:version>
```

IP-XACT objects on disk

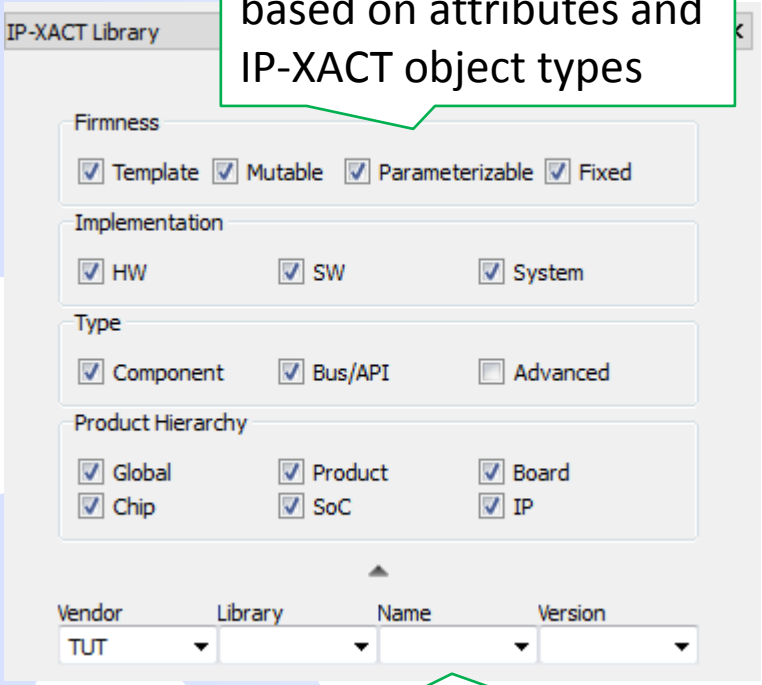
- Metadata is in practice XML-files on disk or database (depending on used tools)
- **IP-XACT metadata objects do not refer to each other by file names or library paths, only by VLNVs**
- Thus, **IP-XACT does not specify the location on disk or name of the XML-files**
  - Can be together with the related files, in a single global folder etc.
- **User is not expected to modify and manage the XML-files**
- **Tools explore the disk/database to find the metadata files** and to build an IP-XACT library model
- Example:



Example: IP-XACT object library

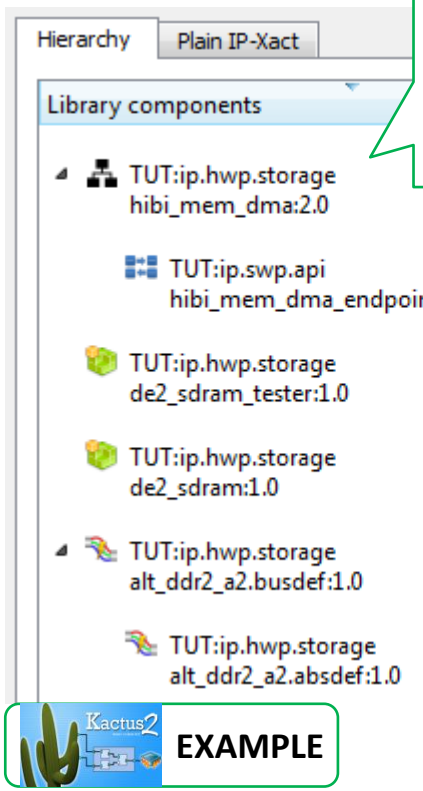
- The libraries will sooner or later blow out with tens of housands of files and IP-XACT objects
- Tools must be used to manage the library

Library display filtering based on attributes and IP-XACT object types

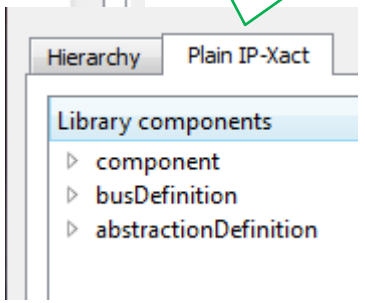


Filtering/Search based on VLNV

IP-XACT object library shown by VLNV references hierarchy starting from topmost level



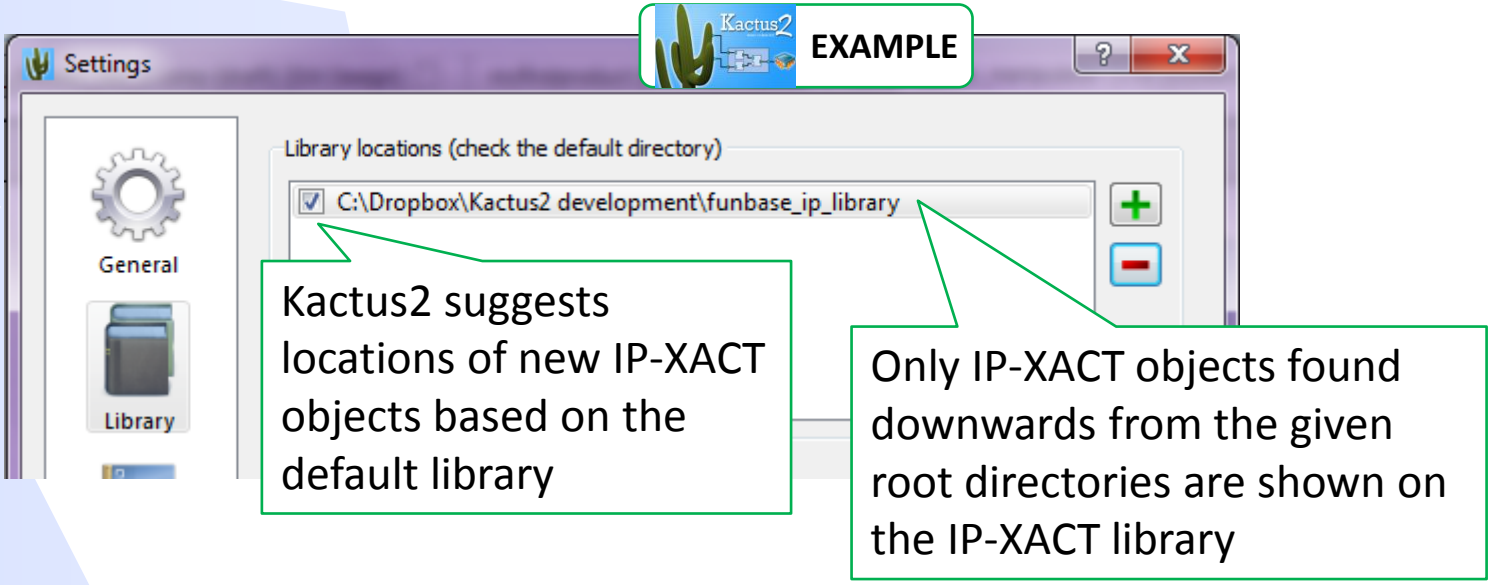
Library shown by IP-XACT object types



EXAMPLE

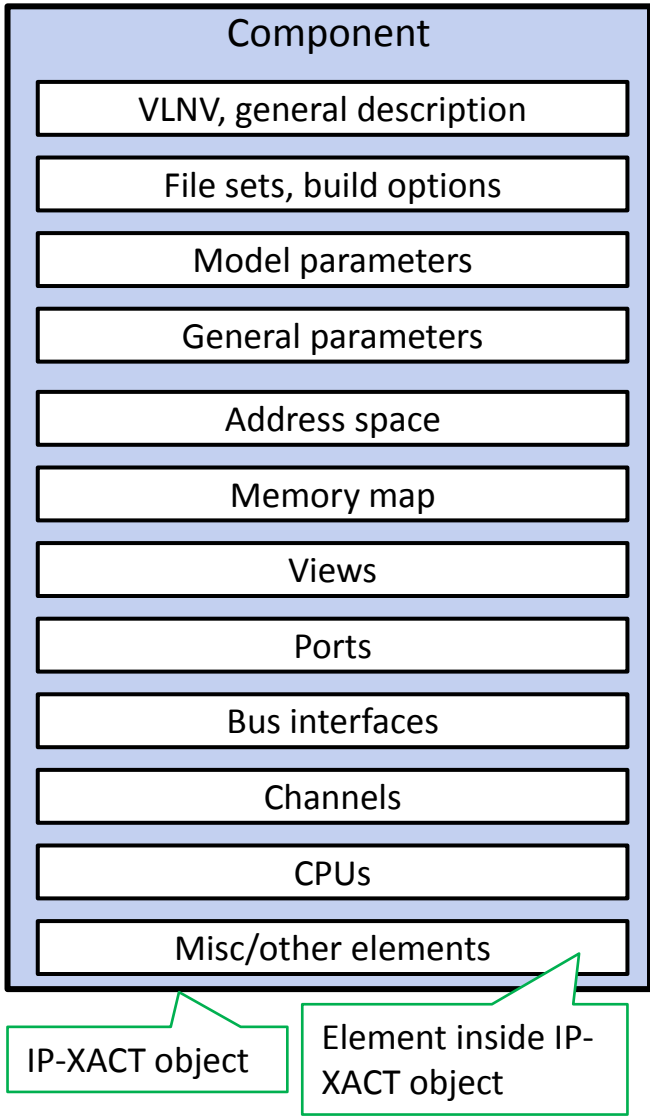
Example: IP-XACT libraries

- Tools create the object library based on XML-files found
- Kactus2 user must give root folder(s) to start scanning



IP-XACT Component

- IP-XACT component is a general placeholder describing all IP block types like processors, memories, accelerators and building blocks for buses and various interfaces.
- A component contains independent **elements** that can be referenced between each other within the component.
- **File sets** and file set **groups** are folder and file link collections
  - include information about used tools, description languages and instructions how to handle the files
- **Model parameters** are used to configure tools/implementation specified in **views**
- General **parameters** can be any configurable values or symbols related to this component
- Addressing includes **memory maps** and **address spaces** define addressable locations seen into and out from the component
- **Views** represent different purposes of the component
- **Ports** and **bus interfaces** are used to connect component to other components or special test structures
- Other information include e.g. signal constraints, routing information inside component, whether this component is a CPU, etc.

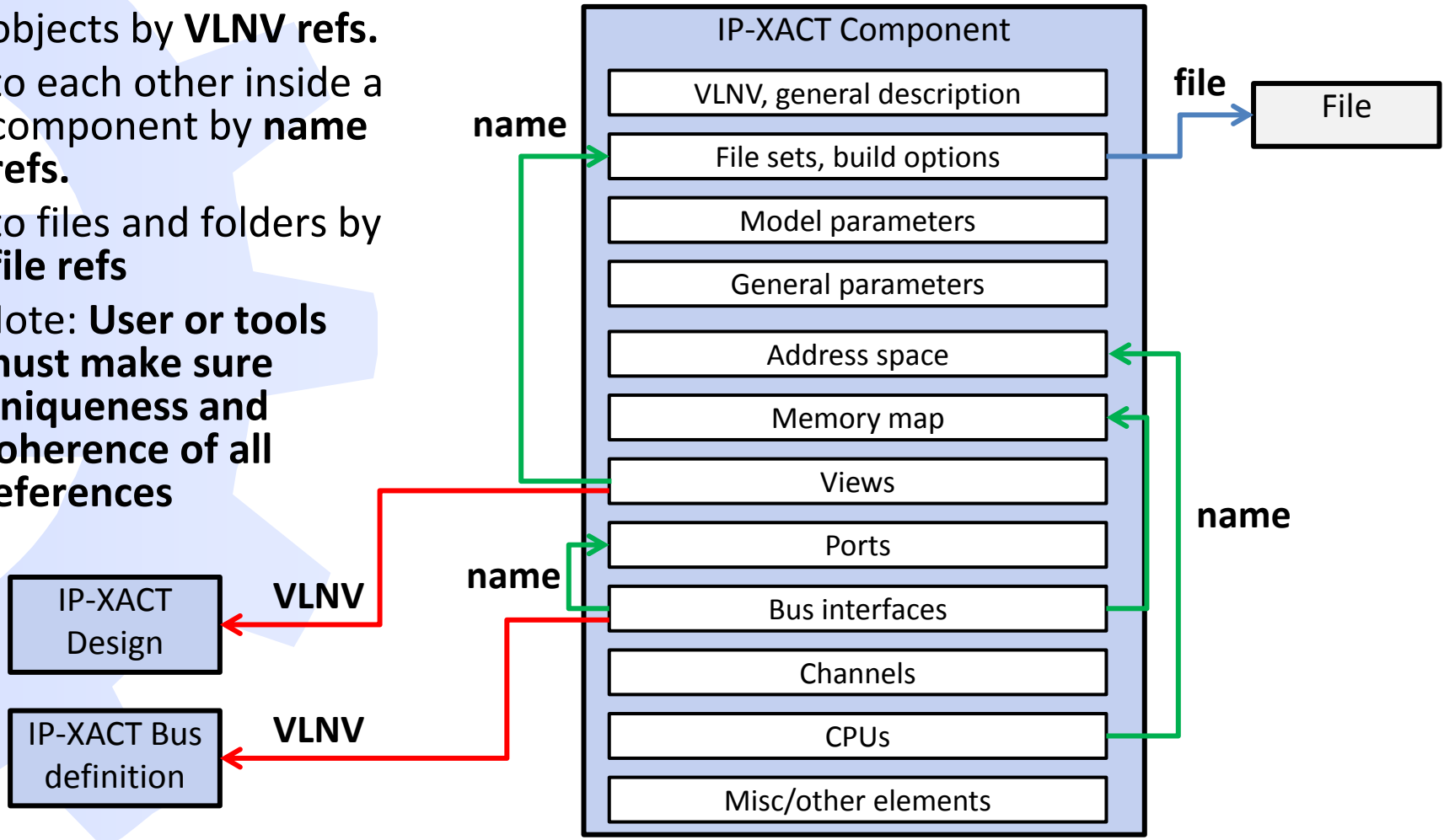


References (IP-XACT component)

- IP-XACT elements refer
1. to other IP-XACT objects by **VLNV refs.**
2. to each other inside a component by **name refs.**
3. to files and folders by **file refs**

- Note: **User or tools must make sure uniqueness and coherence of all references**

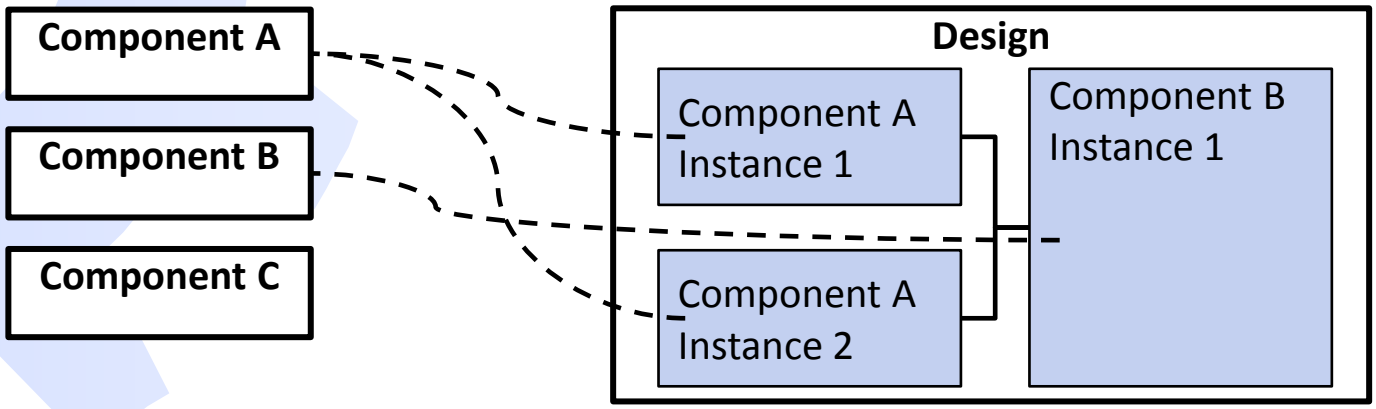
Example (not all refs. shown)



IP-XACT Design

- **IP-XACT design** is like a traditional schematic of components
- Describes a list of **component instances** and **connections** between each other

Library of IP-XACT components

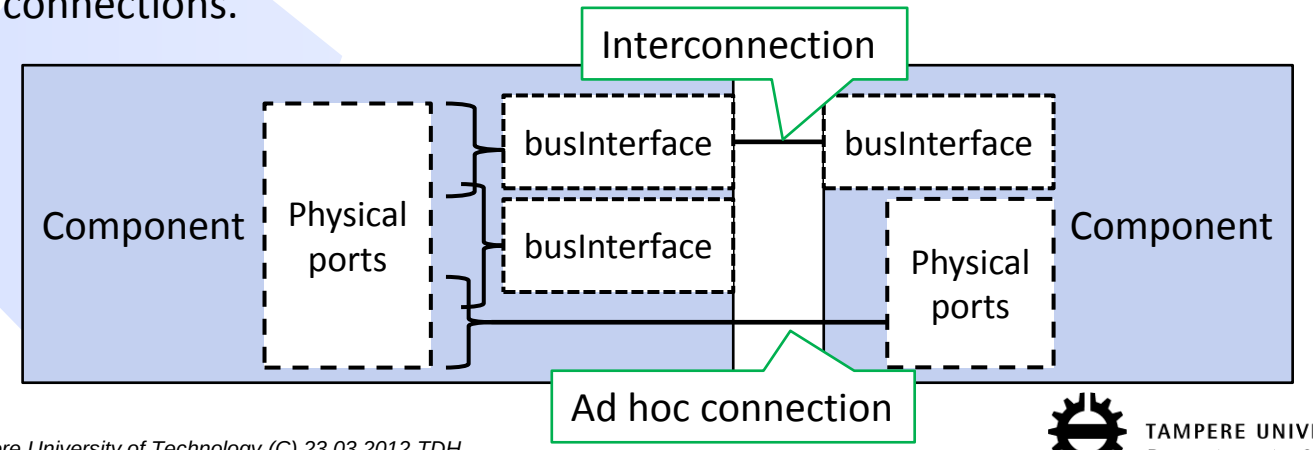


IP-XACT / IEEE1685

Integration and connections

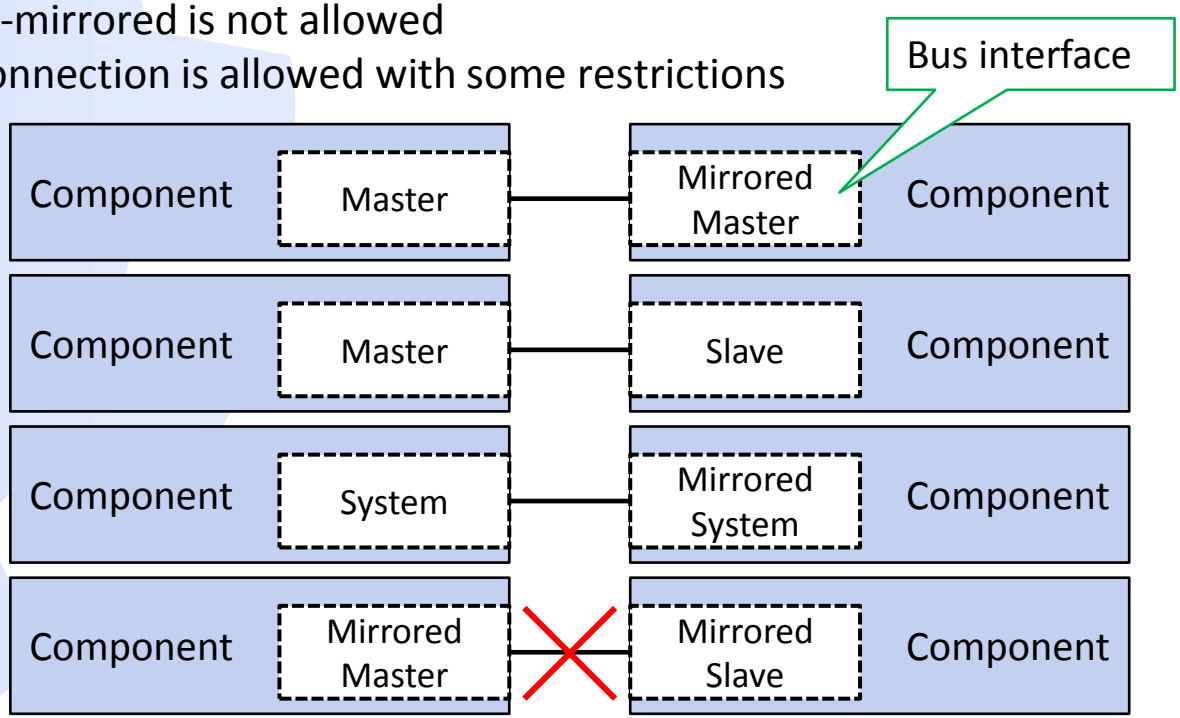
About connections

- Component's external interfaces are **ports** or **bus interfaces**.
- Connections between ports are called **ad-hoc connections**.
- Connections between bus interfaces are called **interconnections**.
- **Bus** is a general term for all kinds of interconnection topologies
  - simple buses, crossbars, network on chip
- **Component port** is an external connection from the component, also called **physical port**.
  - It can be **wire** for implementations or **abstract** for modeling purposes.
  - A port is a single signal (one wire) or a group of signals (vector, set of wires).
  - Port direction is mandatory (in, out, inout, phantom).
  - IP-XACT does not support tri-state or multiple strength values.
- **Component bus interface** is a grouping of ports associated to an **IP-XACT bus definition**
- **Same ports can be used several times** for different bus interfaces and ad-hoc connections.



