2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

A real world application of IP-XACT for IP packaging

Bridging the usability gap

Philip Todd, Dialog Semiconductor, Swindon, England (*Philip.Todd@diasemi.com*)

***Abstract*— The IP-XACT format has been evolving for several years, initially administered by the Spirit consortium and then through Accellera [1] to become an IEEE standard in 2009 [2], then updated in 2014. First impressions are that it should be an essential IP description and cataloguing format for all IP. This paper follows the author's attempt to describe an IP-based design using the IP-XACT format (following the role of component IP provider), describing some conventions that were followed to handle some more unconventional constructs. The flow from initial block design entry, through code and IP-XACT auto-generation to IP-XACT platform assembly is described.**

Keywords—IP-XACT; IP Reuse; Design auto configuration; Design auto generation; Platform assembly

I. INTRODUCTION

An IC design and development flow incorporating IP-XACT IP descriptions is described. For digital IP, a core set of features is captured in a spreadsheet. This data provides the starting point for an auto generation step, where outline RTL and verification (UVM) information is created, along with the IP-XACT elements. The outline RTL and UVM must be completed by hand to add the detailed "user code" functionality. The auto generated IP-XACT description contains sufficient detail to be used directly by "platform assembly" tools, which create a structural netlist and IP-XACT description for sub-systems or a chip top level. Extra metadata is added to the IP-XACT information to assist in cataloguing, search and exploration activities. Where necessary, custom vendorExtension fields are created to support the required level of IP-XACT detail.

For analogue components a similar development route is possible; in this case the block level design metadata is extracted from a Cadence Virtuoso environment using SKILL scripts to create the IP-XACT description.

The platform assembly tool is required to support all of the following: a digital only flow; an analogue only flow; a mixed signal flow with "digital on top"; a mixed signal flow with "analogue on top".

For the digital on top flow the analogue components are treated as black boxes in a digital assembly. For the analogue on top flow a symbol is required for each digital block and the assembled platform must be netlisted in the Virtuoso environment.

II. DESIGN ENTRY

A. *Digital design capture*

Initial design capture and configuration of the IP uses a hierarchical array structure in an Excel spreadsheet. The spreadsheet has separate sections for register descriptions, interfaces and (optionally) IOs and isolation.

Information is not entered directly into the spreadsheet. Instead the designer creates a set of visual basic files that contain the design data and make standard subroutine calls to generate the spreadsheet entries.

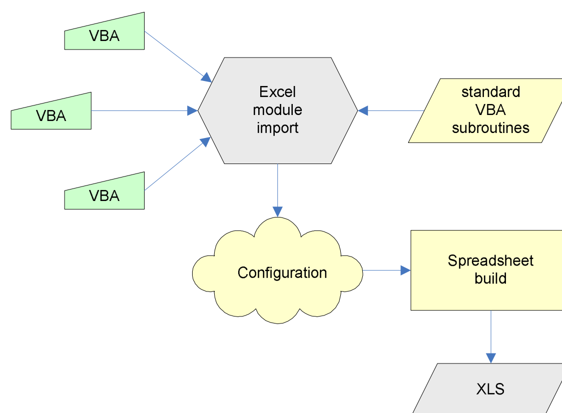

Figure 1: Digital design capture.

Importing visual basic files has a number of advantages over direct spreadsheet entry:

- textual description of functionality for better history tracking in SCMs, as opposed to a binary file.
- modularity. Repeated sub-functions can use the same VBA description, with variable substitution giving unique register and pin names and register offsets. This provides some protection from typos and saves time.
- configurability. A module can be configured after the VBA import step to vary the number of instances and perhaps other related parameters. Different design variants can be created this way from the same imported module elements.

The design capture is now divided into 3 stages: sub-function VBA module coding and XLS import; pre-configuration; expansion. The expansion stage calls the VBA module functions to “flesh out” the spreadsheet with the configured design information. At the end of expansion the spreadsheet is fully populated and ready for the code auto generation step.