Component bus interfaces

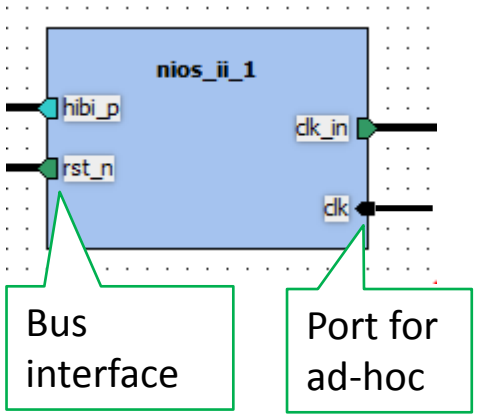
- Background: typical SoC buses have strict **master-slave** roles that are called **bus interface modes** in IP-XACT
  - Master can initiate transfers, slave can only respond
- Bus interfaces are **direct or mirrored**
  - Mirrored interface has the same signals, but directions are reversed
  - A signal that is an input on a direct is an output in the mirror interface
- IP-XACT **default is direct-mirrored** connections
- Mirrored-mirrored is not allowed
- Direct connection is allowed with some restrictions



Example: Ports

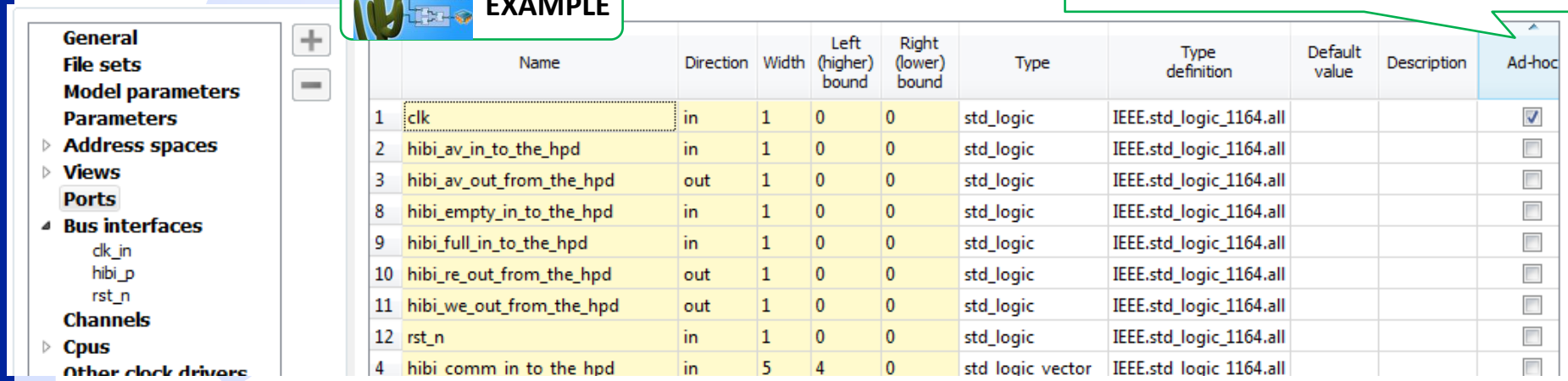
- A port can be a vector (bus) or a single wire with types and default values defined
- **Ad-hoc connections are NOT recommended**, since connection validation is poorer
- Use **bus interface** also for single-wire ports

IP-XACT component instance in a design



If checked, this port is visible for ad-hoc connections in *all instances* of this component

EXAMPLE

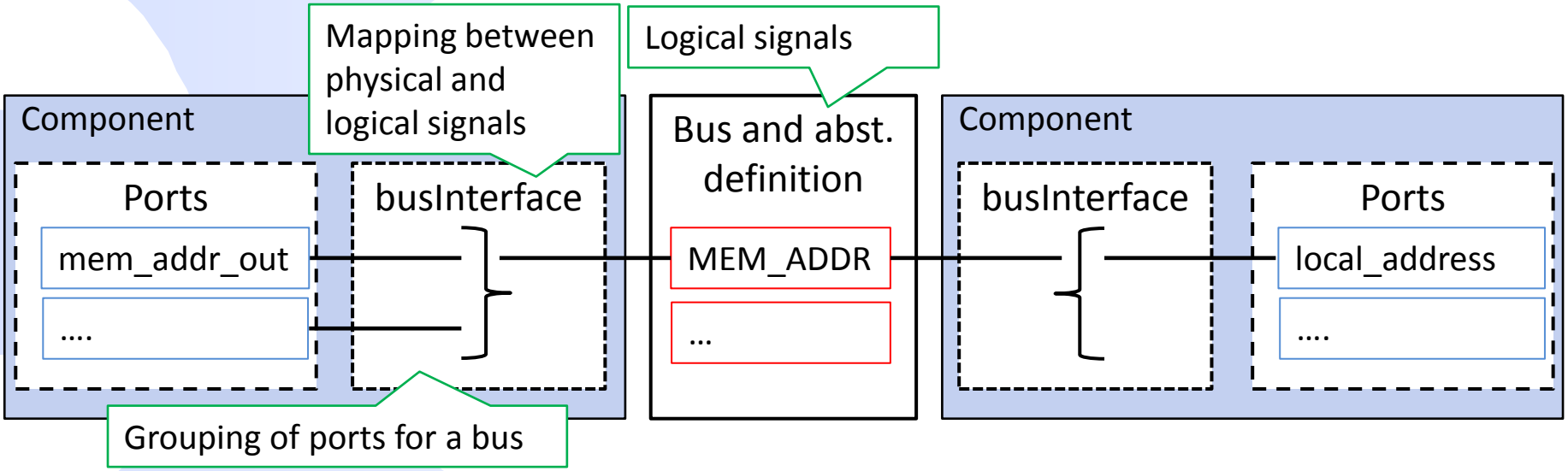


- General
- File sets
- Model parameters
- Parameters
- Address spaces
- Views
- Ports
- Bus interfaces
  - clk_in
  - hibi_p
  - rst_n
- Channels
- Cpus
- Other clock drivers

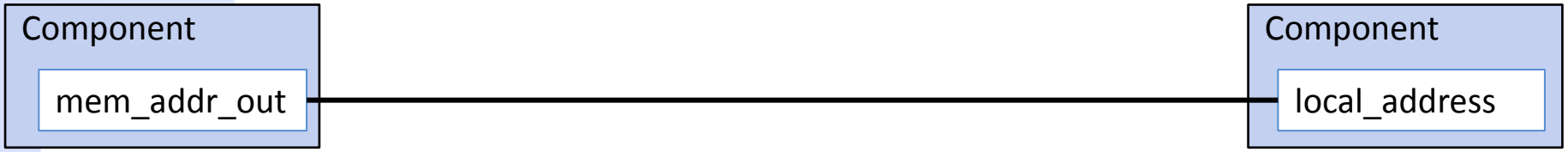
	Name	Direction	Width	Left (higher) bound	Right (lower) bound	Type	Type definition	Default value	Description	Ad-hoc
1	clk	in	1	0	0	std_logic	IEEE.std_logic_1164.all			☑
2	hibi_av_in_to_the_hpd	in	1	0	0	std_logic	IEEE.std_logic_1164.all			☐
3	hibi_av_out_from_the_hpd	out	1	0	0	std_logic	IEEE.std_logic_1164.all			☐
8	hibi_empty_in_to_the_hpd	in	1	0	0	std_logic	IEEE.std_logic_1164.all			☐
9	hibi_full_in_to_the_hpd	in	1	0	0	std_logic	IEEE.std_logic_1164.all			☐
10	hibi_re_out_from_the_hpd	out	1	0	0	std_logic	IEEE.std_logic_1164.all			☐
11	hibi_we_out_from_the_hpd	out	1	0	0	std_logic	IEEE.std_logic_1164.all			☐
12	rst_n	in	1	0	0	std_logic	IEEE.std_logic_1164.all			☐
4	hibi_comm_in_to_the_hpd	in	5	4	0	std_logic_vector	IEEE.std_logic_1164.all			☐

About connection abstraction

- IP-XACT uses **abstraction to connect two components together**
- **Bus definition** and **abstraction definition define logical signals** for a bus
- Ports (physical signals) are mapped to logical in components bus interface

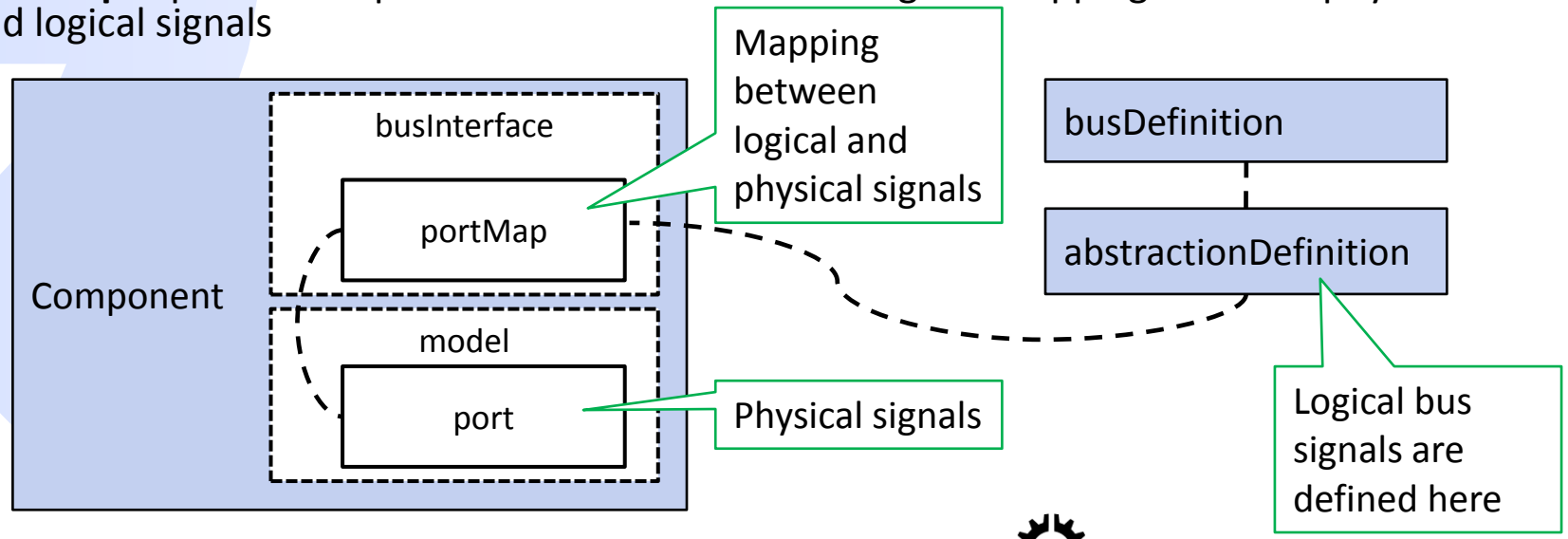


- When the design is implemented in RTL (VHDL), the final result is this:



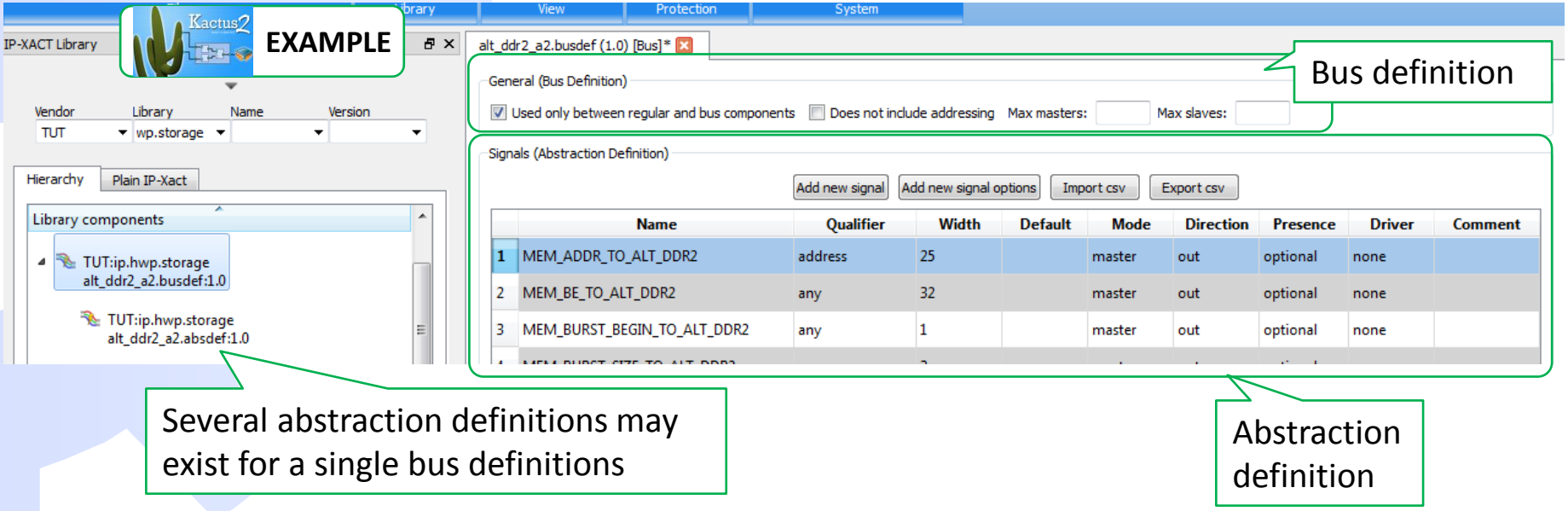
IP-XACT Bus and Abstraction definitions

- **IP-XACT bus definition** specifies general bus properties like
  - Maximum number of masters and slaves allowed
  - Bus is addressable (it is possible to see memory locations through it)
  - What kind of connections are allowed (mirrored-direct, direct-direct)
- **IP-XACT Abstraction definition** is optional refinement object always used together with bus definition. It defines **logical bus signals** and constraints related to them, like
  - bus width
  - direction of logical signals
  - presence of signal with respect to some condition (some with master, some with slave)
- Several abstraction definitions may exist for one bus definition
- **Port map** is part of component's bus interface defining the mapping between physical and logical signals



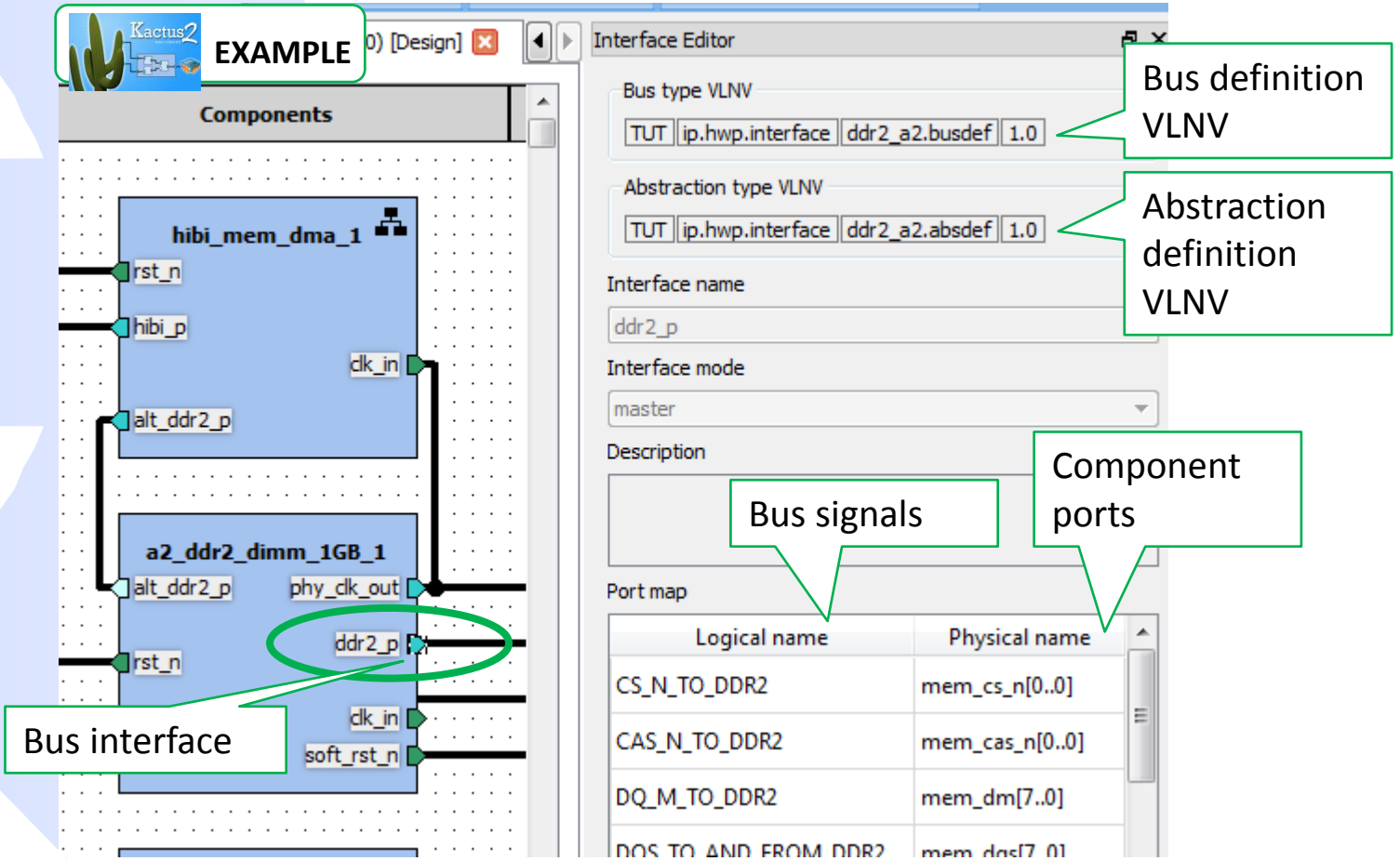
Example: BusDef and AbsDef

- Kactus2 groups together Bus and Abstraction definition, but abstraction definition is not compulsory
- Abstraction definition includes qualifiers (address, data, clock, reset, any) and presence (required, optional, illegal)
- Complex signal conditions can be created
  - E.g. some signal is not allowed if the interface is master



Example: Bus interface port map

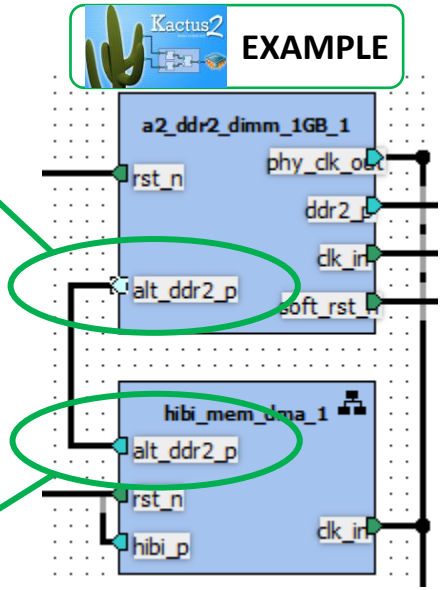
- Kactus2 interface editor displays port map **specific to a bus interface**



Example: Port Map

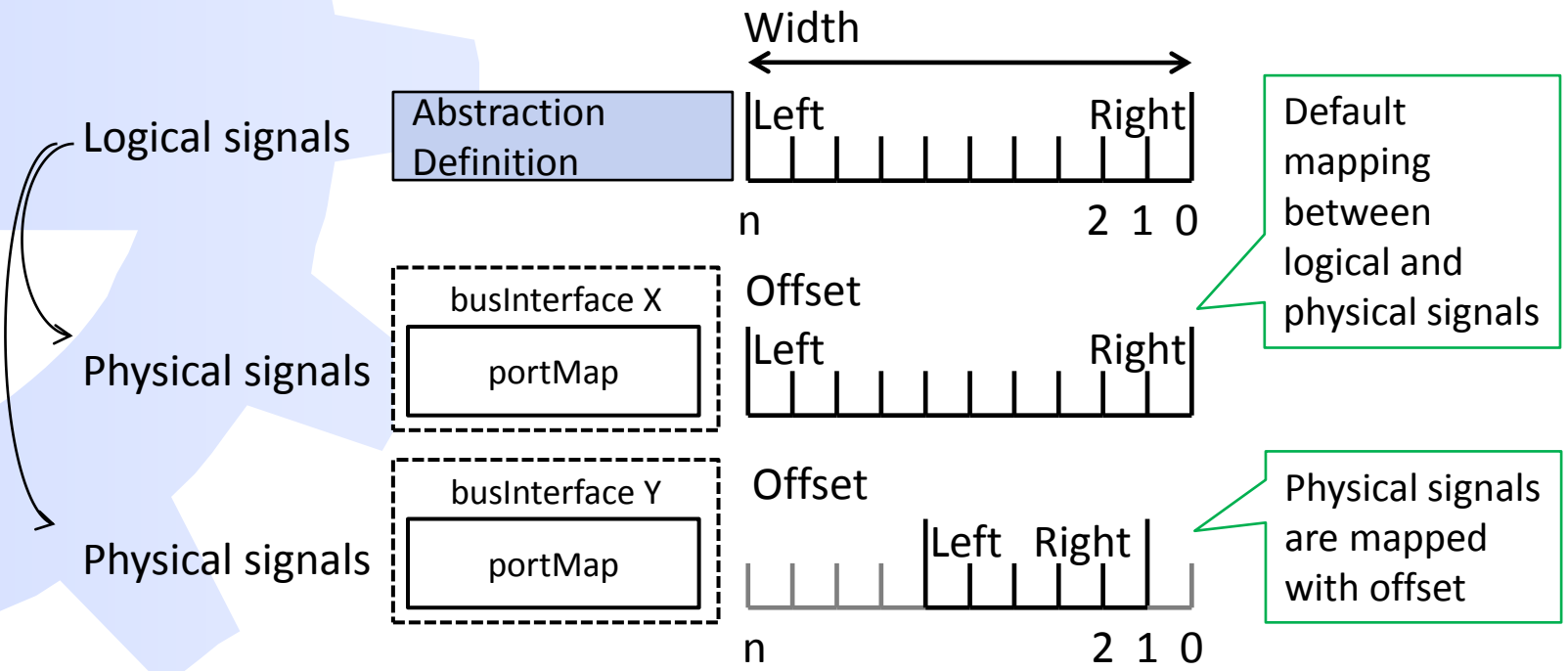
Logical name	Physical name
MEM_ADDR_TO_ALT_DDR2[24..0]	local_address[24..0]

Logical name	Physical name
MEM_ADDR_TO_ALT_DDR2[24..0]	mem_addr_out[24..0]

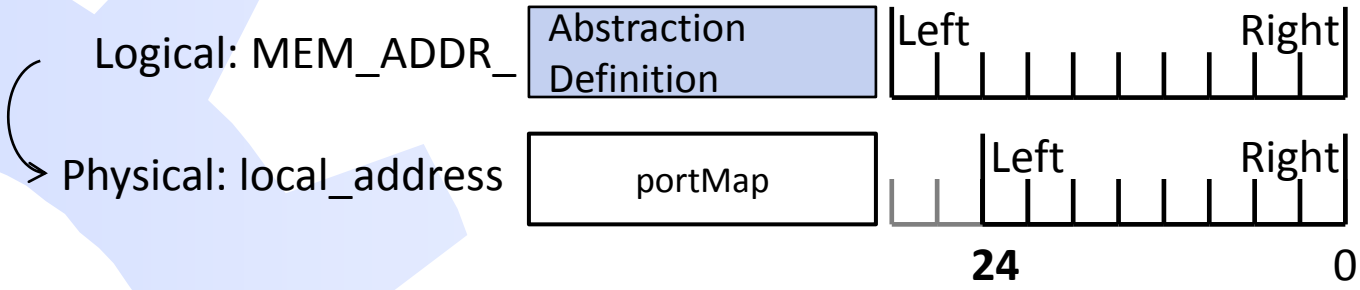


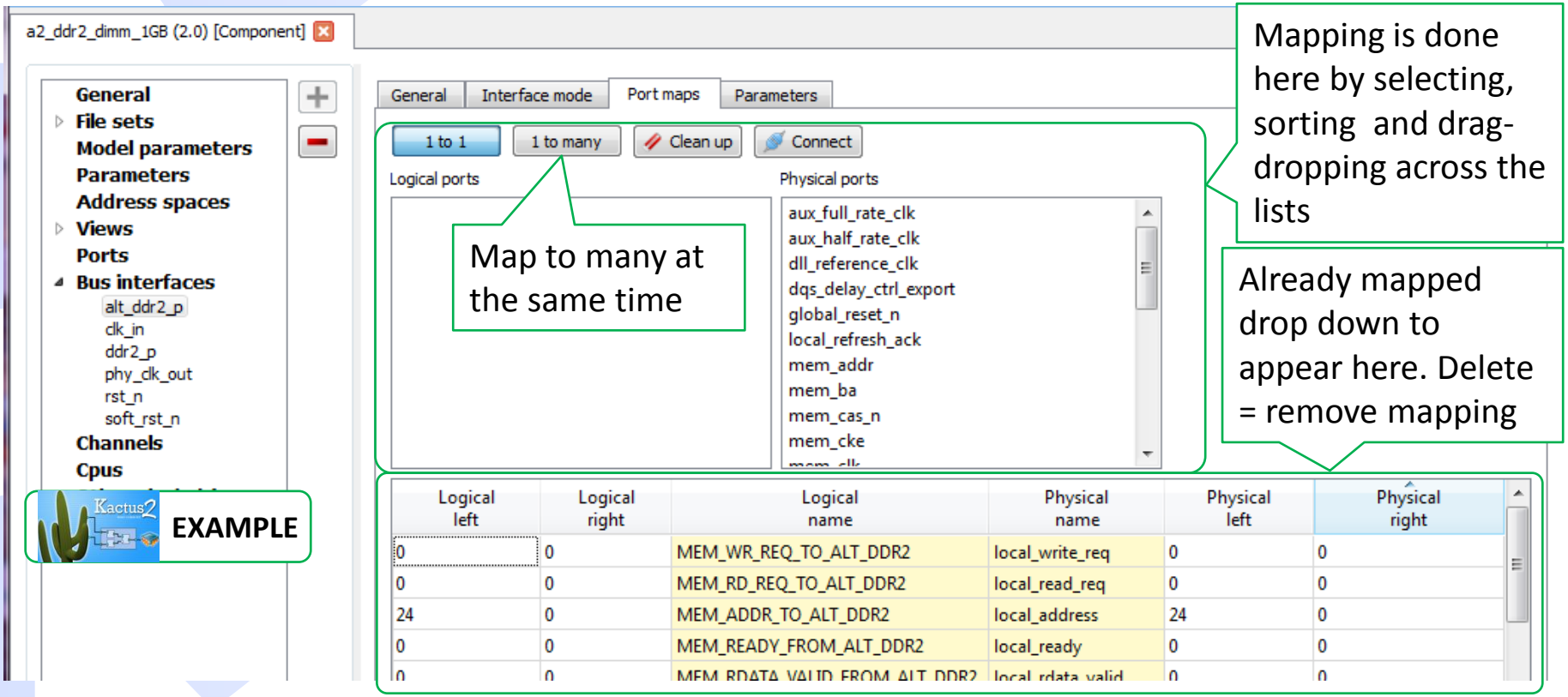
Port map with offset

- IP-XACT allows mapping with offset between physical and logical signals
- Typical uses
  - Components have different physical bus widths
  - Master and slave components have different mappings but same bus connects both
- Extreme example: one very big Abstraction definition for all possible signals. Components map only the signals they need



Example: port map with offset



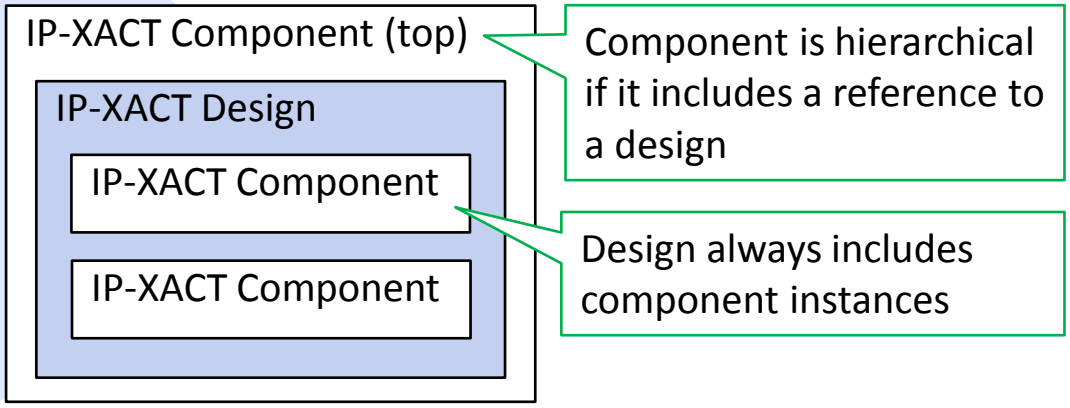


IP-XACT / IEEE1685

Design Hierarchy

Design hierarchy

- IP-XACT **design never refer to other designs**
- A design refers to components that are instantiated into design
- The design itself can be wrapped inside a top level component in order to use it as a subdesign
- References always go **downwards in hierarchy**
  - IP-XACT **component can refer to** a lower hierarchy IP-XACT **design**
  - IP-XACT **design cannot refer to** any **top level component**

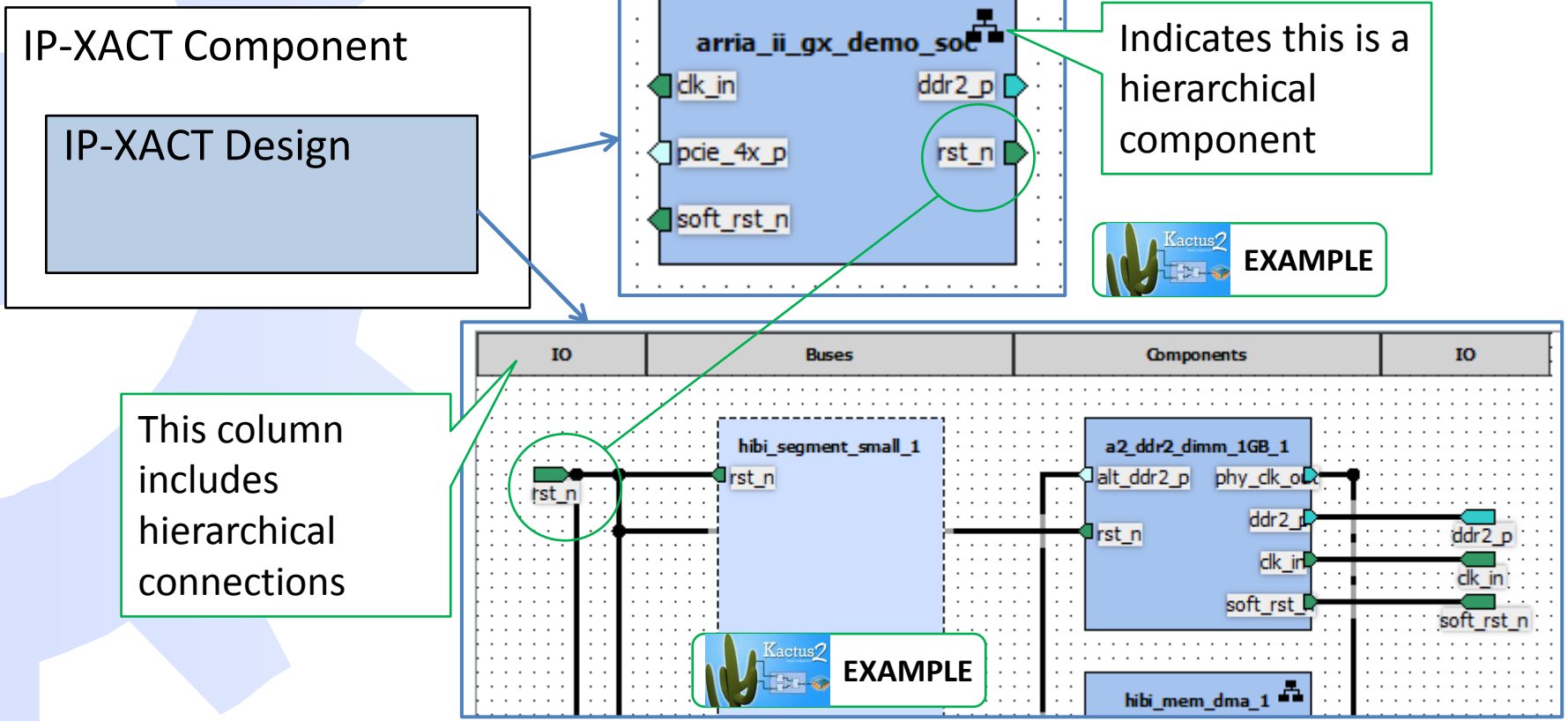


Example: Hierarchy

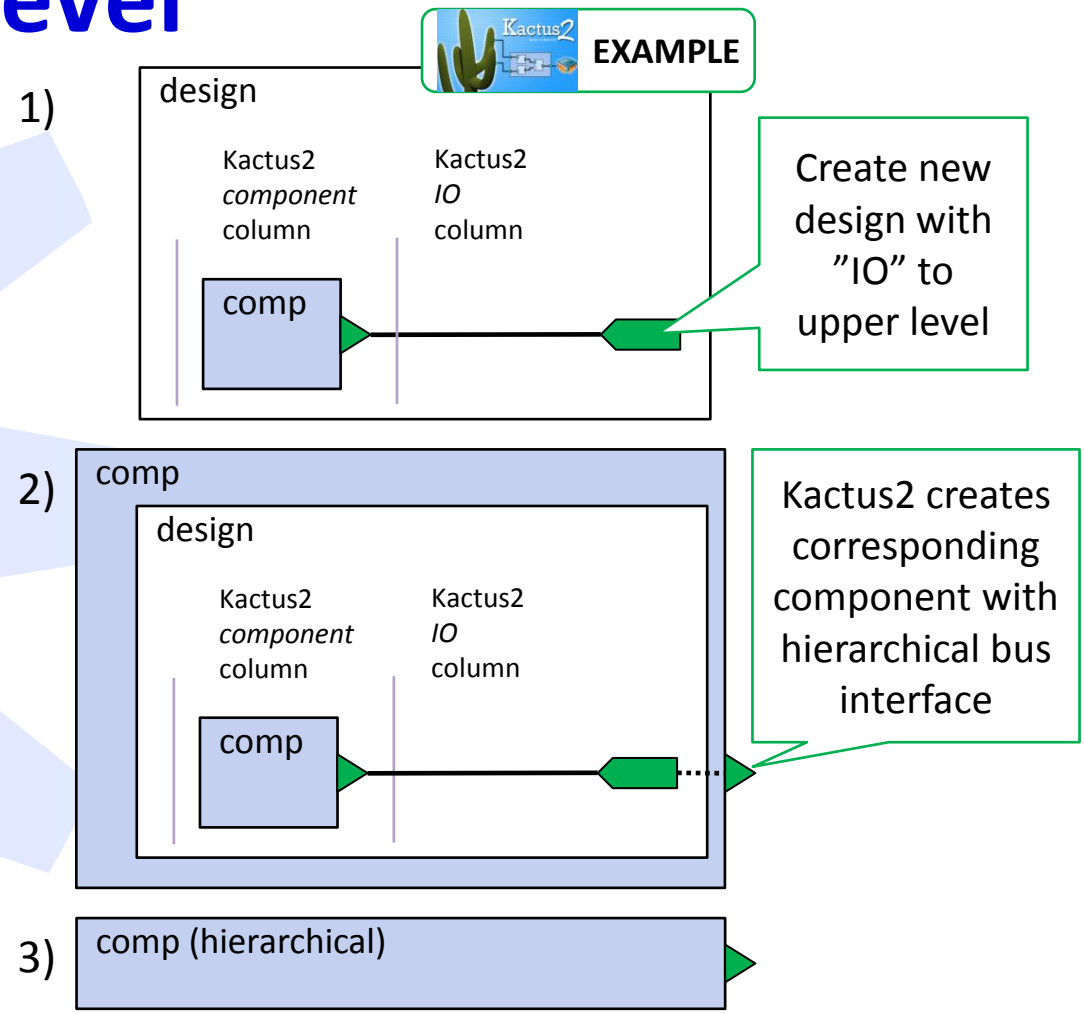
IP-XACT Component: arria_II_gx_demo_soc

└ IP-XACT Design: arria_II_gx_demo_soc

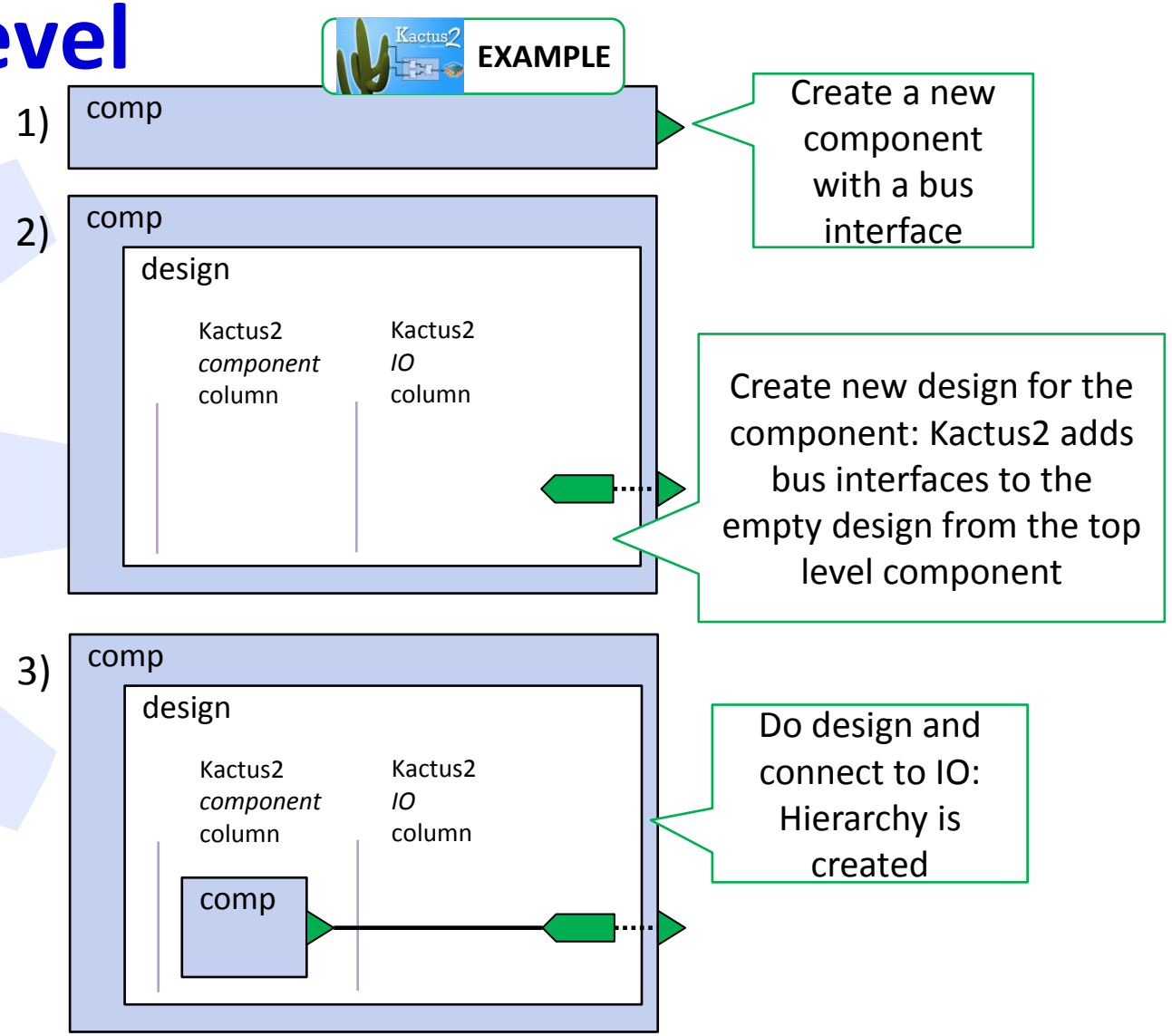
└ IP-XACT Component (instance): hibi_segment_small_1



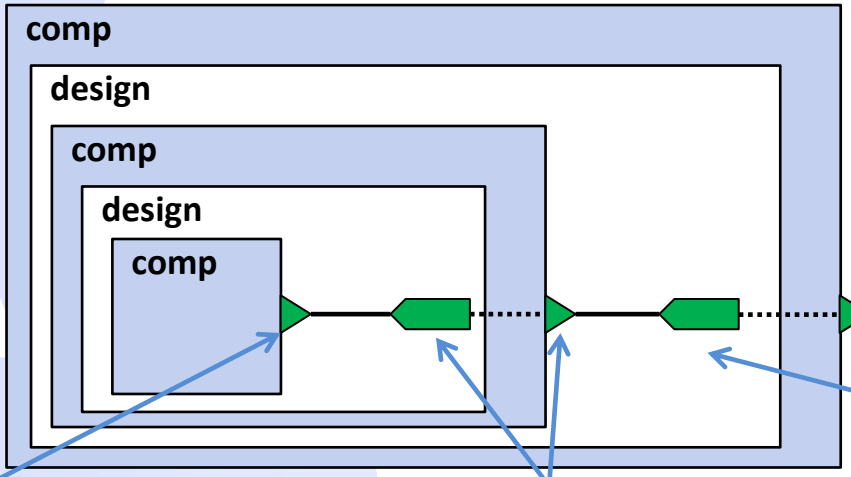
Create bus hierarchy from lower to upper level



Create bus hierarchy from top to lower level



A detailed hierarchy example



pcie_4x_if.busdef

busInterface: pcie_4x_p
busType: pcie_4x.busdef

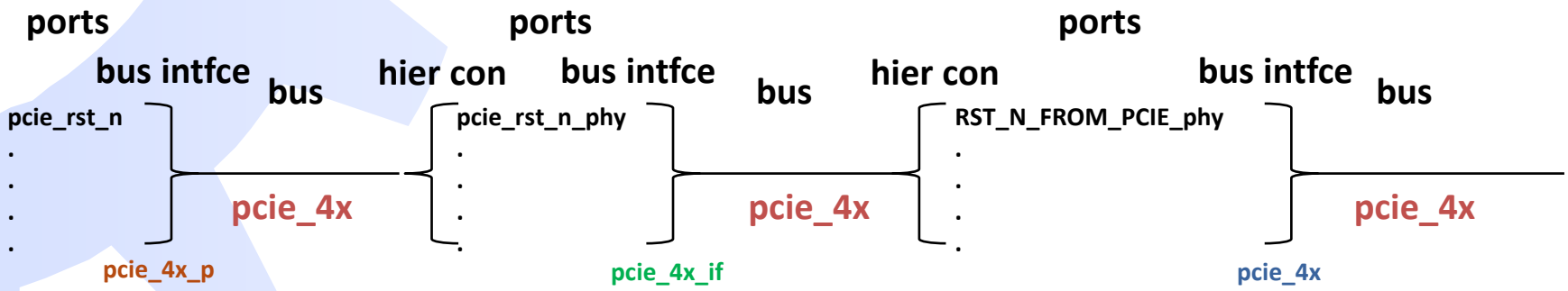
portMap
logicalPort: RST_N_FROM_PCIE
physicalPort: pcie_rst_n

busInterface: pcie_4x_if
busType: pcie_4x.busdef

portMap
logicalPort: RST_N_FROM_PCIE
physicalPort: pcie_rst_n_phy

busInterface: pcie_4x
busType: pcie_4x.busdef

portMap
logicalPort: RST_N_FROM_PCIE
physicalPort: RST_N_FROM_PCIE_phy



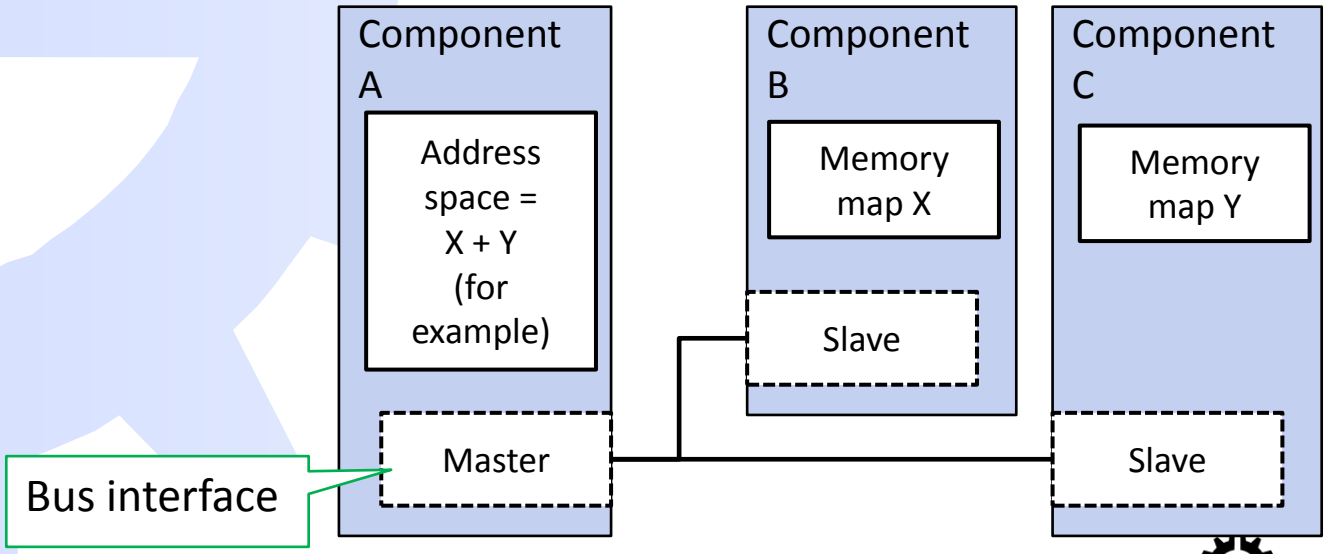
reference direction

IP-XACT / IEEE1685

Addressing and bus components

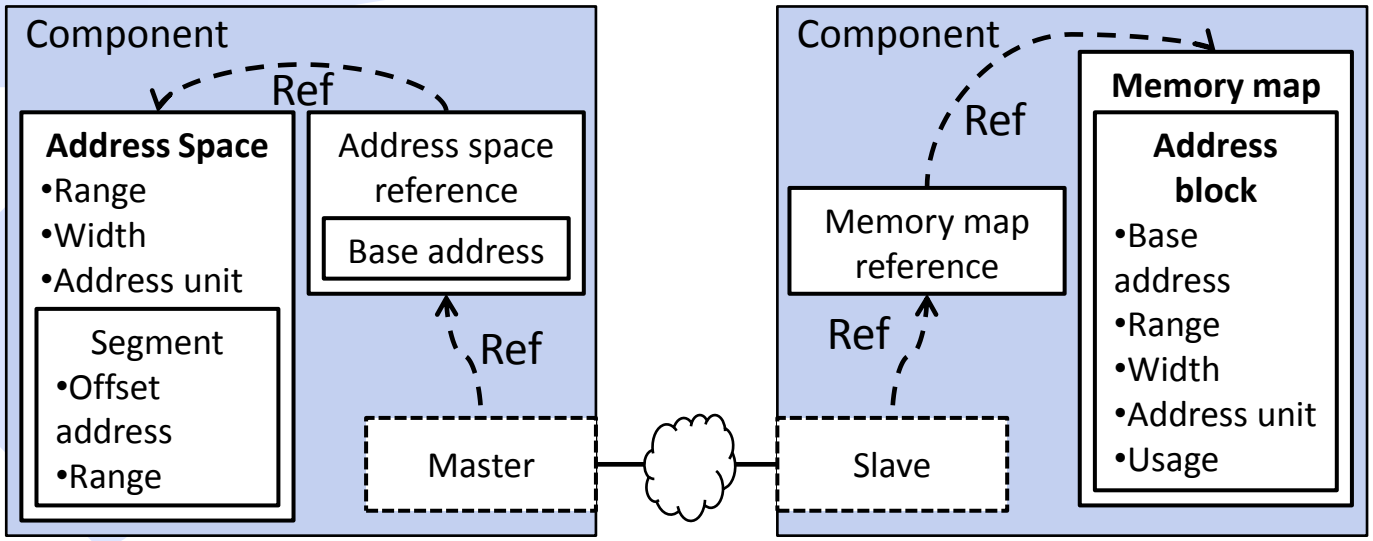
IP-XACT addressing

- **Address space** is defined for a **component's master interface.** It is programmer's view **looking out** from the master
  - These addressess can be remote (in other component) or local (in this component)
  - **Generators** can create this based on connected memory maps
- **Memory map** is defined for **slave interface** to specify registers, memory and IO accessible through this interface
  - These addresses are physically located in this component
- Note: Address space and memory map can exist without any bus interfaces



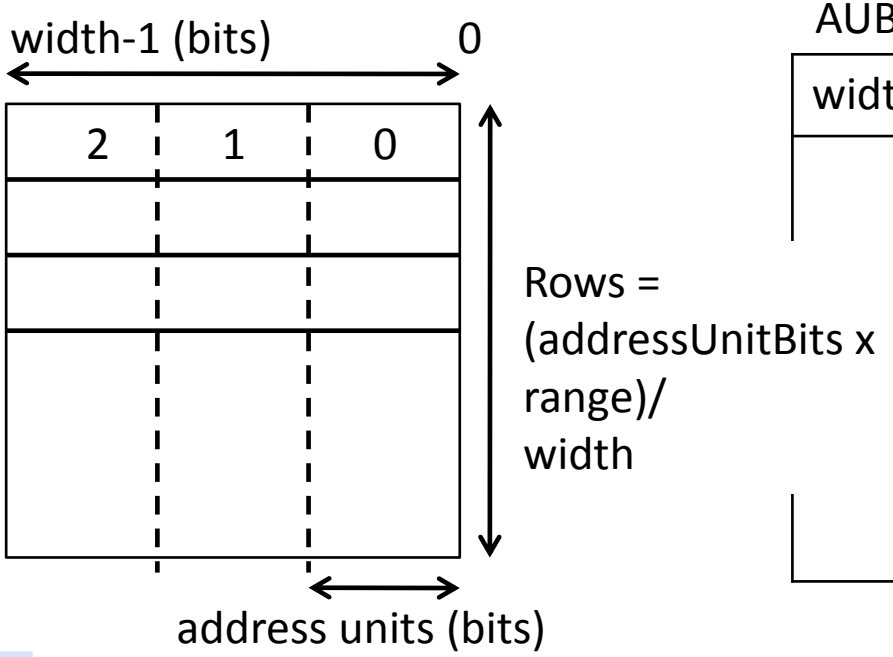
IP-XACT addressing details

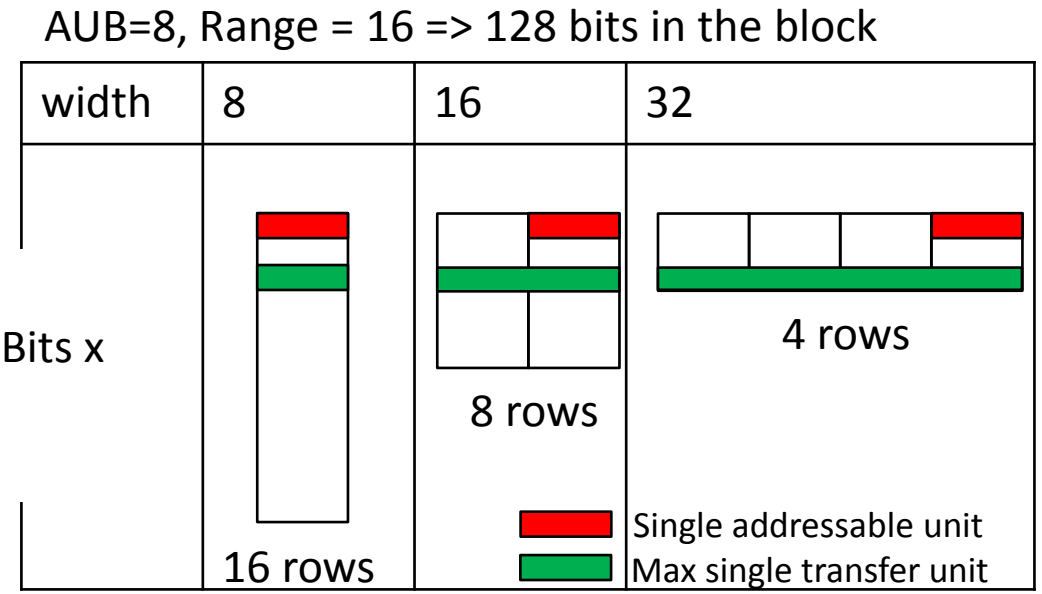
- Same address space can be associated to several master interfaces in a component
- Several address spaces can exist
- Address space may contain several **segments**
- Address space purpose can be defined like “executable program image”
  - Association is done by **address space reference** that also tells the **base address**.