A disadvantage of spreadsheet use for design entry is that it only supports single user data entry. This may be a greater problem for chip/system level design entry than for IP blocks, but it is largely mitigated here by the use of multiple VBA modules that can be individually coded in isolation. The module import, pre-configuration and expansion can be a final stage of the process.

B. Digital auto generation

The “Template Toolkit” tool (TT2) [3] is used to extract the datasheet information into a set of arrays and then populate the design files. Each TT2 “filter” script holds the template patterns for the required output syntax and expands these templates using the supplied array information.
At the end of the auto generation step a complete RTL design “shell”, including interfaces, registers and synchronisation, debounce and event circuitry is created, along with an IP-XACT description. A placeholder is left in the RTL for the user to manually add the core functionality of the block. If the spreadsheet is later changed, the auto generation process can be rerun to update the code (without changing the manually edited part).

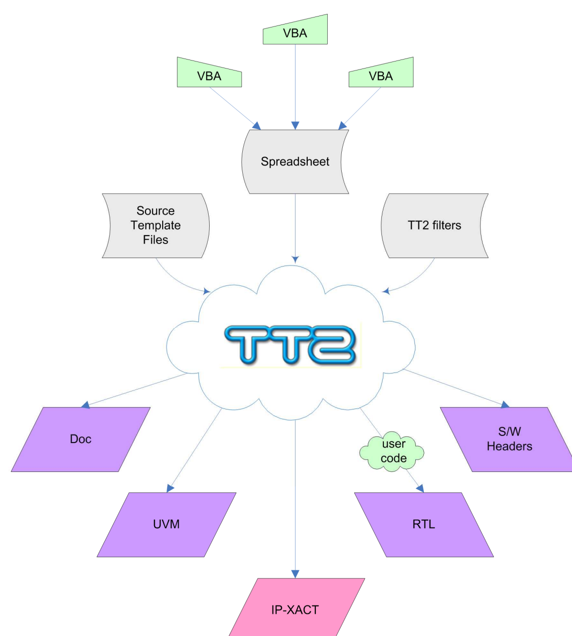

Figure 2: Digital auto generation.

The auto generation step also creates a UVM verification shell for the design, including environment, register model/address maps, drivers, monitors and scoreboard elements, covergroups and a test harness. To this the user adds sequence and scoreboard code to test the user code RTL functionality.
A third part of the auto generation flow creates headers for use in applications software, development tool GUIs and AMS simulations.
Finally an additional set of visual basic functions are generated that can be imported into Microsoft Word to create interface and register sections of the design documentation.

C. Analogue design capture

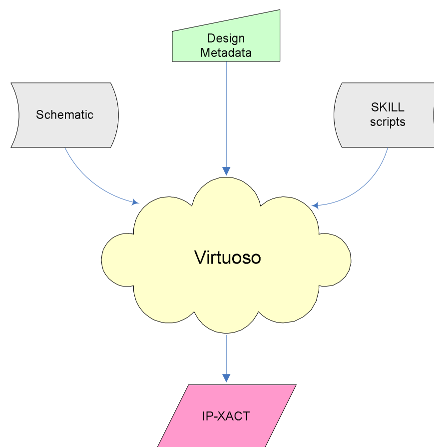

Figure 3: Analogue IP-XACT generation flow.

Analogue design entry follows a Cadence Virtuoso schematic editing approach as used in previous designs. As an additional step, once the design is complete, a SKILL script extracts key metadata from the design library and generates an IP-XACT description. This analogue IP-XACT description consist of a list of ports and some descriptive metadata carried as parameters in the *model / instantiations / componentInstantiation* element.

III. Creating the IP-XACT

A. IP-XACT extensibility

A feature of IP-XACT is its “extensibility”, so where a required feature has not yet been implemented in the standard syntax, a *vendorExtension* can be added. Hooks for vendorExtensions are provided at most points throughout the IP-XACT hierarchy.
For this project, one goal of IP-XACT usage is to remain within the standard syntax as much as possible and minimise the use of custom vendorExtensions. Custom fields complicate the process of IP exchange between vendors.
A subset of IP-XACT features is chosen for the IP description, with a set of common conventions used to ensure compatibility across the IP database.