- **Memory map address blocks** define basic information and usage like “ROM” and “non-volatile”.
  - Advanced definitions include **banks** and address handling especially for **bridge components**.



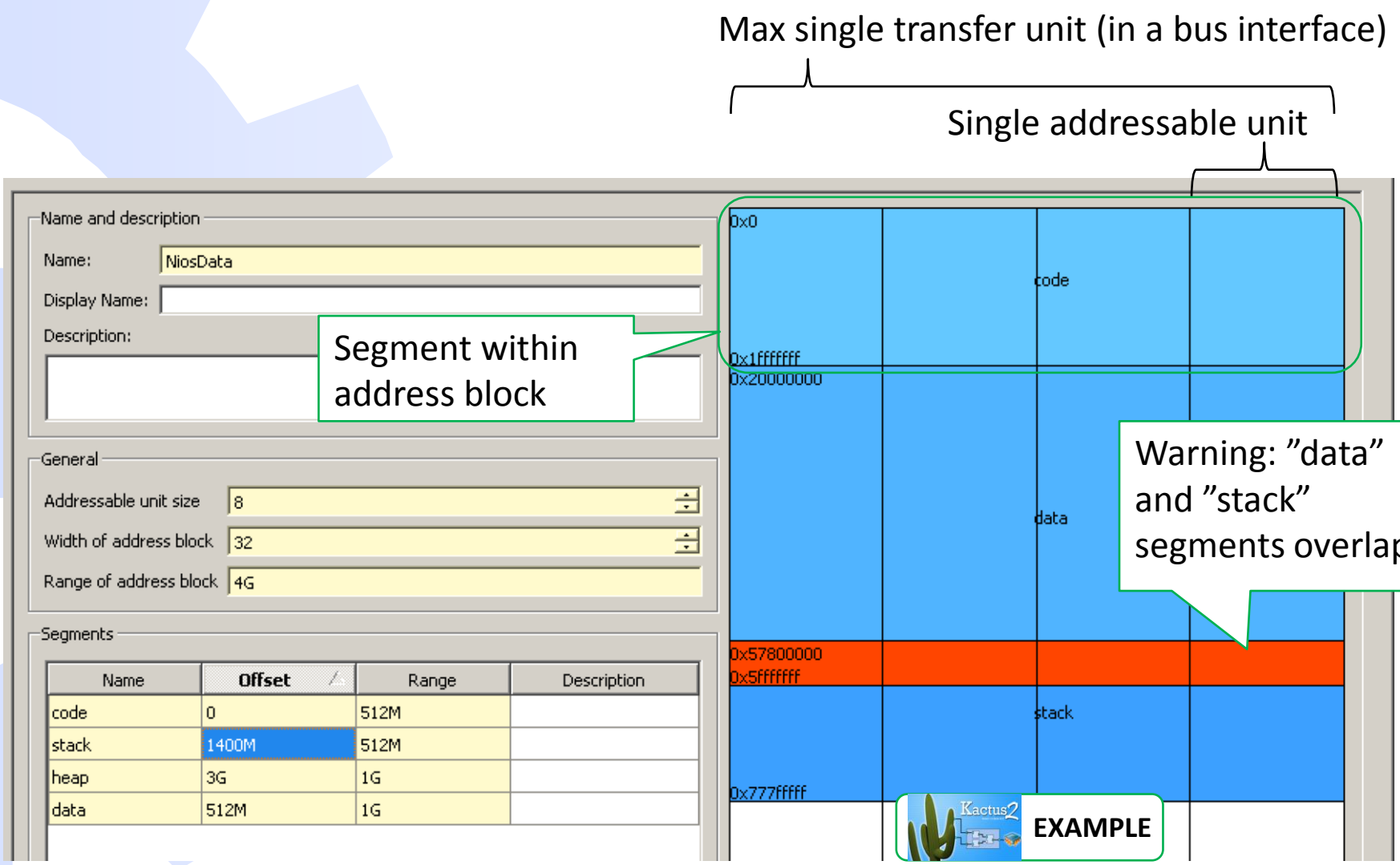
Address definitions

- **Address unit bits** tells the number of data bits in each increment in address (default is 8 bits)
- **Range** specifies the range as the **number of addressable units**
- **Width** specifies the row width. It tells the maximum single transfer size by a bus interface
- **rows x width == addressUnitBits x range == bits**



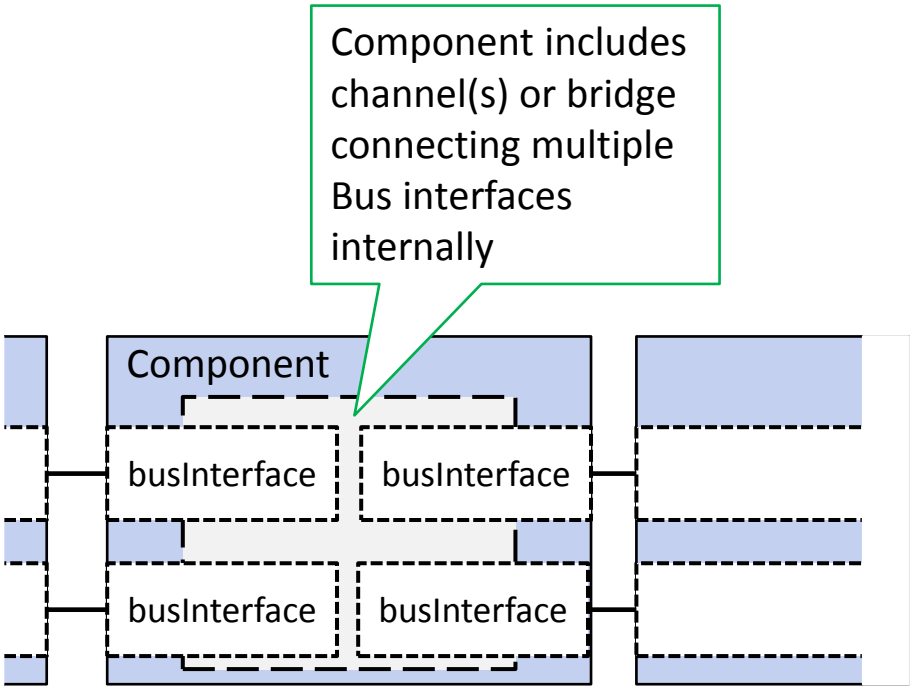


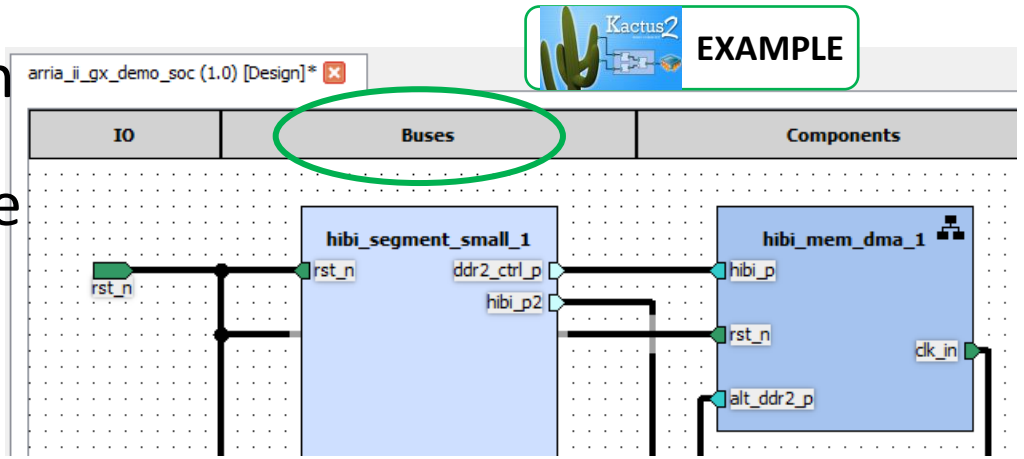
Example: Address space



Bus components

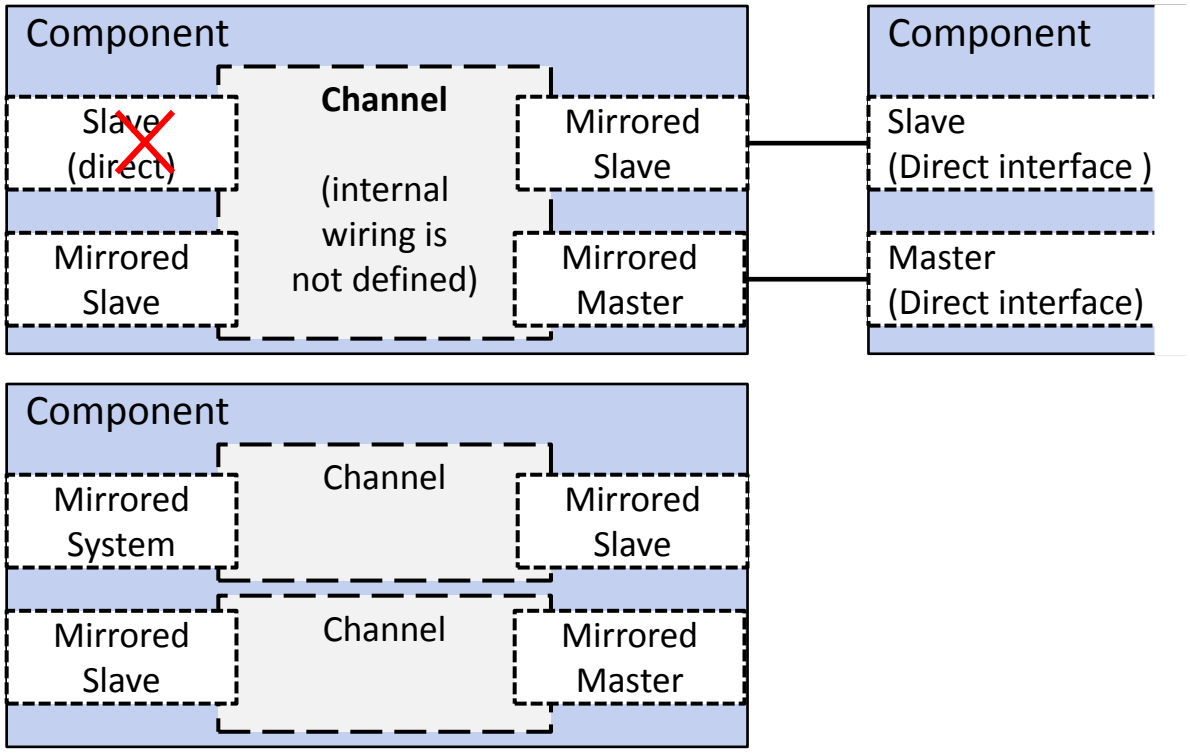
- On-chip bus or other network is implemented with IP-XACT component that
  - include several bus interfaces
  - Bus interfaces are internally connected as **channels** or **bridges**
- Kactus2 uses separation to regular and bus components that can be placed on separate columns for better readability





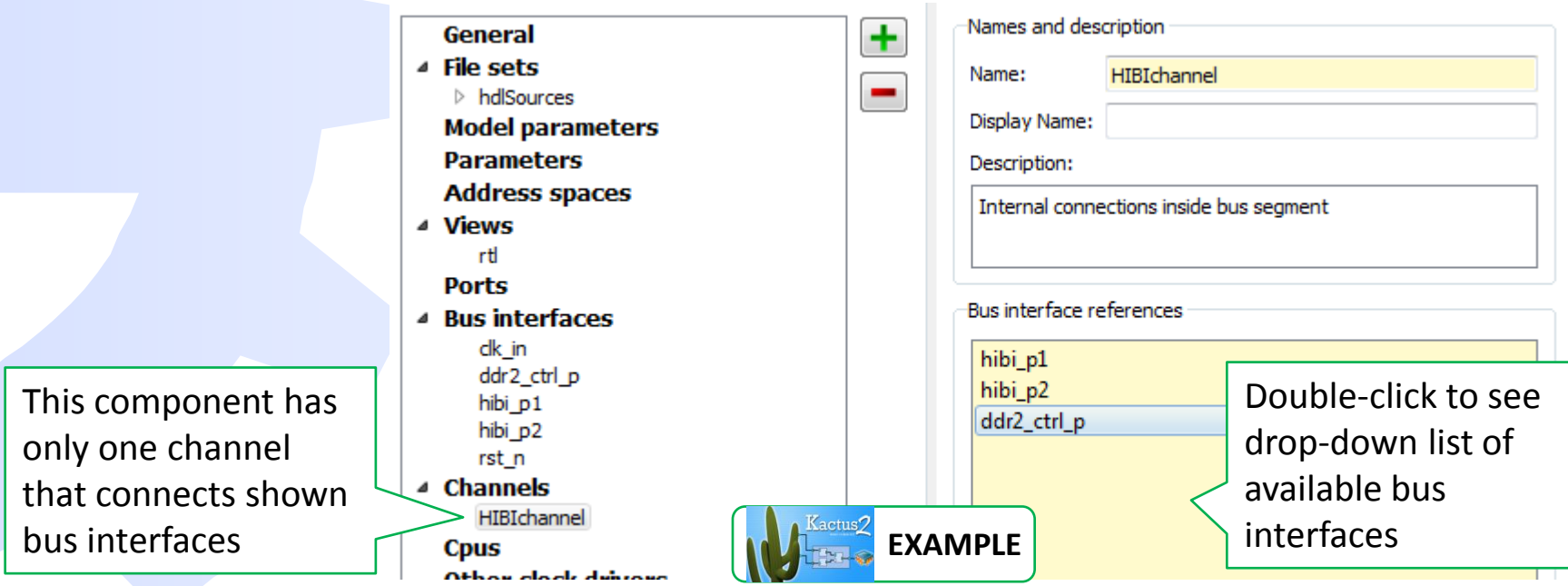
Channel

- **Channel** is often used to implement a single bus
- **Only mirrored** bus interfaces are allowed.
- Channel has **only one address space** that is same **for all mirrored master interfaces**.
- Mirrored slaves can have subset address spaces.
- A component can have several channels.



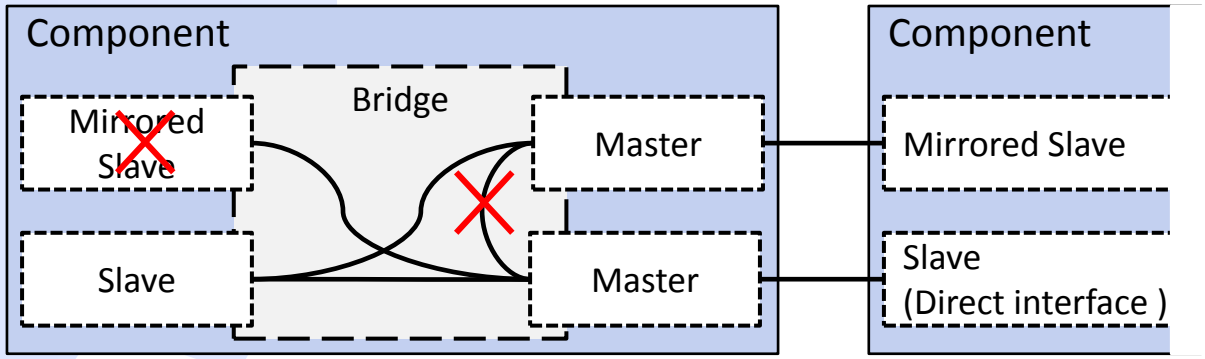
Example: Channel

- Bus interfaces can be associated to one or more channels
- Channels do not specify any explicit signal-by-signal mapping between bus interfaces, but just information which ones are logically connected



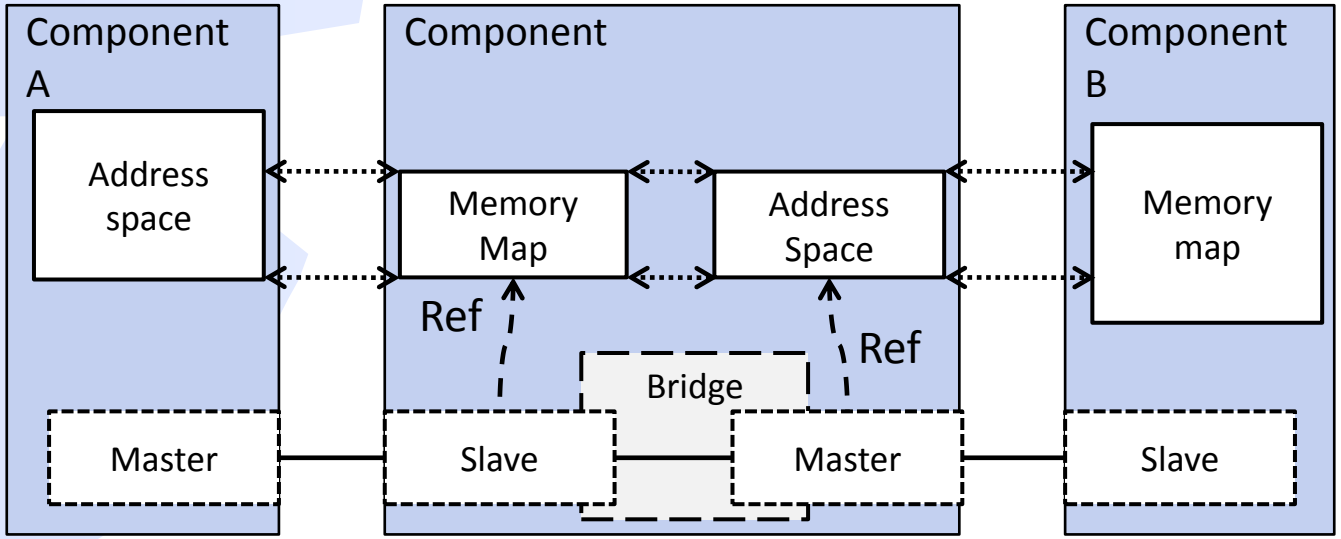
Bridge

- **Bridge** is often used to connect different buses together.
- Bridge defines **for each slave** its connections to one or more master interfaces.
  - Internal wiring is explicitly defined
- Bridge has **only direct** interfaces, and it can connect to direct or mirrored interfaces.



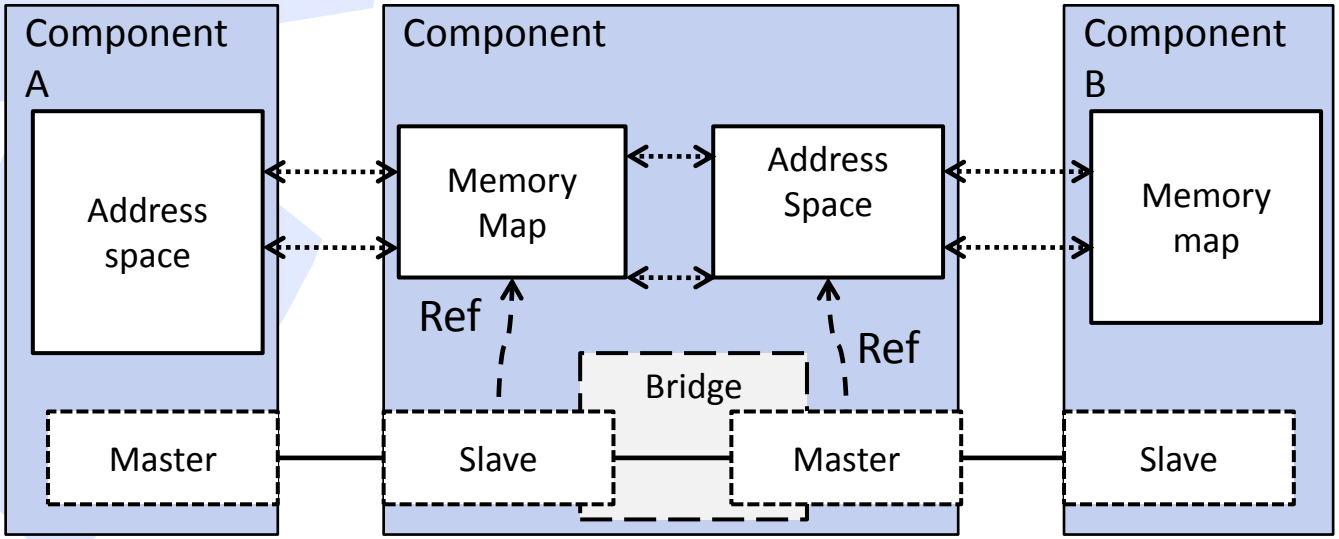
Transparent bridge addressing

- In **transparent bridge**, the address space seen from the master interface (out from bridge) is seen as such at slave interface.
- If bridge connects multiple masters to a slave, the slave memory map contains all master address spaces.
- Note that bridge master address space may not cover all of the available memory of other component (B). Component A can transparently see B's space as defined in bridge's master address space.



Opaque bridge addressing

- In **Opaque bridge,** the address space on bridge's master interface is not directly seen from the slave interface.
- In this case, the bridge makes complex mappings with offsets and base addresses. See IEEE1685 Annex H for complete presentation.