B. IP-XACT hierarchy

Each IP description is distributed across a number of constituent XML files, each relating to a particular feature. Some of the files are referenced in multiple IPs. A *Catalog* XML file carries the information that links all of these description together with the target IP module.

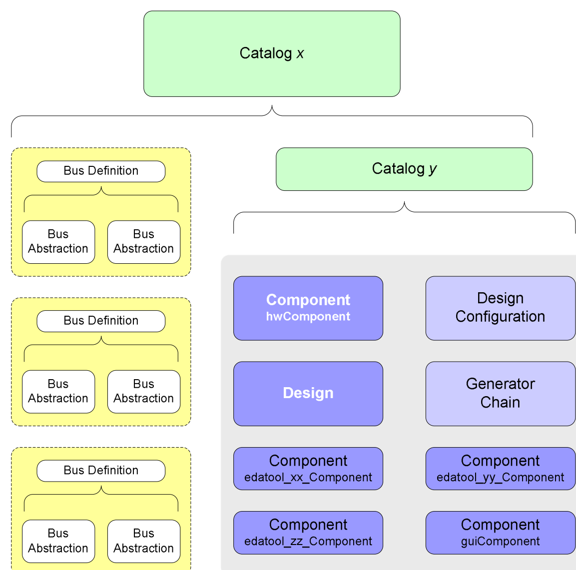

Figure 4: Example catalog container structure

Figure 4 shows the selected IP cataloguing hierarchy. The component and design information files are grouped in one catalog element (*catalog* y), then this catalog and the bus definitions and abstractions are referenced by a second, top level catalog (*catalog* x). This structure allows the separation of block design specifics from the global interface definitions. (for example in a design with multiple components there is no need to repeat the global xml elements in each IP reference, so only *catalog* y would be used).
A further subdivision is made within the component description to separate the core design information from additional eda tool and gui descriptors and vendor extensions. Multiple xml files for a single component are enabled by using a complex definition for the VERSION part of the VLNV descriptor. VERSION follows the format
*<viewType>***:***<versionMajor>***.***<versionMinor>*
where *<viewType>* identifies which component part is being described. The Vendor, Library and Name sections of the VLNV are identical for each component part.

C. Component

The basic building block for IP packaging is the *component* element. The component describes the user (e.g. software or integrator) view of the IP, and although the standard declares all of the constituent parts as optional, a minimum common subset is chosen to give an adequate description of the design:
Port lists: The *model* element contains a list of the IP block's top level pins, with a cross reference between the name used in the design (RTL, netlist) and an abstract logical name (useful when accommodating slight mismatches in naming when connecting signals between IPs). A list of the global bus interfaces (groupings of logical pins with related functionality) used by the IP is also included.
Memory maps: In its simplest form the memory map is a list of registers, register access types and addresses in the IP, as seen from the external world. Often a single memory map view isn't sufficient, as pin and register settings can impact this external view. IP-XACT allows for this with memory remapping.
Views and **instantiations**: In addition to physical port lists, the model element of a component can include multiple *views* and *instantiations*, which add more information about the component and any lower level IP connectivity.

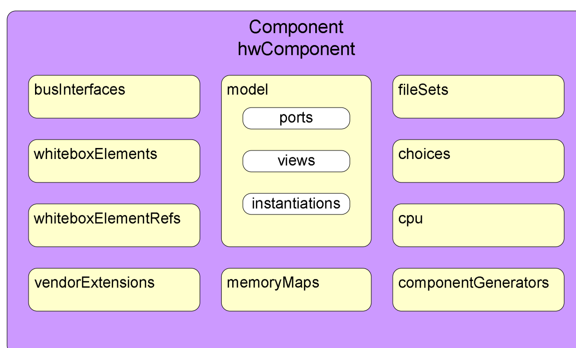

Figure 5: Example constituents of an IP-XACT component.