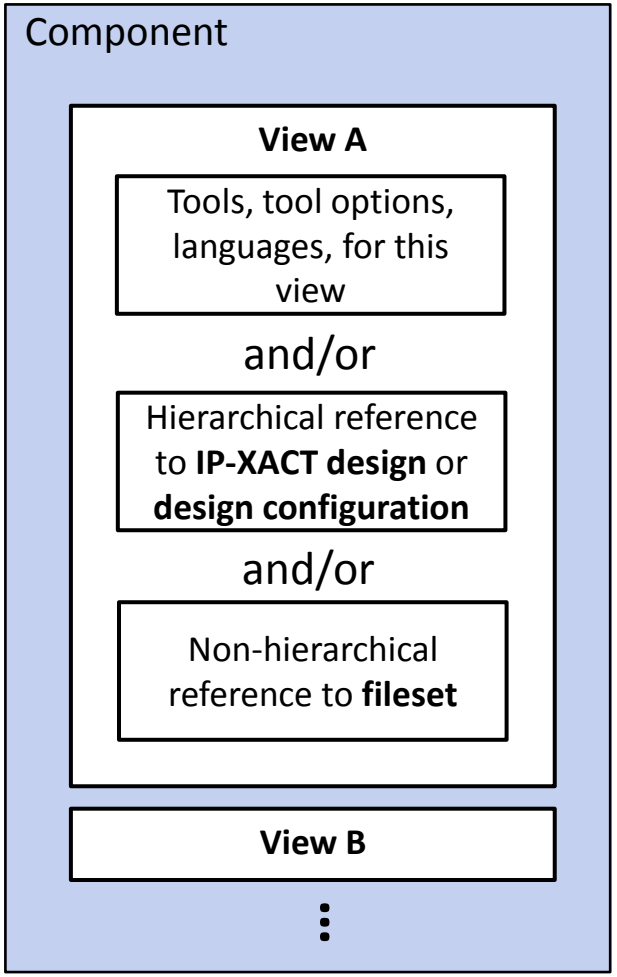


IP-XACT / IEEE1685

Design configuration

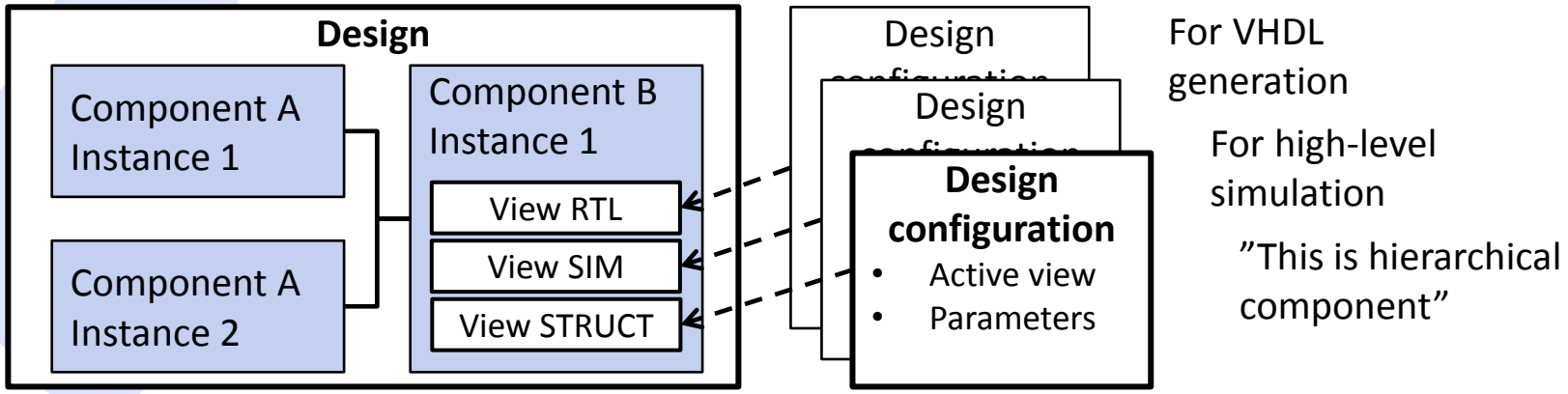
Component View

- A component can have several views that describe
  - information about implementation environment
  - design hierarchy
  - associated files
- **Views can be seen as different purposes** of the component
  - "RTL view" may describe the source hardware module/entity with its pin interface
  - "A documentation view" may define the written specification of this IP
  - "Structural view" may refer to an IP-XACT design for hierarchical designs
- IP-XACT standard allows only one active view for a component when instantiated in a design



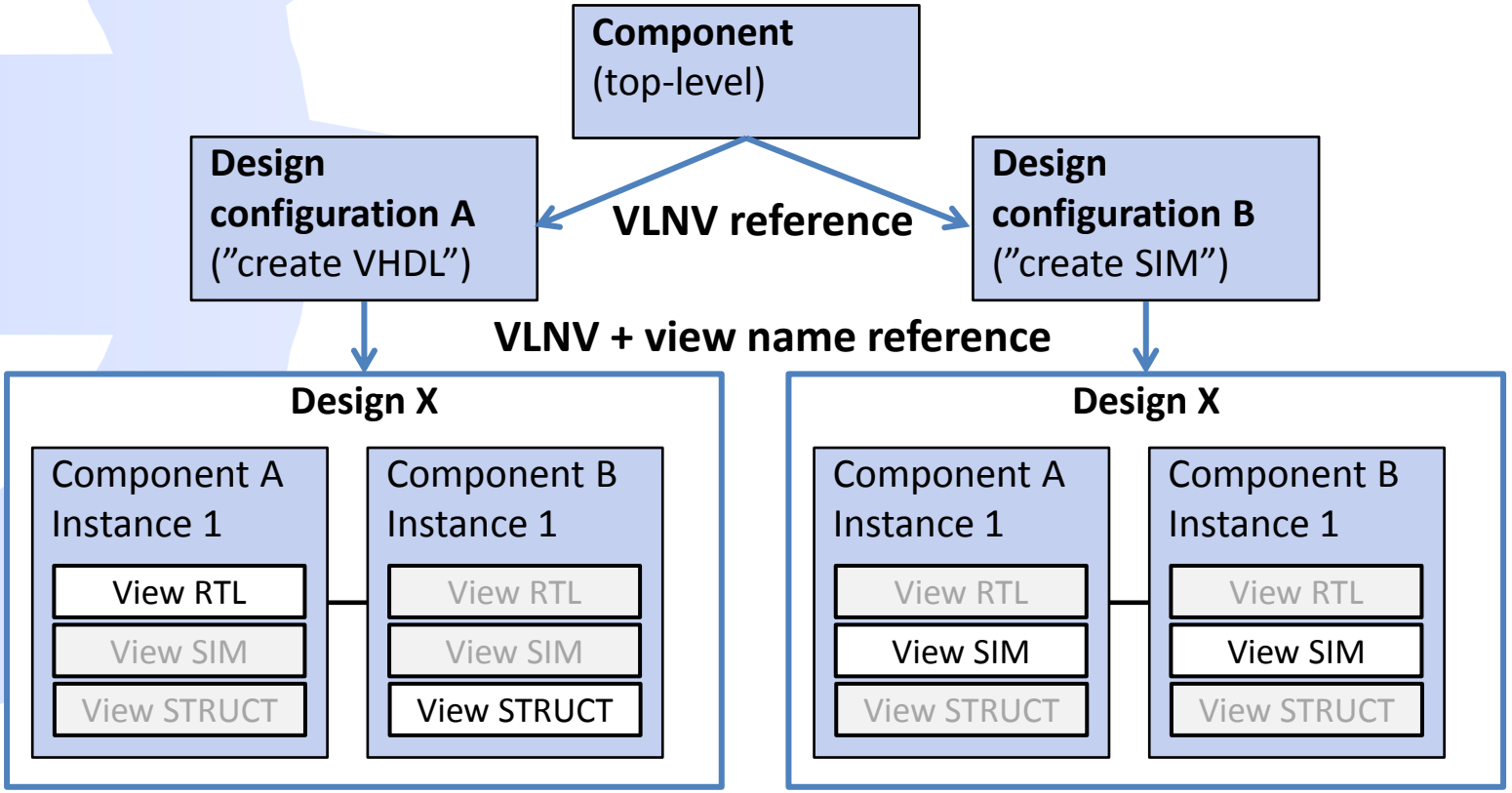
IP-XACT Design configuration

- Defines
  - **Active (current) views** for all component instances of the IP-XACT design
  - Configuration for interconnections between the same bus types (bus definition) but with different abstraction definitions. This uses **IP-XACT abstractor objects**
  - Configurable information for parameters defined in generators in **IP-XACT generator chain** object
- A single design configuration applies to a single design, but **a design may have multiple design configurations**



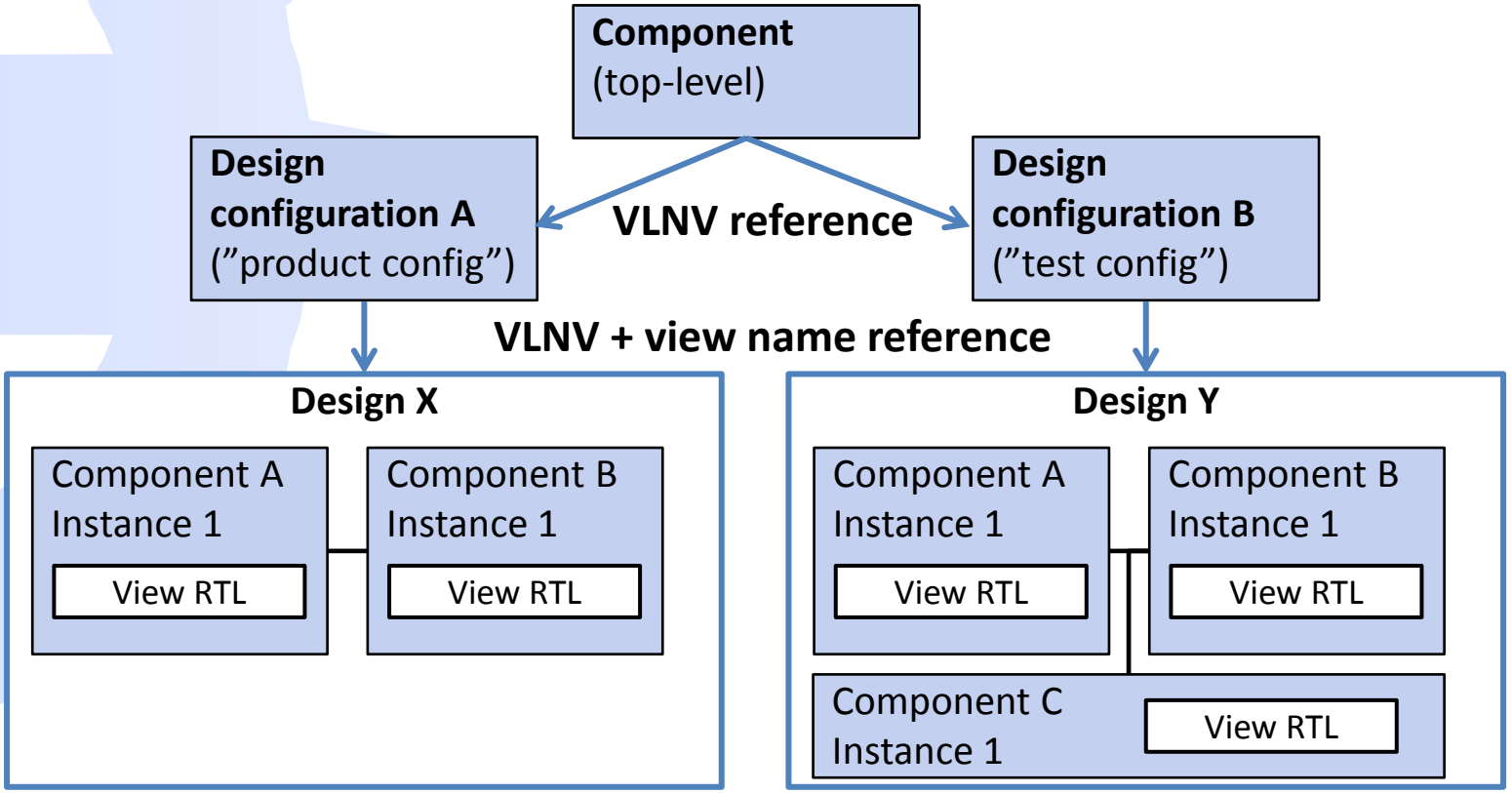
Configurations: Basic case 1

- Design configuration selects active views of component instances in IP-XACT design
- E.g. one for implementation and one for simulation purposes



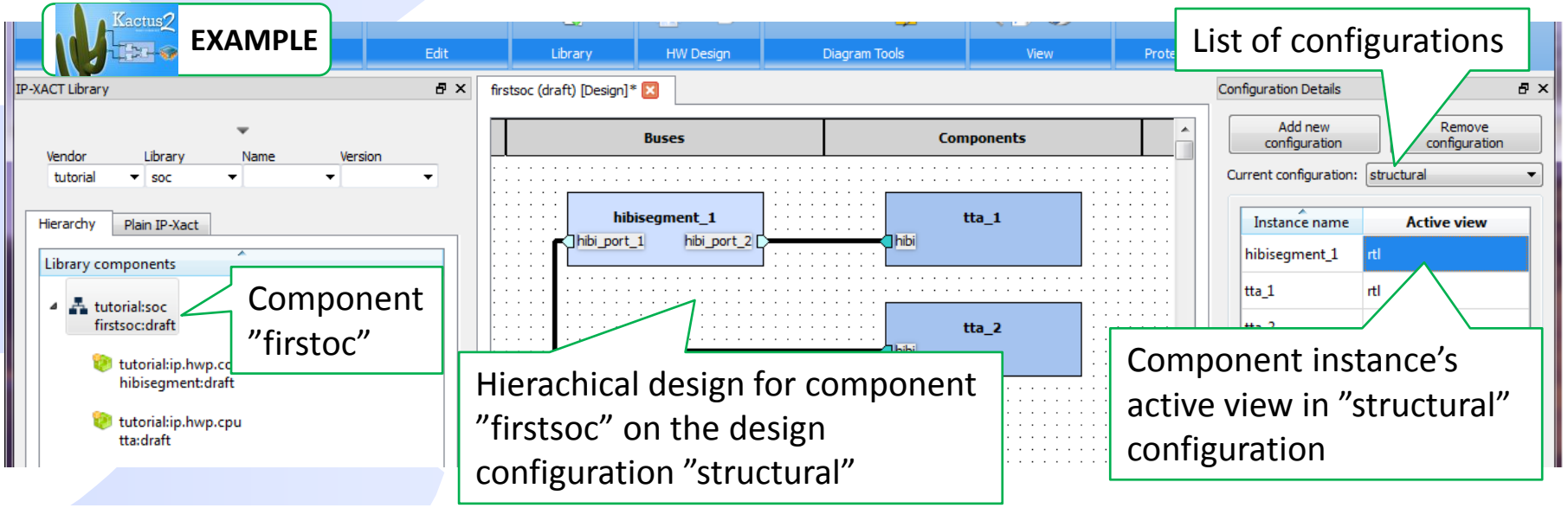
Configurations: Basic case 2

- Design configuration selects different IP-XACT designs
- E.g. two different implementations for production and testing
- Note: views can also be configured at the same time

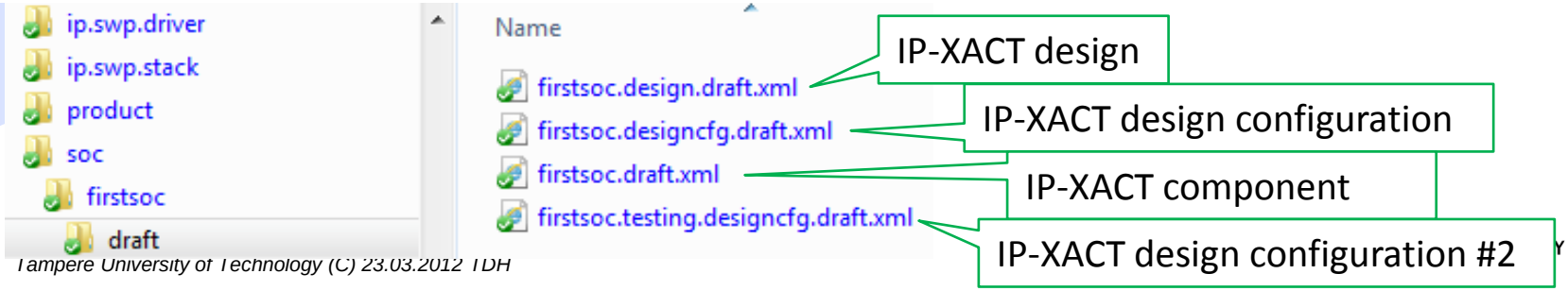


Example: Design configurations

- Kactus2 manages/lists design configurations on right hand panel
- Kactus2 creates IP-XACT component, design and design configuration objects for user
- User can add more configurations

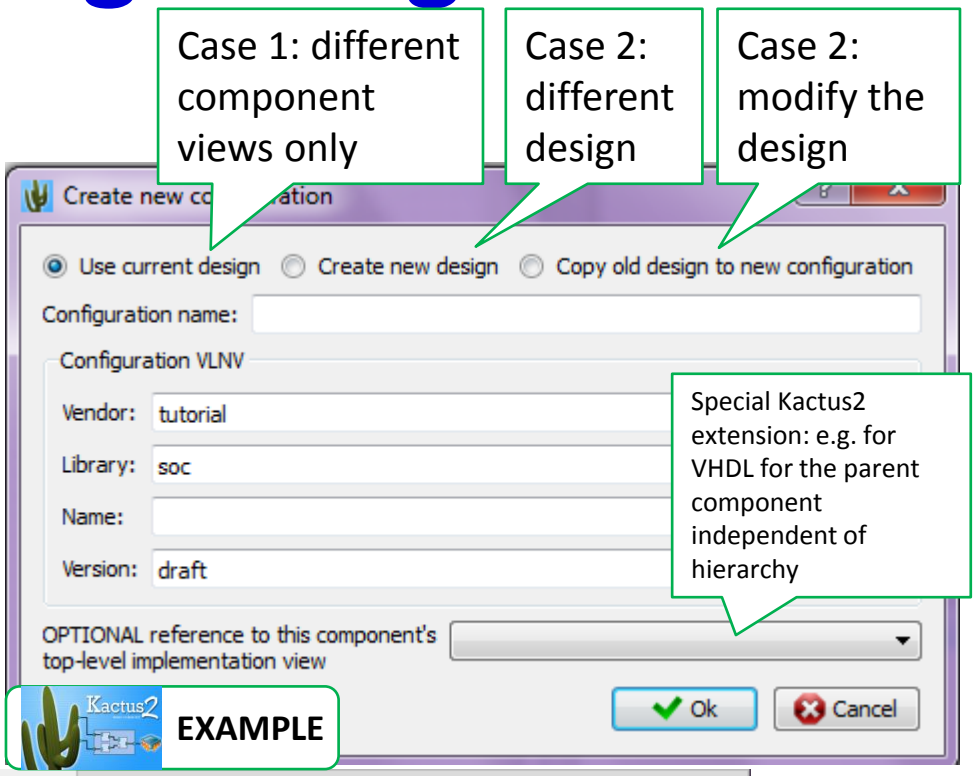


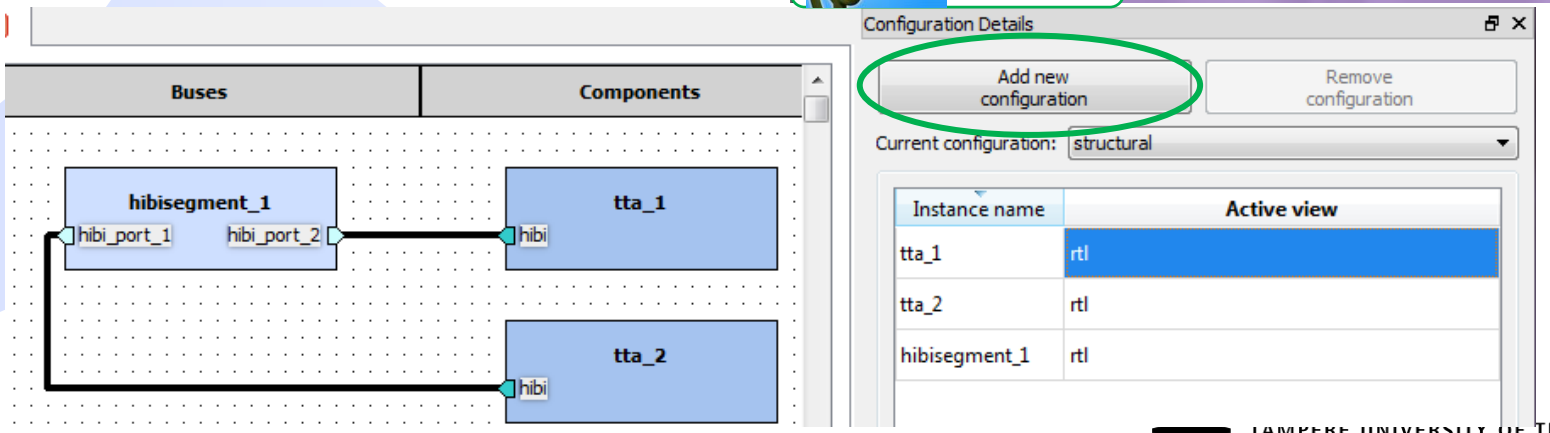
Objects on disk (note: Kactus2 takes care of XML files, do not edit manually)



Example: Creating configurations

- Example: A component have one design and one configuration that defines component instance views
- A new configuration may add
  - new set of views and/or
  - changes to the design structure
  - a totally different design
- Note: it is **not recommended to modify hierarchical connections** in a design for some configuration
  - Violates component-design hierarchy consistence
  - If needed, create a new component (copy & give a new name or version)



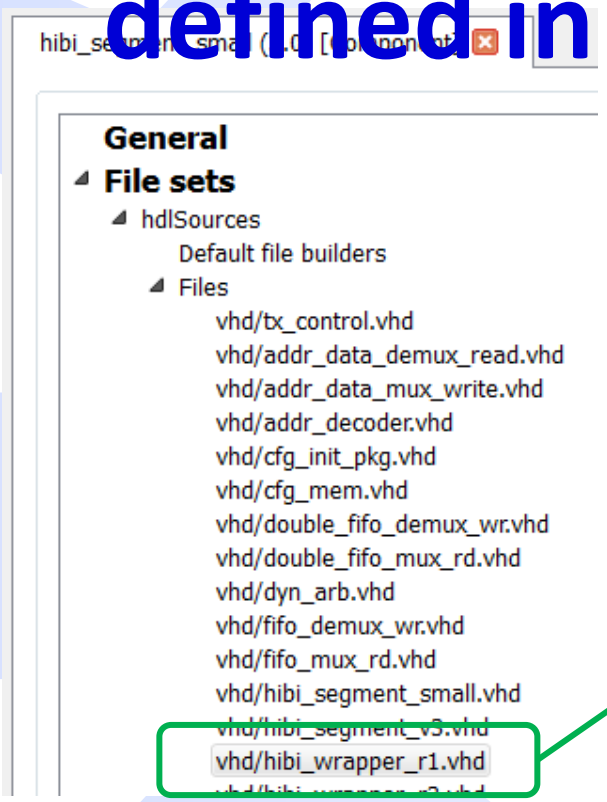


Parameters

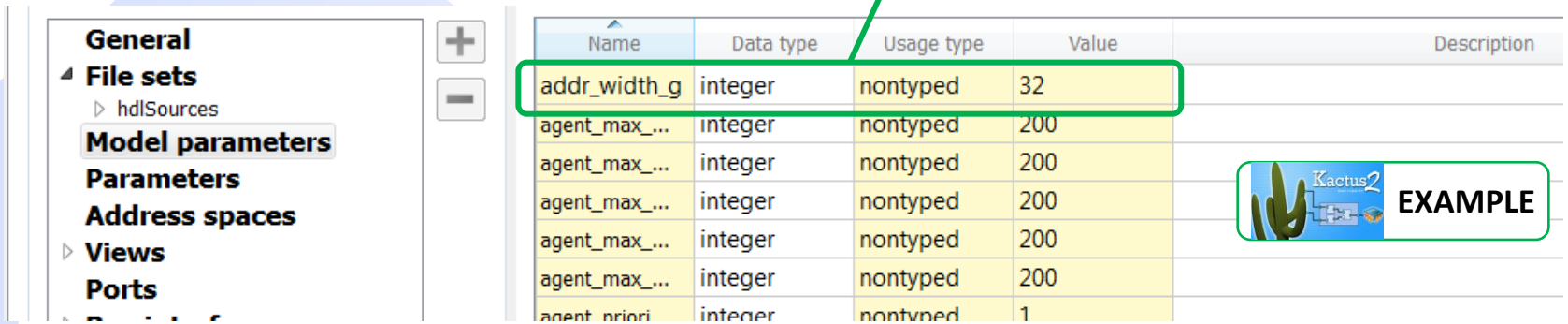
- Parameters make components configurable
- IP-XACT objects have **parameters** that can be used to configure or hold any information
  - name, value, attributes
  - Parameter scope is not defined (global/local)
  - Parameters can be of any type
- IP-XACT components may have also **model parameters**
  - specific to the implementation of the IP-XACT object
  - E.g. VHDL generics
  - <name, type, usage, value>
- Values can be omitted or given as the defaults
  - Design/instance specific values can override the defaults

Example: Model parameter defined in IP-XACT component

- Model parameters are globally defined (not specific to a source file)
- User must take care that a model parameter does not affect unintentionally if the same name is used in several source files (e.g. ”input”)

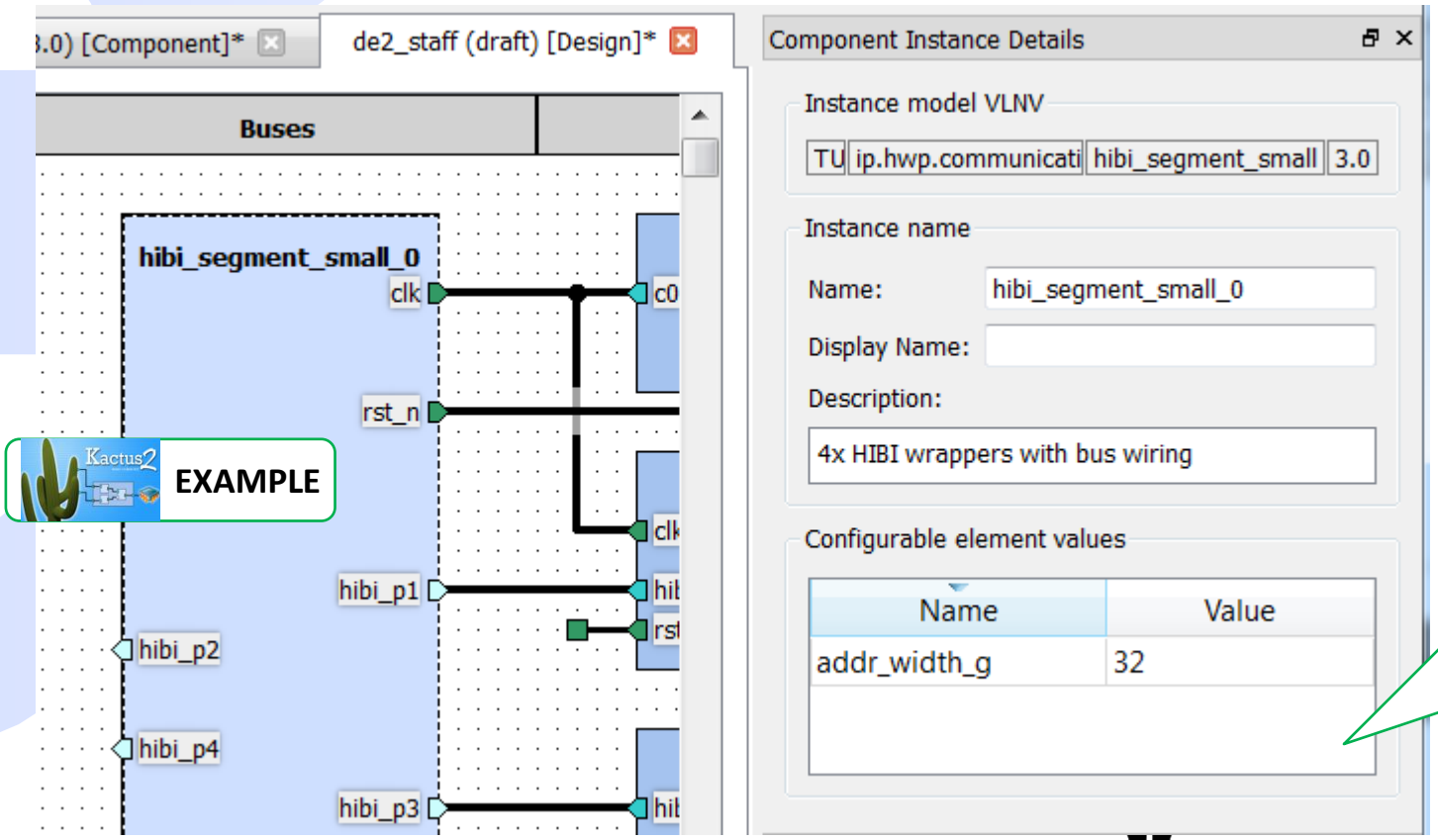



```
entity hibi_wrapper_r1 is
 generic (
  -- Structural settings.
  --  All widths are given in bits
  addr_width_g    : integer;
  ...
```



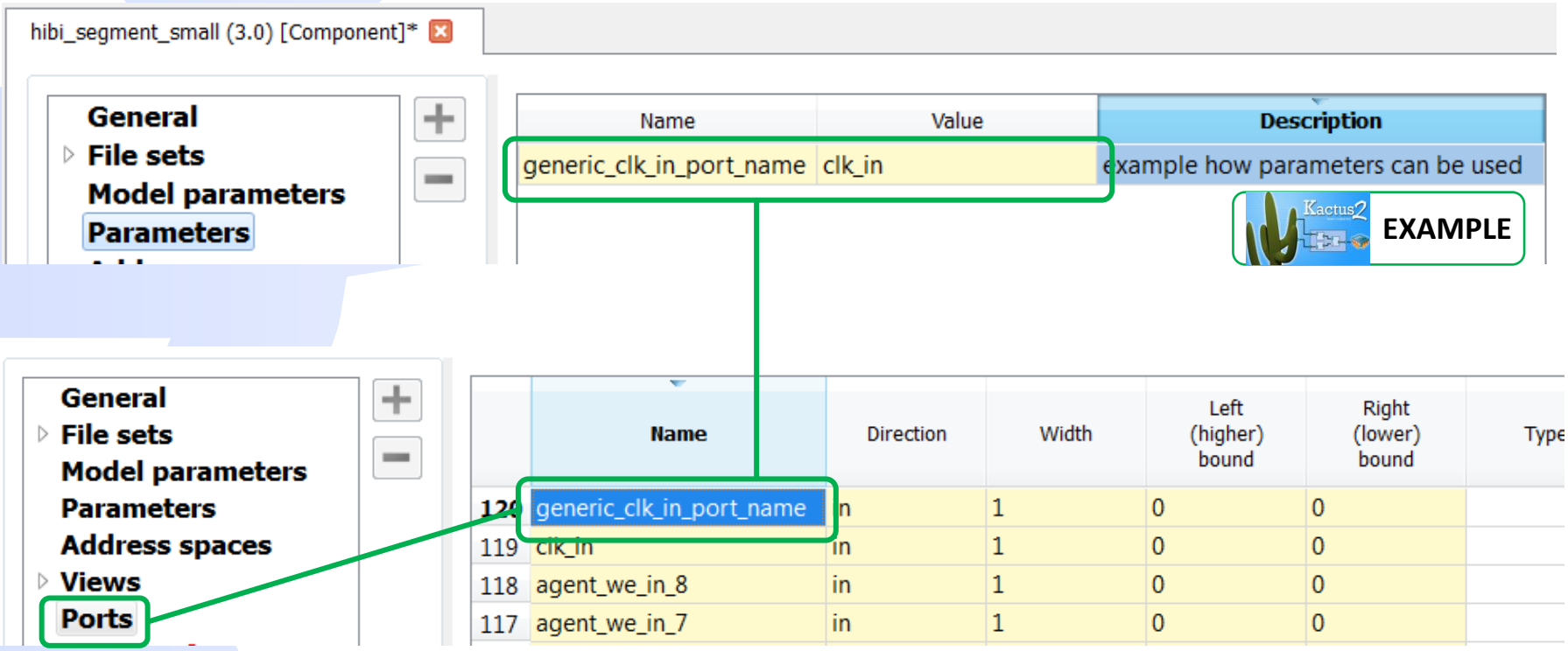
Example: Model parameter set in IP-XACT design

- Model parameters are set in IP-XACT design for component instances
  - If not set, default values are used

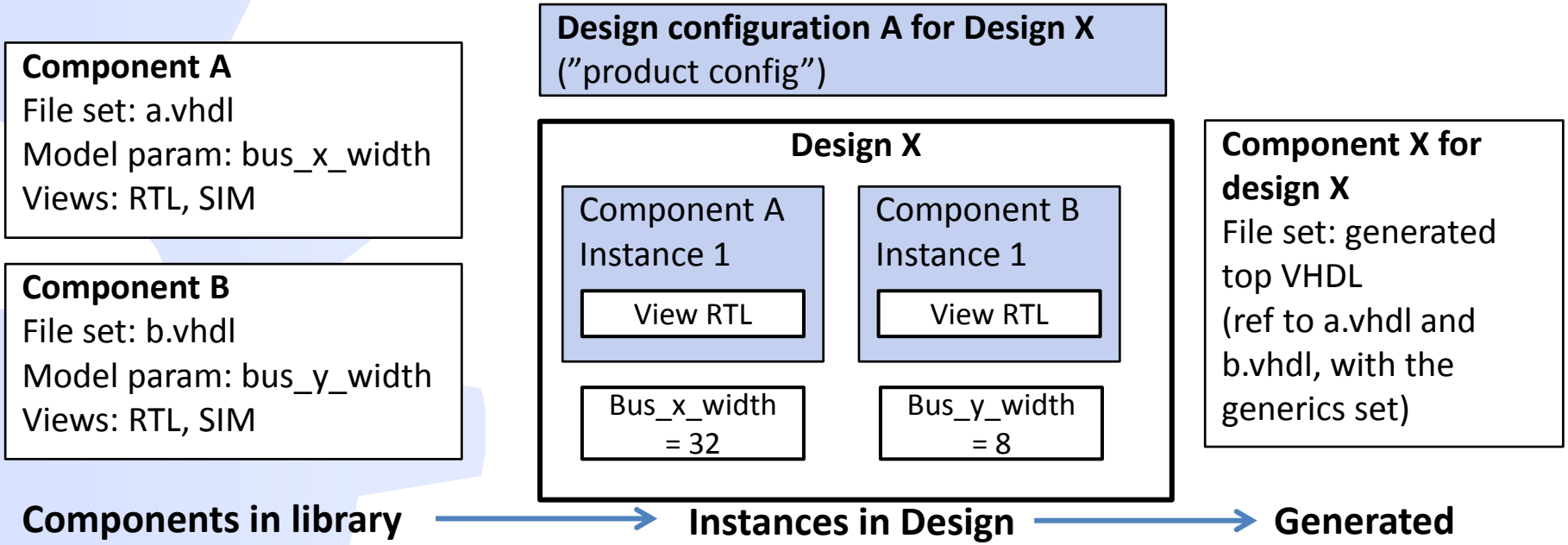


Example: Parameter

- Any parameter-value pair can be used within the IP-XACT component



Configuration process



- IP-XACT component have properties that are common to all its instances
- IP-XACT design and design configuration hold all instance specific properties
  - "Configurable element value"define model parameter value
- Generated files can be stored to the file sets of the top component of the design
  - Not defined in the standard, but follows the hierarchy

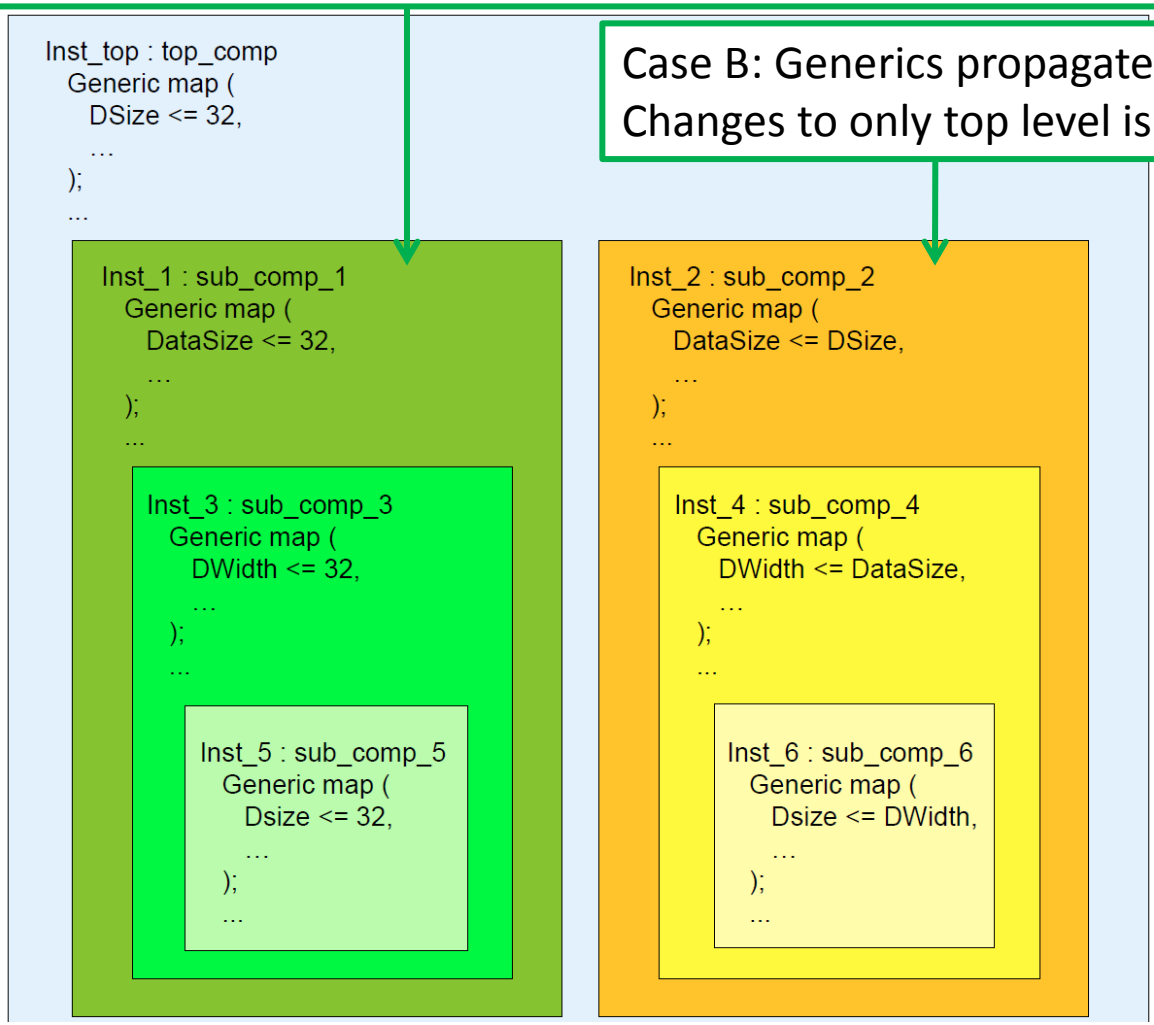
IP-XACT model parameter propagation approach

- Basic idea: IP-XACT objects encapsulate their own properties that should not depend on other objects (for maximum reusablility)
- IP-XACT makes **only downward hierarchy references**
  - IP-XACT component can refer to a design (lower hierarchy))
  - IP-XACT design can not refer to a component (upper hierarchy), only for a component instance at lower hierarchy design
- IP-XACT **propagates model parameters only for one level**
  - From design to component instances
  - Not from a component to designs
- Notes: Using fixed values (no parameters at all) are the safest (debugging), but blows out the number of objects in library
  - E.g. individual components for all fixed bus width variations
  - Laborous to handle manually, but not a problem for automated tools

Propagation of generic values in VHDL design

Case A: Generics do not have any dependencies between hierarchy levels: each component should be edited separately if something changes in top level

Case B: Generics propagated from top level: Changes to only top level is required



VHDL generic benefits and problems

- Easy to make different versions by propagating generics downwards
- Less files in disk, since only top-level components are created for each configuration
- From a sample VHDL file it is difficult to tell what generic values were used and where they are coming from
- Difficult to debug and track product configurations
- Propagation of VHDL generics can be implemented in IP-XACT in two ways
  - Implement tool automation for method A (preferred)
  - Mimic VHDL generics propagation in method B (possible but as error prone as VHDL)

IP-XACT model parameter in VHDL Case A

- Basic mechanism: **design has configurable element values** (name, value) that set **component model parameters** (instance specific)
- If fixed, no parameters are required
- Still possible to use modelParameter, but no dependency from design to its top level component
- Tools can automate the process

Component: top_comp (topmost level)
modelParam: DSize = 32

Design: top_comp
confElementValue: none, or DataSize = 32

Component instance: sub_comp_1
modelParam: none, or DataSize, or DataSize=32

Design: sub_comp_2
confElementValue: none, or DWidth = 32

Component instance: sub_comp_3
modelParam: none, or DWidth, or DWidth = 32