Other component elements are added to provide hooks into the design environment, as shown in Figure 5.

1) Memory maps and remapping

An example IP block contains one slave interface and an associated *memoryMap* element. Within this there is an *addressBlock* that includes all registers with their least restrictive access type[1] (the register superset), followed by a *memoryRemap addressBlock* for each modified view of the registers.

A modified register view is defined as a subset of the full register list, possibly with a different access type to the main memoryMap register definition, identified by a combination of port and/or register field settings. This is the memoryRemap definition.

Selection of the currently applicable memoryMap or memoryRemap is controlled by a combination of *remapStates* and *remapConditions* elements. The remapConditions element is included in the standard vendor extensions VE1.1 specification [1].

In general memoryRemap elements can be used to identify register groupings, test mode and restricted host views of a register map.

When there are many registers to describe the result can be a quite verbose description and a large XML file, because the entire memory map must be repeated for each remap. (A more compact implementation would allow remapping of only the subset of registers that have a different view to the default memoryMap description.)

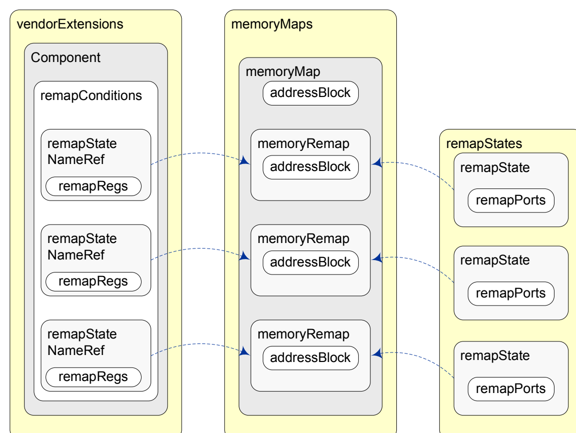

Figure 6: memoryMap, memoryRemap, remapStates and remapConditions interaction within a Component.

a) remapStates

[1] least restrictive access type: for example R/W if the register appears as both read and write enabled to at least one bus master. If the same register should appear as something other than R/W in another view, then a memory remap is required.

remapStates define the port name and value associated with the memoryRemap elements. There should be a single remapState for each memoryRemap, but a remapState may contain multiple port/value pairs. The remapState name must match the *ipxact:state* entry of the associated memoryRemap.

b) remapConditions

remapConditions define register values associated with the memoryRemap elements. The *remapStateNameRef* entry links the remapState and memoryRemap elements with the remapCondition. Each remapCondition can include multiple register entries (there is no field resolution – instead register *value* and *mask* are used).

Example:

A component contains a single register with two fields; cfg[2:0] and trim[3:0].

The following requirements have been specified for the register fields, depending on the value of ports "emstr" and "tvalid" and register field "tlock":

- cfg[2:0]: This field is visible and has R/W access if emstr is '0'.
- trim[3:0]: This field is visible and writeable if tlock is '0' and either of tvalid or emstr is '1'. If tlock is '1' then the field is visible (but with read-only access) only when tvalid is '1'.
- tvalid and emstr shall not both be '1' at the same time.

There are five possible views of the register:

	tvalid	emstr	tlock	cfg [2:0]	trim [3:0]
Map				R/W	R/W
Remap #1	0	0	x	R/W	
Remap #2	0	1	0		WO
Remap #3	0	1	1		
Remap #4	1	0	1	R/W	RO

If none of the remap prerequisites are met then the default memoryMap view is selected.

2) Registers

The IP-XACT registers are contained in addressBlocks, which begin with base address and range information, then add details for each register and register field. Most of the detailed information is placed in the field descriptions, with only name, description, address offset and size required at the register level.

Field resets are made up of value/mask pairs and a *resetTypeRef* (which relates the reset value to the source of the reset) Multiple reset sources are permitted, so a distinction between power-on reset, hardware and software reset responses is possible.