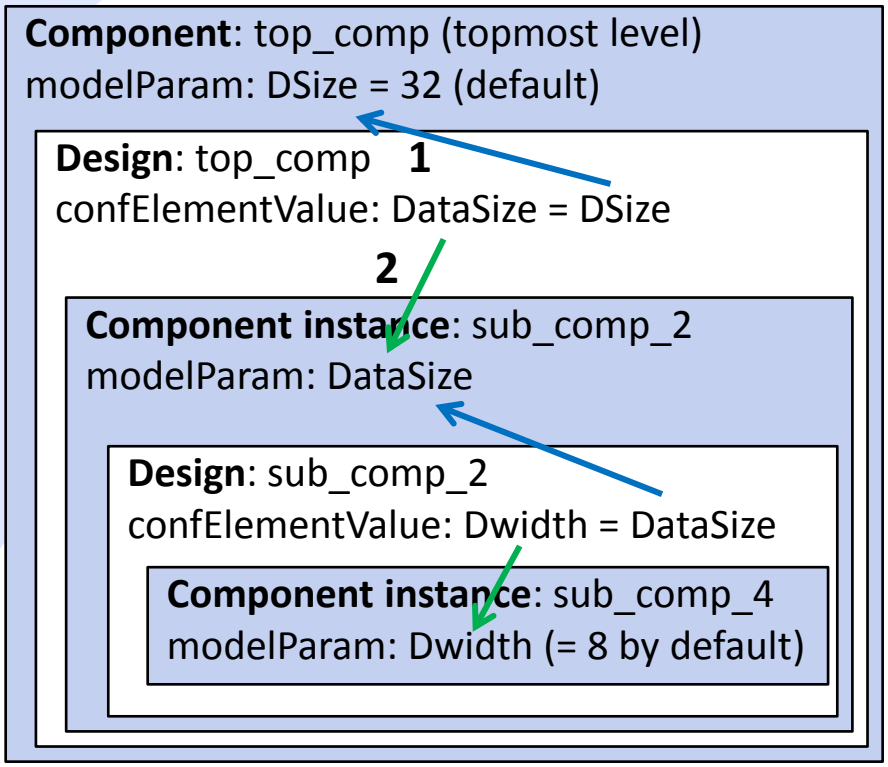
```
Inst_top : top_comp
  Generic map (
    DSize <= 32,
    …
  );
  …
    Inst_1 : sub_comp_1
      Generic map (
        DataSize <= 32,
        …
      );
      …
        Inst_3 : sub_comp_3
          Generic map (
            DWidth <= 32,
            …
          );
          …
            Inst_5 : sub_comp_5
              Generic map (
                Dsize <= 32,
                …
              );
              …
```

IP-XACT model parameter in VHDL Case B

1. From component to design
This is **not specified in standard**, but possible: use the same parameter names in components and designs

2. From design to lower hierarchy component
The standard way: IP-XACT design define values for component instance model parameters

- Propagation of generic values
- Kactus2 v. 1.3 VHDL generator follows this princpiple




```
Inst_top : top_comp
  Generic map (
    DSize <= 32,
    …
  );
  …
```

```
Inst_2 : sub_comp_2
  Generic map (
    DataSize <= DSize,
    …
  );
  …
```

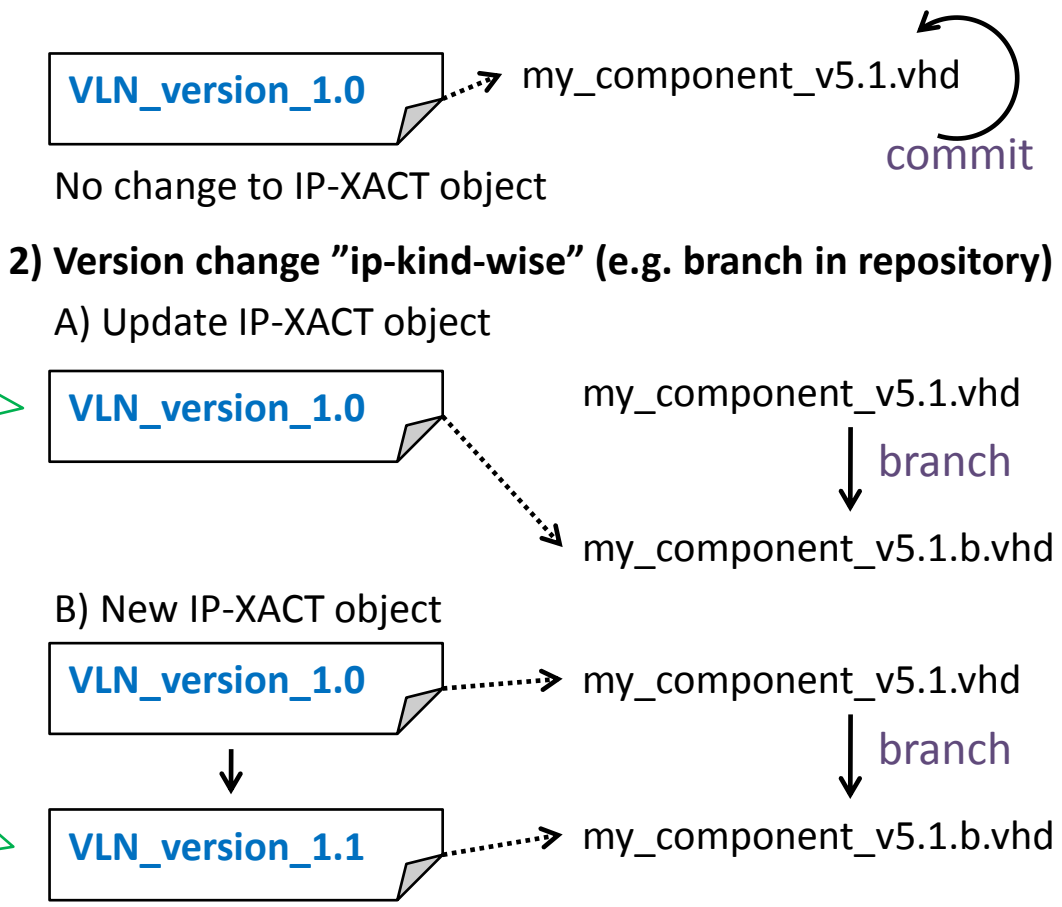
```
Inst_4 : sub_comp_4
  Generic map (
    DWidth <= DataSize,
    …
  );
  …
```

```
Inst_6 : sub_comp_6
  Generic map (
    Dsize <= DWidth,
    …
  );
  …
```

About versioning

Benefit, but also a drawback if misused: propagate change to all objects

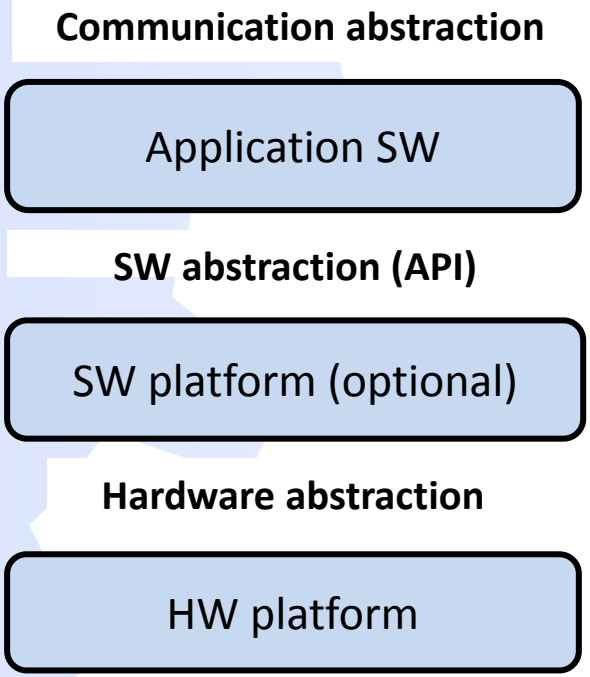
No automatic propagation, but requires modifying other IP-XACT objects referring to original version ”1.0”



Introduction to Kactus2 extensions for SW, HW/SW mappings and communication

The general layer model

- Reuse is implemented by strict use of layers



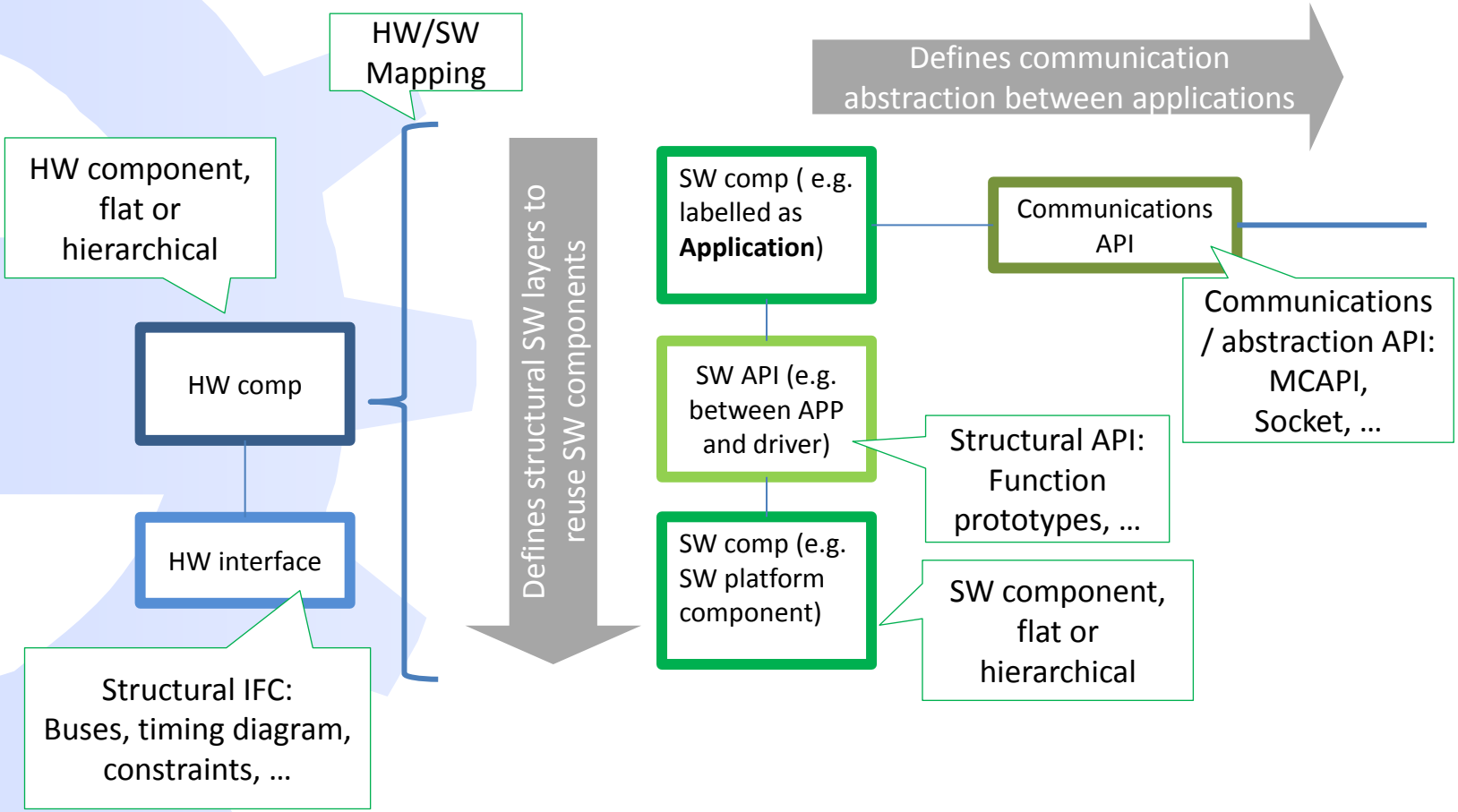
Communication protocols between applications

Platform independent SW code (may be also OS independent)

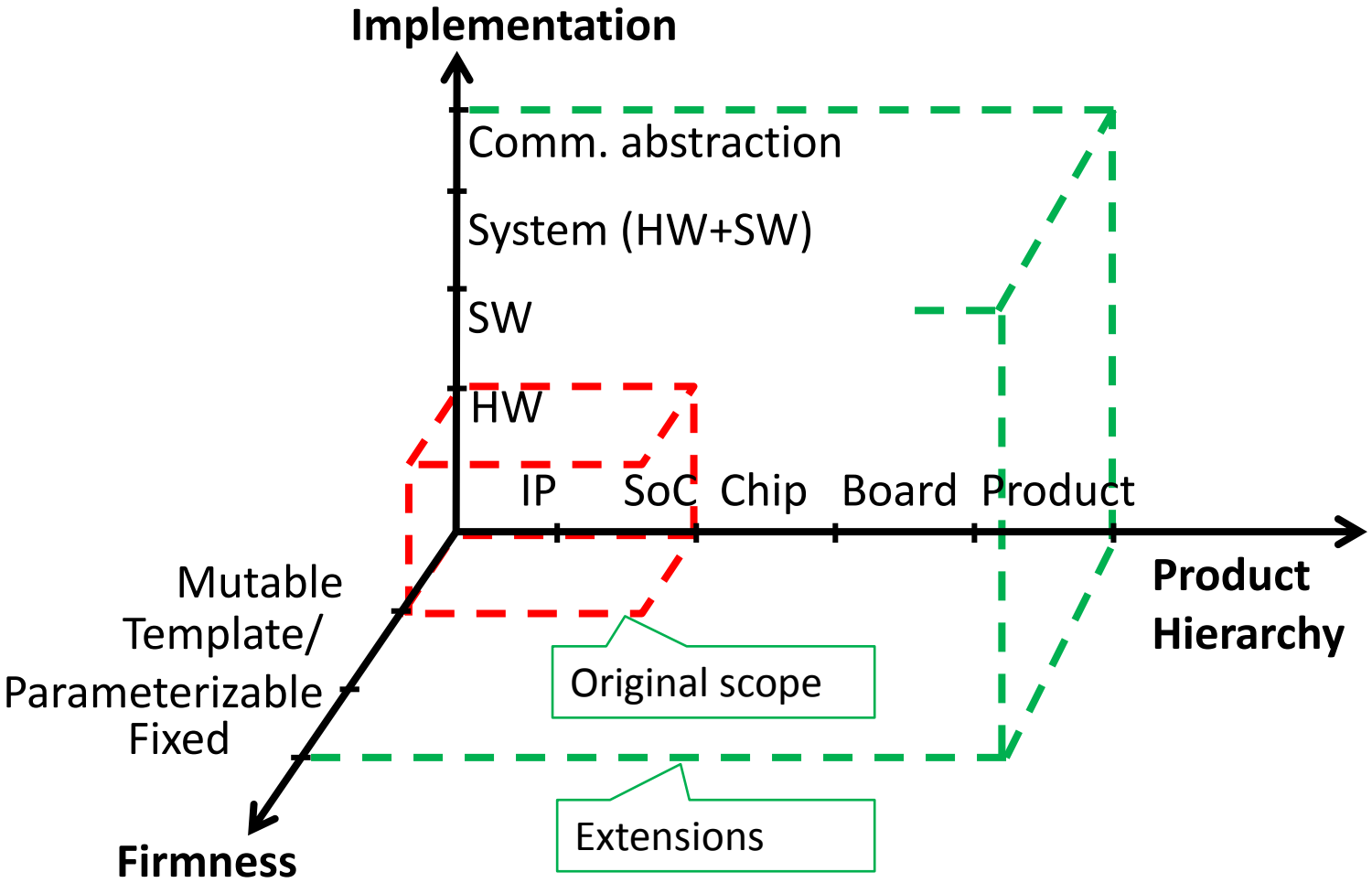
Application independent SW code (Operating system etc.)

Physical execution, storage and communication

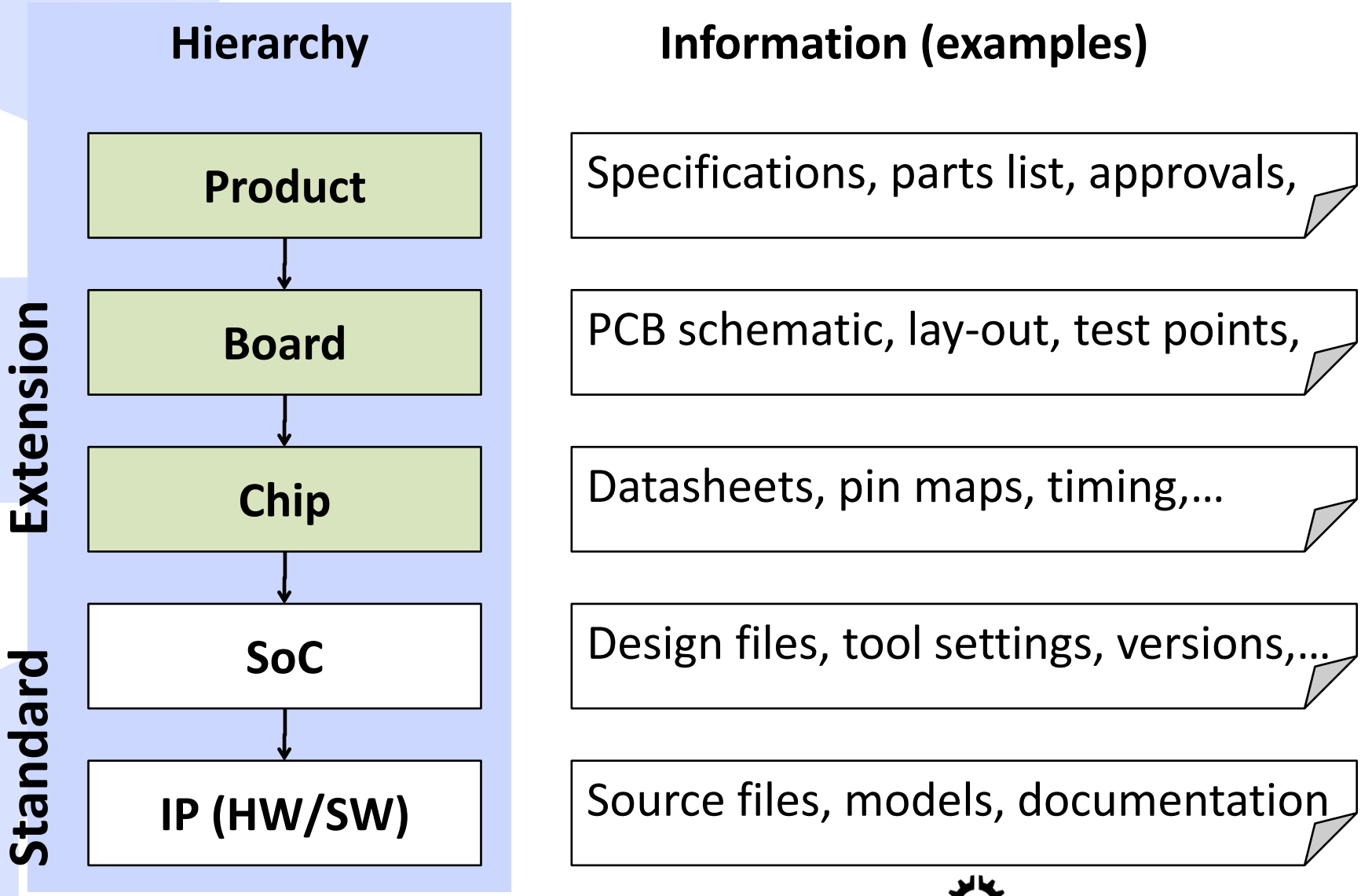
Kactus2 layer model



Kactus2 IP-XACT extensions

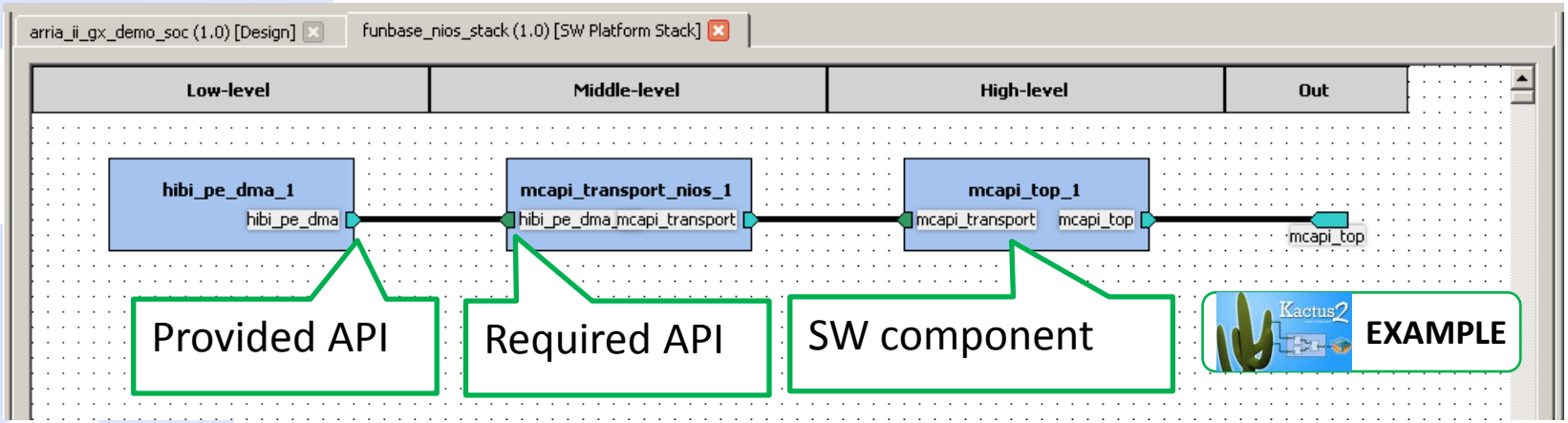


Kactus2 IP-XACT Extension: Hierarchy



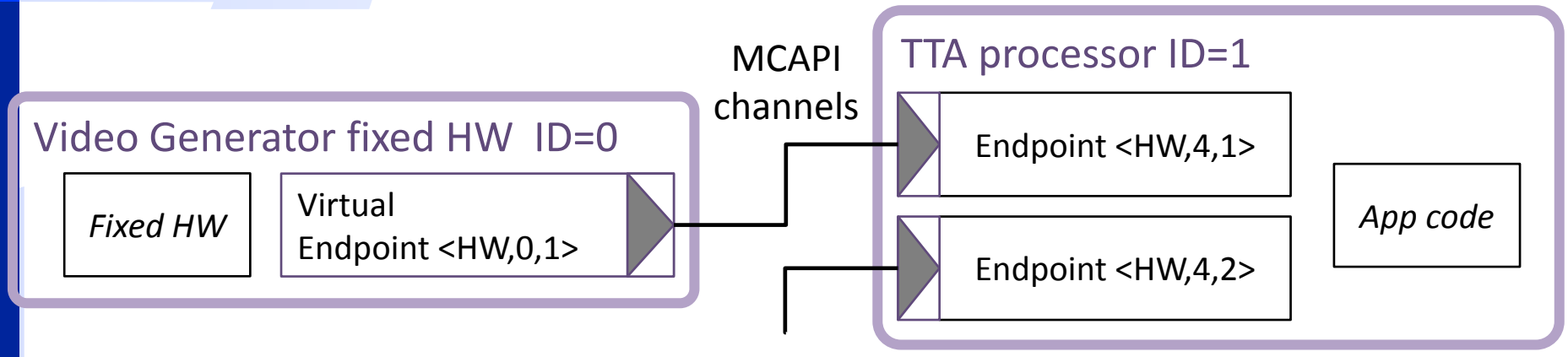
Kactus2 IP-XACT Extension: SW

- SW components have VLNV identity
- Structural description of SW
- Provide means of SW code validation against formally defined APIs
- Example: a software platform stack

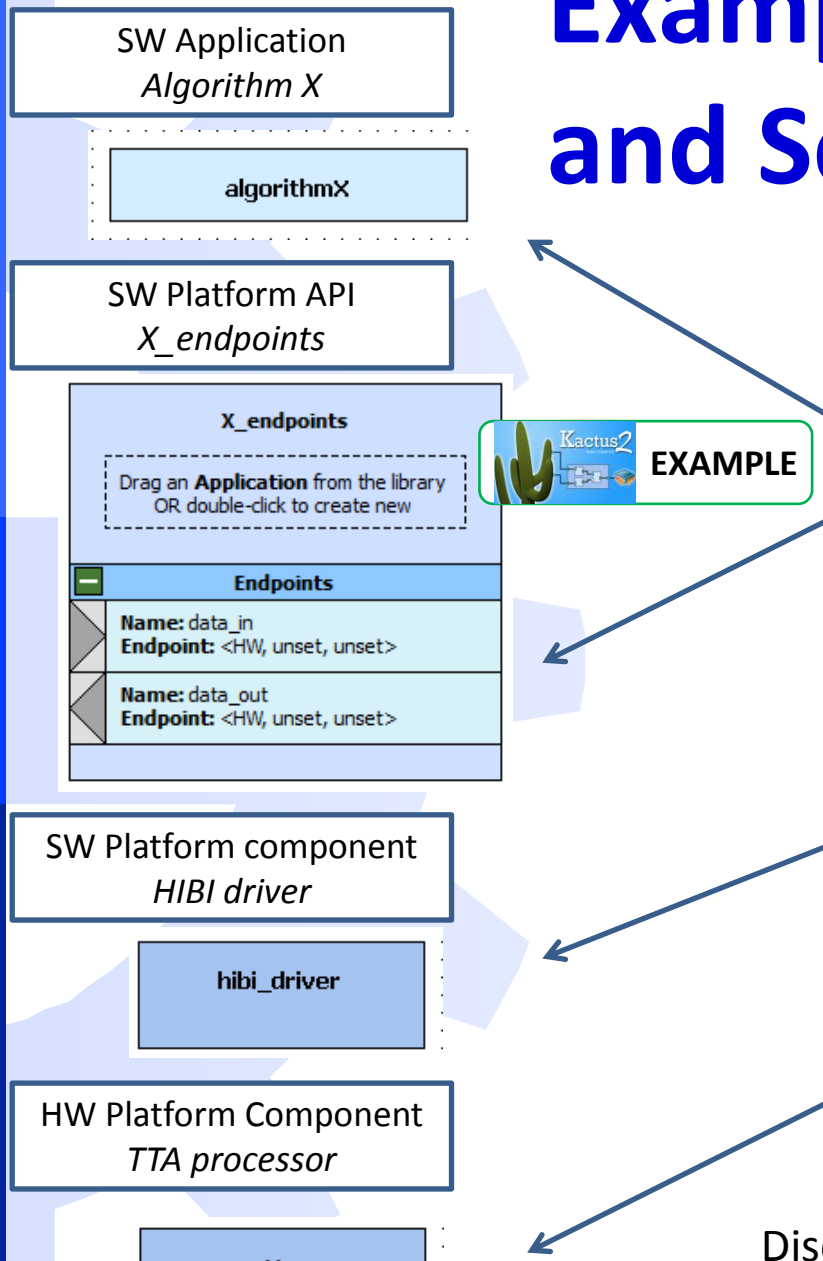


Extension: Application Communication interface

- Defines how HW or SW application communicate with each other
- First implementation: Multicore association MCAPI
  - Defines logical communications topology
  - Programmers view to product through hierarchies
- HW components are seen as *virtual MCAPI nodes*
- Benefit: Applications are portable between HW/SW and SW/SW



Example IP-XACT Library and SoC Layers



Communication abstraction

Application SW

SW abstraction (API)

SW platform (optional)

Hardware abstraction

HW platform

Disclaimer: Kactus2 v1.4 onwards will present changes

Component creation

SW Application
Algorithm X

algorithmX

SW Platform API
X_endpoints

X_endpoints

Drag an **Application** from the library
OR double-click to create new

Endpoints

Name: data_in
Endpoint: <HW, unset, unset>