To provide verification hooks, the hierarchical path to each register field is included in the accessHandles element.

IP-XACT provides a number of modifiers to the field access type, such as *modifiedWriteValue* and *readAction*, that allow register functionality to be accurately described. Where this is still not sufficient for writing the writeValueConstraint value is set to useEnumeratedValues and a set of enumerations is declared.

An example is a control register type. Each control register takes 2 bits – a master bus write of 01_b sets the control and a write of 10_b clears it. Register reads always return 00_b or 01_b (01_b when the control is 1).

The IP-XACT *enumeratedValues* are coded as follows for the control register:

usage	name	value
write	unchanged	00_b
write	reset	10_b
read-write	set	01_b
read	clear	00_b

3) Quirky register handling

Register *quirkyness* becomes more apparent when trying to describe some of the register field options allowed during the design entry stage. In some of these cases custom vendorExtensions are required.

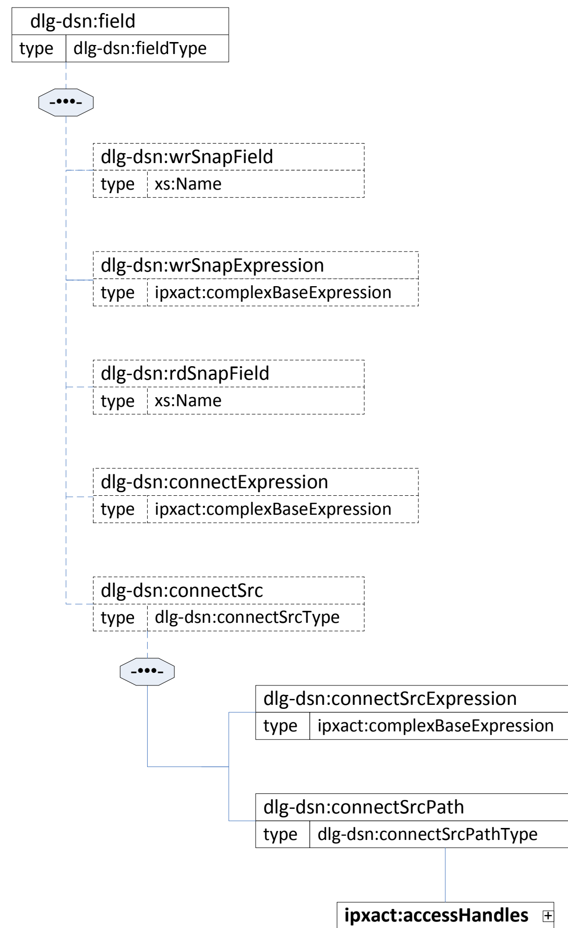

Figure 7: custom field vendorExtensions.

a) Write snapping

A write snap register can be viewed as a 2-stage pipeline, where the master bus write access fills the first stage but the second stage output (to the design target) depends on either a signal value or a write access to another register field. Updates of multiple registers can therefore be synchronised by using a common write snap master to control the update timing.

field write snapping: the snap control is triggered when another register field is written. The control field is identified by the **wrSnapField** custom vendorExtension.

expression write snapping: this time the write snap control is a SystemVerilog expression; the full RTL expression is copied into the **wrSnapExpression** custom vendorExtension.

b) Read snapping

A read snap register is also used for synchronisation; this time the read value is updated only when the read snap control field is read. Multiple status registers can therefore be updated simultaneously (useful when sampling a timer value spread across multiple register addresses – the read snap field here being set to the least significant register of the timer). The **rdSnapField** custom vendorExtension is used.

c) Register to pin output connections

Direct connections between register outputs and module pins. This information is required in the IP-XACT description in case the output pin itself becomes a remap state control or a status register source signal in

another module higher up in the hierarchy. The platform assembly process would need to take this into consideration when creating the combined memory map/remap tables. It also provides some white box hints to verification.
The custom vendorExtension is **connectExpression**; the HDL path to the register field output is already supplied in the accessHandles element, while the vendorExtension is any legal SystemVerilog syntax for the left hand side of the assignment.

d) Pin to register input connections

A description of the input path from pin to status registers is also needed. This time however there is no ready-made access handle for this side of the register. A complex vendorExtension is created to include both the access handle to the register input and the SystemVerilog expression that drives it.
The custom vendorExtensions in this case are **connectSrc, connectSrcPath** and **connectSrcExpression**.