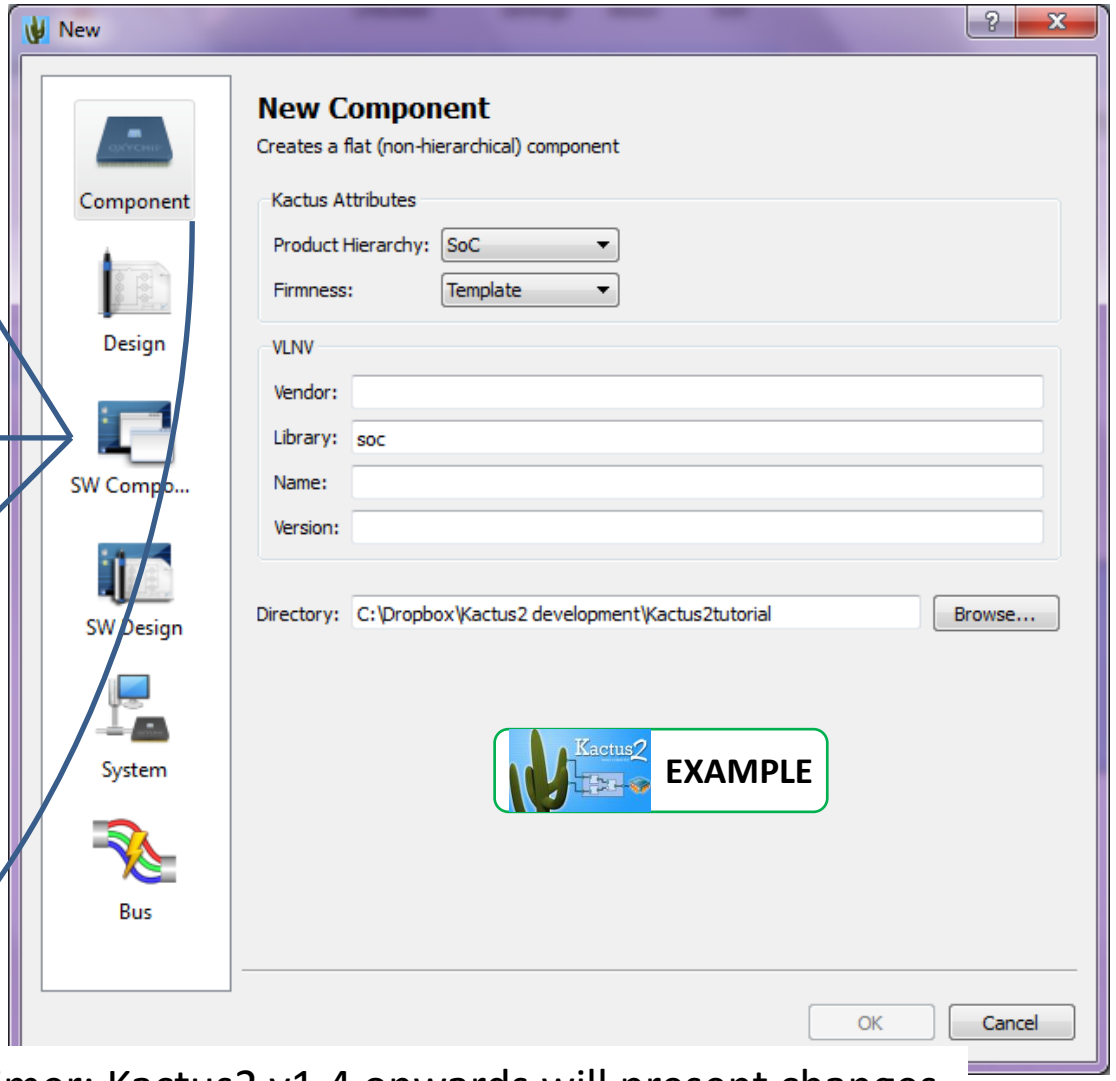
Name: data_out
Endpoint: <HW, unset, unset>

SW Platform component
HIBI driver

hibi_driver

HW Platform Component
TTA processor

tta
hibi



Disclaimer: Kactus2 v1.4 onwards will present changes

SW component creation

SW Application
Algorithm X

algorithmX

SW Platform API
X_endpoints

X_endpoints

Drag an **Application** from the library
OR double-click to create new

Endpoints

Name: data_in
Endpoint: <HW, unset, unset>

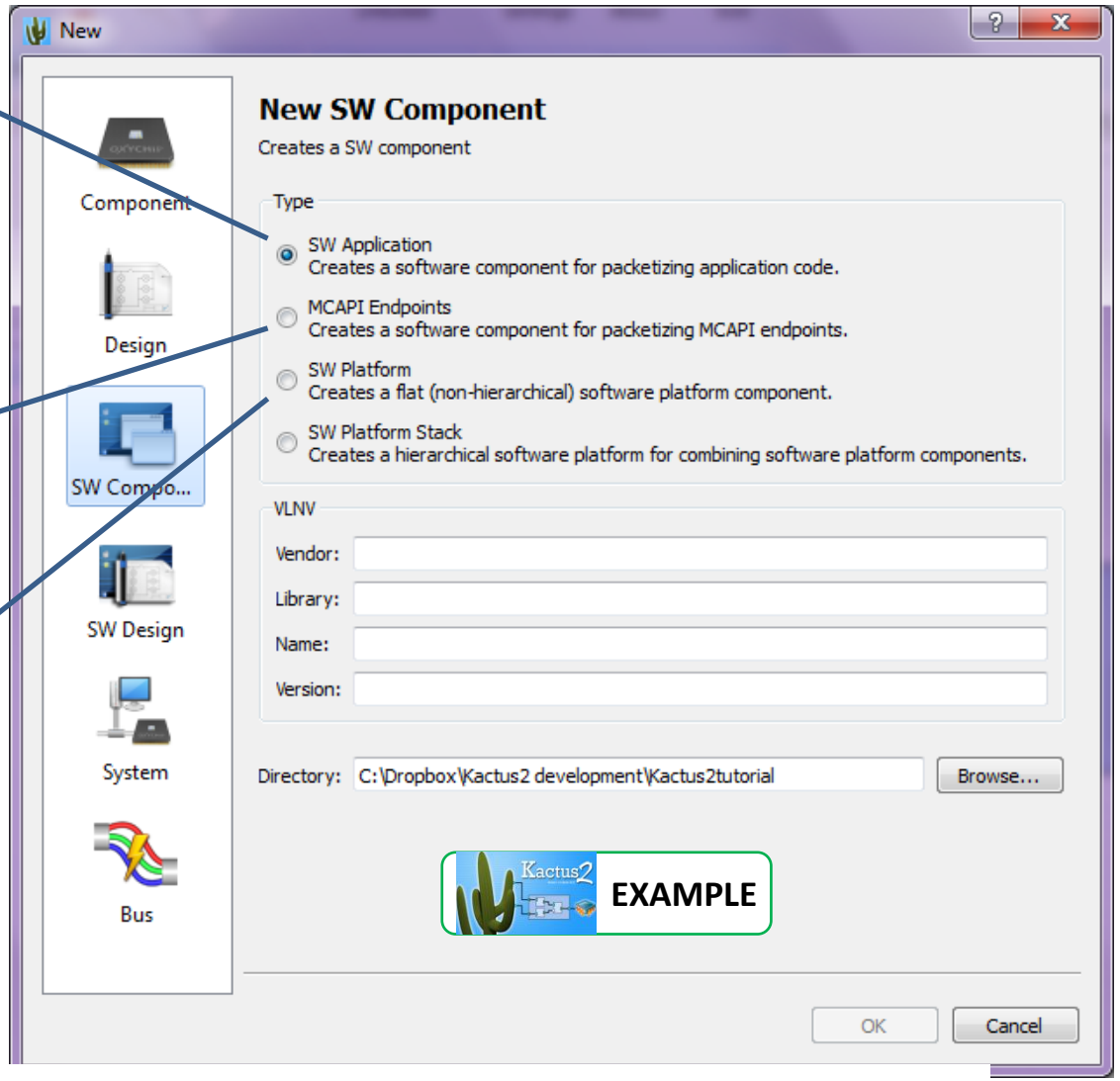
Name: data_out
Endpoint: <HW, unset, unset>

SW Platform component
HIBI driver

hibi_driver

HW Platform Component
TTA processor

tta
hibi



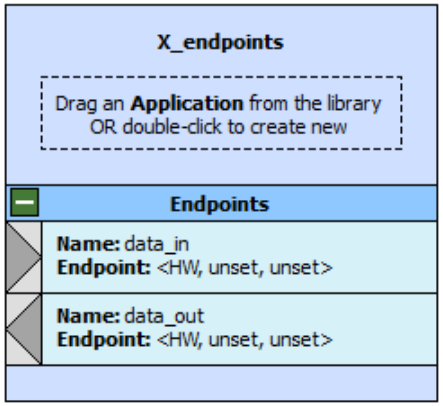
Disclaimer: Kactus2 v1.4 onwards will present changes

System design from HW and SW components

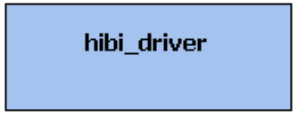
1. Library items

SW Application
Algorithm X

SW Platform API
X_endpoints



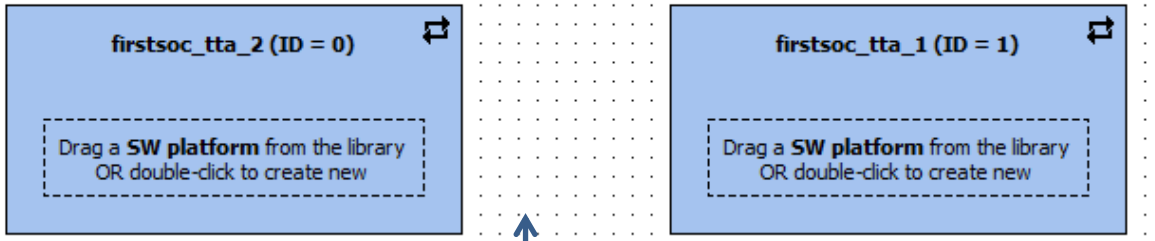
SW Platform component
HIBI driver



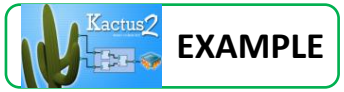
HW Platform Component
TTA processor

2. HW design

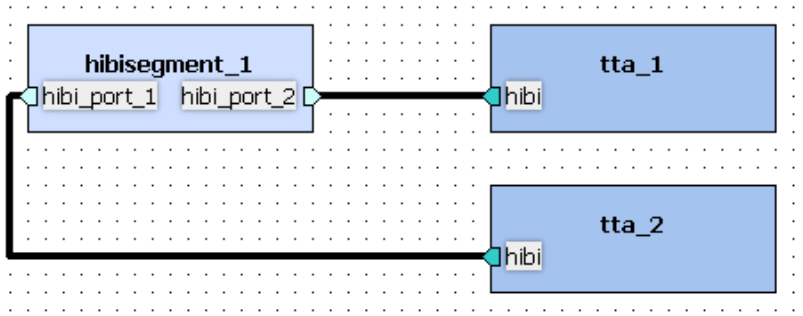
System design
Firstsoc



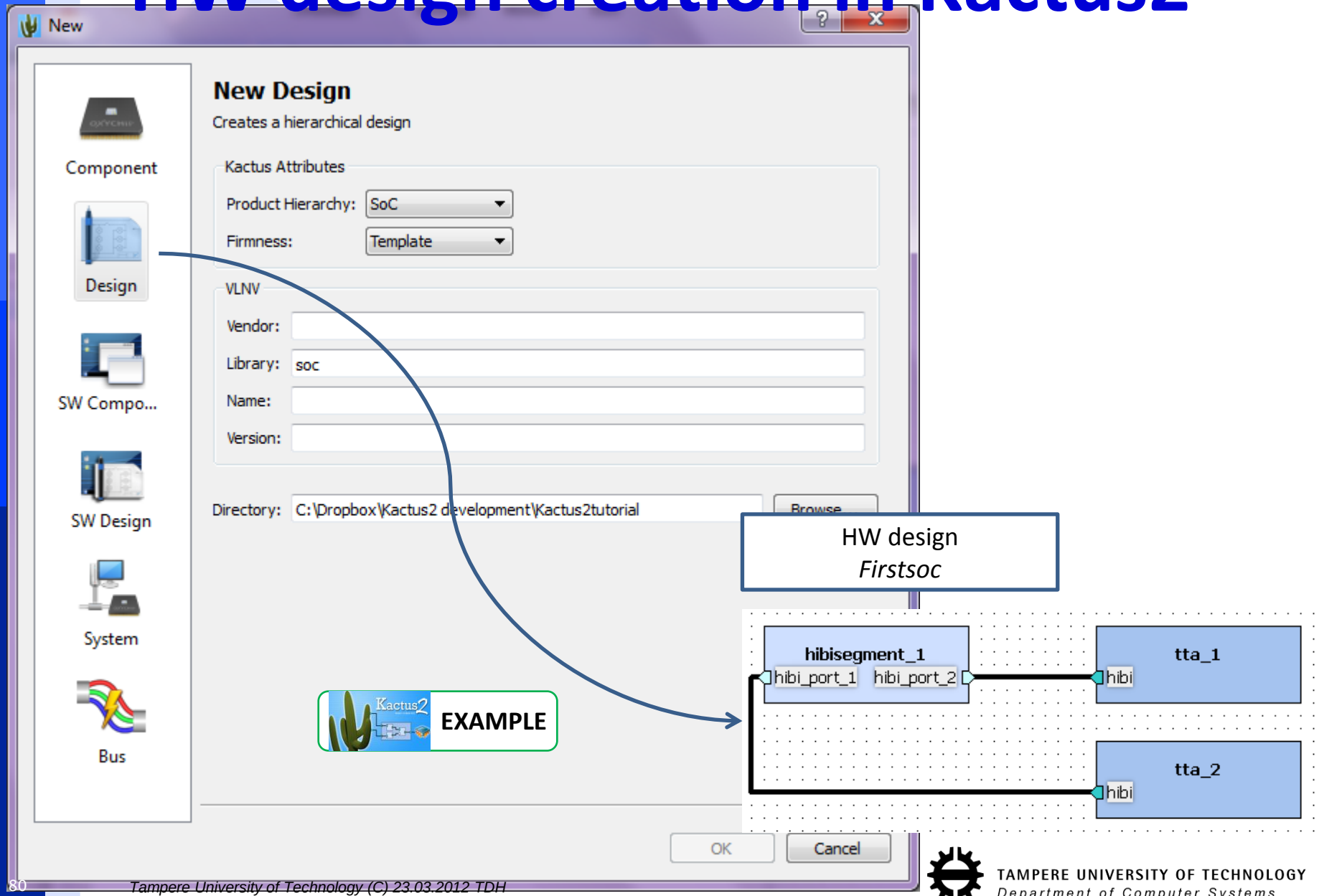
3. SW to HW mapping = Kactus2 system design



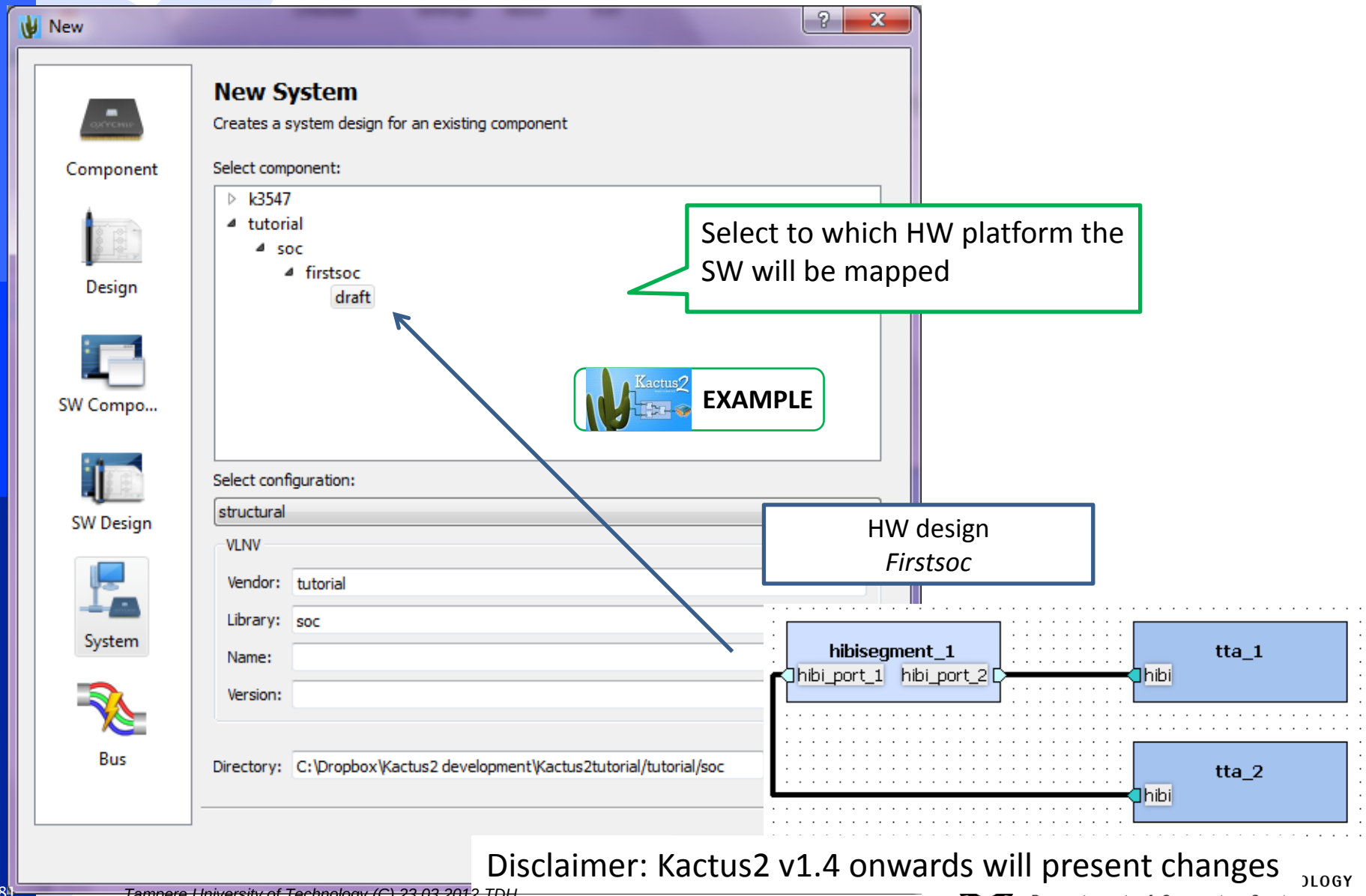
HW design
Firstsoc



HW design creation in Kactus2



System creation in Kactus2

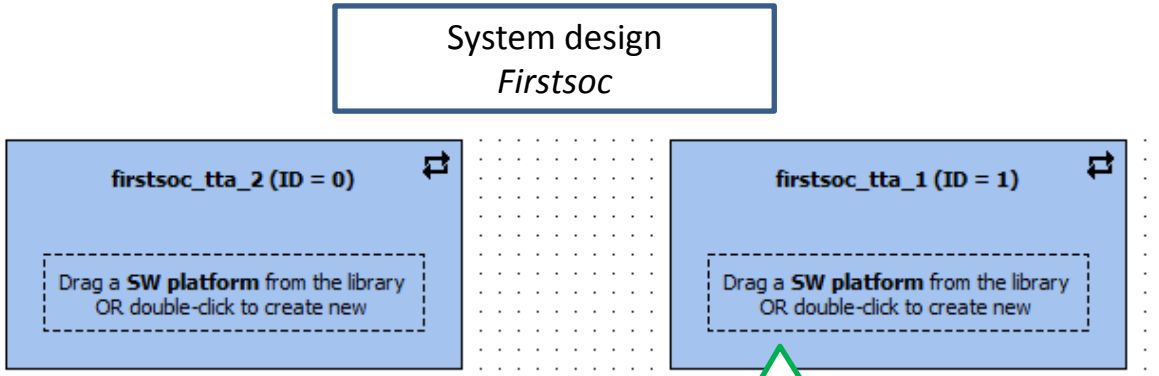


Disclaimer: Kactus2 v1.4 onwards will present changes

System creation in Kactus2

This is the HW component instance that accomodates SW instances

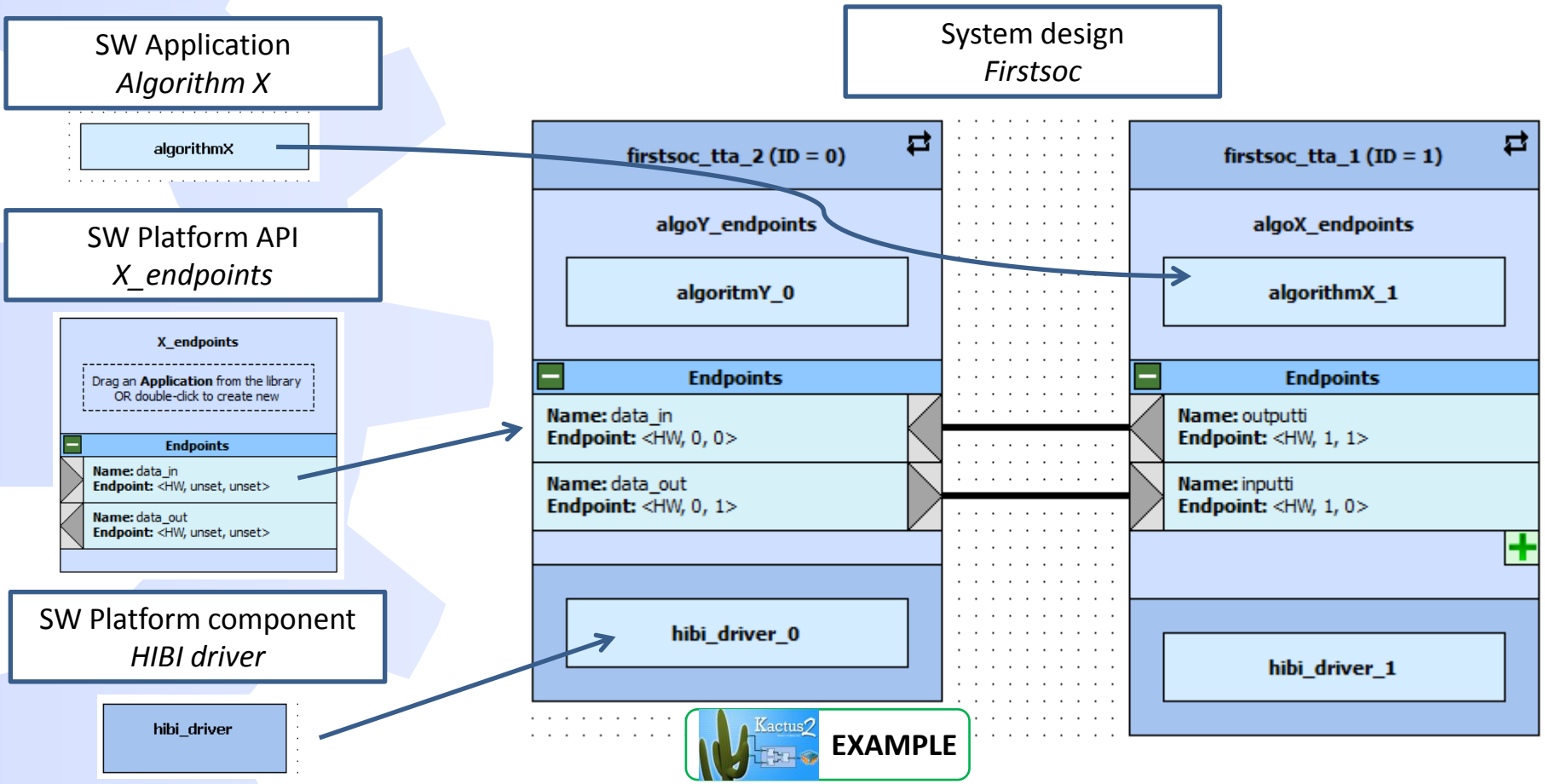
EXAMPLE



Drag-drop, or
Click to create from scratch to complete:
- SW platform or stack
- Endpoints
- Applications

- Note: Kactus2 v 1.3 supports application SW components only together with MCAPI endpoints (but you can use **empty endpoints** if needed)
- **Kactus2 from v1.4 generalizes communication interface to support also other than MCAPI abstraction**

Complete Kactus2 system design

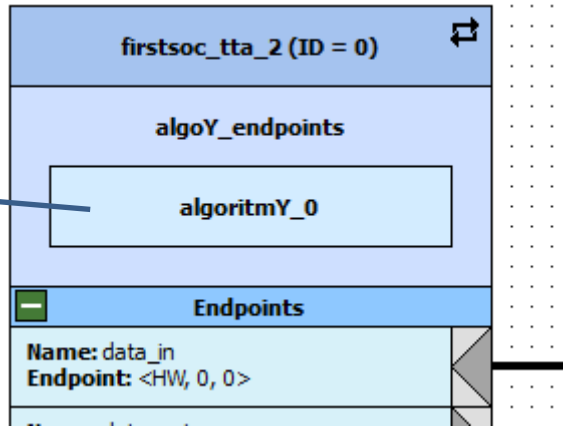


Disclaimer: Kactus2 v1.4 onwards will present changes

App code template generation

firstsoc (draft) [System]* | X_endpoints (draft) [MCAPI Endpoints] | main.c [Code]

```
/*
 * File: main.c
 *
 *  Generated by Kactus2 on 2012-01-23.
 */

#include <stdlib.h>
#include <stdio.h>

// This header includes the Kactus2 generated MCAPI code.
#include <ktsmcapicode.h>

int main(int argc, char* argv[])
{
    // Initialize MCAPI.
    if (initializeMCAPI() != 0)
    {
        // TODO: Write error handling code.
        return EXIT_FAILURE;
    }

    // TODO: Write your application code here.

    // Finalize MCAPI before exiting.
    mcapi_finalize(&status);
    return EXIT_SUCCESS;
}
```



Kactus2 EXAMPLE

TO BE ADDED LATER

SW and HW/SW mapping, attributes in detail

References

- IEEE1685-2009 standard
- Lauri Matilainen, Antti Kamppi, Joni-Matti Määttä, Erno Salminen, Timo D. Hämäläinen, "KACTUS2: IP-XACT/IEEE1685 compatible design environment for embedded Multiprocessor System-on-Chip products", Tampere Univeristy of Technology, Report 37, 2011, ISBN 978-952-15-2625-1
- http:// funbase.cs.tut.fi
- http://sourceforge.net/projects/kactus2/