4) Analog and exploration metadata and component instantiations

The information extracted by SKILL script from the Virtuoso schematic consists of a set of module ports and a list of name value-pairs for the additional metadata. Additional information can also be added (for both analogue and digital designs) to assist with later catalog searching and design exploration tasks.

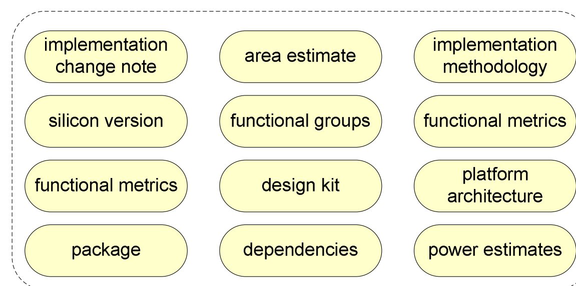

Figure 8: Example metadata parameters.

Component metadata <name><value> pairs are added as *parameters* in a *model / instantiations / componentInstantiation* element.

Parameters can have the *choiceRef* attribute. This is intended for use by generator tools where an interactive value selection can be made from a selection of legal values, but here it provides a hint of legal parameter values to a checker. Each choiceRef is a reference to a *choices* element (in the component) that defines all of the legal values for this parameter name.

D. Interfaces

1) busDefinition and busAbstraction

IP-XACT bus interfaces are port groupings, split into a bus definition (high level information including whether the bus contains addressing information that relates to a memoryMap element) and one or more abstraction definitions (providing low level information such as member logical port lists, widths and directions in different interface modes). There might be multiple abstractions (e.g. RTL, TLM) for a bus.

The *busDefinition* and *busAbstraction* elements are used for any grouping of ports in a component that constitute a bus and include memory map (register) access controls. In some cases it is appropriate for other ports that are related by some functional grouping.

Component elements use *busInterfaces* to connect logical ports of a busAbstraction to the physical ports of the component and select the mode of the interface that applies (for example master or slave).

2) SystemVerilog interfaces

SystemVerilog (SV) interfaces have a close conceptual relationship to IP-XACT bus interfaces, but the exact mapping between the two isn’t very well defined within the current standard.

In this flow SV interfaces are described using a combination of wire and transactional busAbstractions.

Each transactional busAbstraction defines one modport of one interface with the naming convention svif:<*if_name*>**.**<*modport_name*>. There is a single transactional port, onSystem type with logicalName <*if_name*>**.**<*modport_name*> and group <*modport_name*>.

All constituent interface ports are identified in a single wire busAbstraction. For each port the width and direction is added to the logicalName and modport group name.

The transactional elements are sufficient to describe platform interconnect in IP-XACT designs, but the design auto-generation and checking tools need to be SV interface aware. For example, if a declared remapPort is a member of an SV interface then it will not be visible to the generator/checker unless the transactional busAbstraction is cross-referenced into the wire busAbstraction and expanded.

IV. Platform Assembly

The automatic platform assembly step involves initially creating a top level IP-XACT catalog element that lists all of the constituent catalogs for the IP being assembled. A configuration file is also created to complete the baseAddress expression for each enclosed memoryMap (by setting a parameter value), and add the top level port list and any additional name-matching rules to help with the platform interconnect.

Using a combination of SystemVerilog interfaces and/or well-defined IP-XACT bus definitions can minimise the number of extra rules that will be required, making the connectivity step a relatively straightforward exercise of (mostly) name matching.

The following is one example set of platform assembly rules and conventions that have been used:

- A top level port list is read from the configuration file.
- Some “standard” common connections are created for all IP, e.g. clock, test interfaces, slave bus interface.
- Dedicated port to IP bus connections are found and connected by name, with the naming convention *<ip name>_<if type id>_<instance>*.
- Special rules for certain interface connection types are applied, e.g. daisy-chaining. These might be identified by naming convention or explicitly in the configuration file.
- Internal IP to IP “sideband” signals are accumulated in a common SystemVerilog interface (“levels”), with individual modport definitions for each (reusing the IP module name as the modport name). A local IP version of this interface exists in each IP module definition, with the same interface name but containing a single modport. The new accumulated interface definition is used in place of all of the individual IP level interfaces. The modport signals in each connected module must have matching names, so the instance connection is simply made using *<if_name>.<ip_name>* signal naming.
- Module to master “events” (interrupt flags) are accumulated in another SystemVerilog interface and ORed together (within the interface definition) to create a single event flag output port. Following the same approach used for the sideband signals, there is an IP version of this events interface for each IP module that outputs events, with a single modport in each. These IP interfaces are again replaced with the accumulated top level version.
- Internal signal declarations are created for signals/interfaces used in assembly but not in the port list.

In addition to the automatically assembled region, a user code area is reserved for manual additions (if any are required).

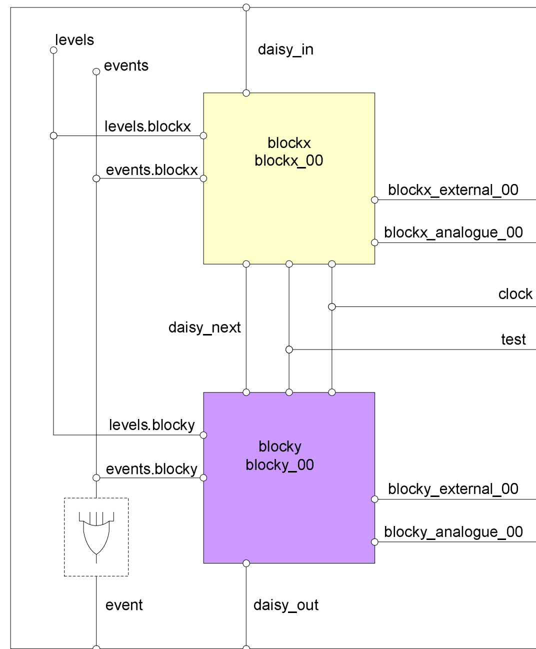

Figure 9: Example IP platform structure (SV interface connections shown).

The IP-XACT files for each IP must be read and processed as part of the new top level design IP-XACT description. IP level memoryRemap definitions can cause complications when attempting to build the top level, so some manual intervention is occasionally required to complete the process.

The output of platform assembly is a new module netlist including IP instantiations and interconnect, IP-XACT component descriptions with consolidated memory maps and port descriptions and all of the other auto-generated file types required for an IP block.

V. Conclusions

A part-automated IP design entry, generation and assembly flow is presented. Each step of the process is outlined, from register and pinout visual basic description through template toolkit auto-expansion of design, verification, IP-XACT and documentation files to the cataloguing and use of IP-XACT files in the design hierarchy. At each stage the judicious use of common templates reduces the development cycle and enforces a reusable IP structure.
The paper shows how IP-XACT descriptions can form a key part of an IP reuse methodology, but also that continued development beyond the IEEE 1685-2014 release of the standard is expected, leading to extra functionality and reducing the need for custom vendorExtensions.

VI. References

[1] IP-XACT Technical Committee. [Online] http://www.accellera.org/activities/committees/ip-xact.

[2] IEEE. *"1685-2009 IEEE standard for IP-XACT, Standard Structure, Packaging, Integrating, and Reusing IP within Tool Flows."*. Pascataway, New Jersey : s.n., 2010. ISBN 978-0-7381-6159-4 (PDF).

[3] Template Toolkit. [Online] http://www.template-toolkit.org.