Using Chassis JTAG BFM to address SoC Integration and Verification time

Shivaprashant, Bulusu (SRR2-1C-1693) shivaprashant.bulusu@intel.com
Sachan, Sunjiv (SRR2-1C-1694) sunjiv.sachan@intel.com
Wiznerowicz, Mike J (JF5-2-C11) mike.j.wiznerowicz@intel.com
Koay, Jun Hong (PG12-L2-C2) jun.hong.koay@intel.com

Abstract

Current SoC designs within Intel have large quantities of TAPs ranging between 70-170 TAPs that are arranged in a hierarchical network topology. Accessing a single TAP from a standard JTAG port requires a combination of Instruction Register (IR) shifts to store the opcode that points to the desired register and finally shifting data into and out of the Test Data Register (TDR) that is pointed to by the IR. Extending this principle to access a leaf level tap deep within a multi-level hierarchy from the Chip-Level TAP (CLTAP) requires programming each parent TAP that controls their sub-network. From a pre-silicon validation perspective if this process was composed of manual test writing then it would be tedious, error prone and unproductive. The Chassis JTAG BFM reduces TAP network validation complexity while providing the ability to maintain the same validation tests when TAPs move due to unexpected floorplan or architectural changes.

We defined a format in System Verilog that captures the SoC TAP hierarchy. This forms an input to the BFM. Chassis TAP tool (CTT) and collage output these files for BFM. In this paper CTT examples are illustrated to show the capability of the BFM. Several new API's were implemented to provide the validation test writer with a level of abstraction where tests can be written as if a single TAP is addressed. This was possible because the CTT performs topology checking that enforces unique names for each TAP in the entire SoC design. This greatly reduces fullchip test development time. The BFM also allows IP level sequences to be directly ported to fullchip without any modification. However, the BFM can only utilize the information that is entered or imported into the Chassis TAP tool. Users are encouraged to fully populate all fields of the TAP tool as accurately as possible for a richer set of validation collateral available today and potentially new capabilities developed in the future. This solution is applicable to all SoC designs that are compliant to Chassis DFx Gen1 or Gen2 standard.

1 Introduction

When IP blocks containing TAPs are integrated at SoC level with tools like Collage, the first step for the fullchip DFx validation engineer is to integrate a JTAG BFM into their verification environment. The second step is to validate that the TAPs are stitched according to the architectural requirements defined by the SoC integration team's DFx lead. Most often in the past this process takes weeks for projects to achieve. Chassis JTAG BFM addresses both of these issues. Since the BFM is architected with OVM methodology, the integration of the BFM into the validation environment is modular and independent of any underlying TAP network topology. The Chassis TAP Tool addresses to organize and assemble the topology from the SystemRDL files. The DFx lead constructs the SoC specific fabric topology with the Chassis TAP tool that is consistent with Chassis DFx architectural requirements then imports the SystemRDL files from the IP-blocks. Once the overall data set is in place then the tool can output the six System Verilog header files that are directly consumed by the BFM. The validation engineer is now ready to immediately direct targeted tests to validate the TAPs that are connected according to SoC architectural requirements.

In previous generation SoCs, providing an abstraction of the TAP network to a test writer was addressed by C++ APIs and integrating it with VCS through custom PLIs. This required a deeper knowledge about the parser and DPI2SV understanding to debug through custom language and inherent knowledge of integrating multiple flows (C++ compilation, integrating it in VCS, etc). However this solution was not modular. The current approach resolves all those concerns by implementing in SV/OVM and leveraging TAP SystemRDL standardization. In this instance, it is the advent of a standardized TAP machine readable format that did not exist before. Namely, TAP specific properties using the functional register SystemRDL language as a starting point. If all IPs delivered SystemRDL files, then the Chassis TAP Tool (CTT) becomes an aggregator and first line specification checker to reduce dependencies on BFM for nuisance bug discovery. For example, at the click of a button CTT is performing unique name checking, unique slave ID code checking and security inversion checking. This then provides the validation engineer greater time to focus on corner cases involving network connection issues.

The JTAG BFM requires information about the whole SoC network in System Verilog include files. The files contain information on several network attributes such as: unique enumerated TAP names, total number of TAPs, total number of hierarchies and tree structure connection information from root to leaf level. Other TAP specific data like Instruction Register width, number of opcodes, their width and associated data register length adds register access capabilities to the BFM.

2 Components of the Chassis JTAG BFM

2.1 New BFM features

Previous versions of this verification component supported standard TAP operations such as loading the Instruction and Data registers with known values, traversing the state machine, the ability to control resets, etc. However, these basic capabilities are inadequate to validate SoC TAP hierarchies. During the development phase of a SoC, TAP hierarchies may change several times due to adding or removing of clusters and IPs. The validation architecture must be nimble to re-align with the changes otherwise we are adding unnecessary effort to the project schedule. Limitations such as these require new thinking and a validation methodology that is aware of the network to make adjustments seamlessly to reduce burden on the validation engineer. Thus, BFM network awareness was architected in a manner to maintain backward compatibility to the existing features/APIs while providing new forward looking enhancements without additional effort in test writing. With the new BFM APIs, the validation team can address any TAP and any register at any level of hierarchy throughout the entire network as if the test writers had no knowledge of the structure of the underlying TAP hierarchy.

Figure 1 shows the standard components inside an OVM agent marked inside the boundary of the JTAG BFM package. To the left of it are the new extensions that are layered on top of the standard package to add network awareness while retaining backward compatibility. Together they are bundled and delivered as part of the standard collateral available in the Intel Reuse Repository. The BFM does not require any features from SAOLA to perform the TAP operations. Hence to keep it more generic it has been implemented using Industry standard methodologies and not tied to anything specific to Intel. Furthermore, this allows us to meet the requirements of Intel Custom Foundry (ICF) customers to not expose any proprietary methodologies specific to Intel.

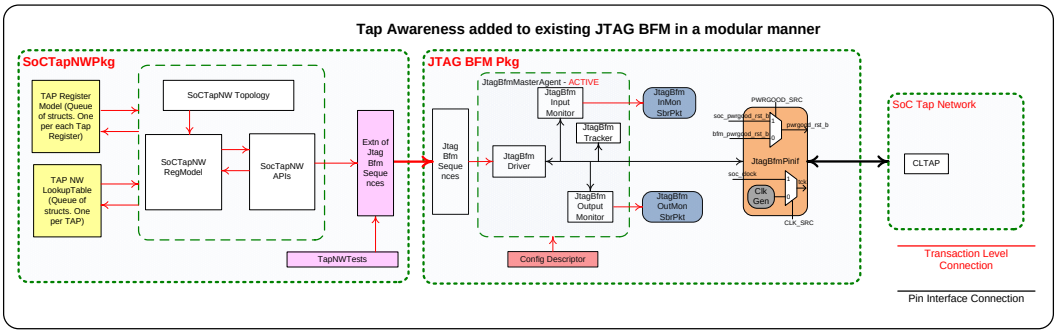

Figure 1. Tap Awareness extension to existing JTAG BFM

2.2 Class Hierarchy

Utilizing the class inheritance feature of the System Verilog language allowed the BFM to maintain its backward compatibility. The newer TAP Aware APIs extend the existing JtagBfmSeqeunces class. This class is a bundle of all the APIs that the BFM provides. In order to tell the BFM the various TAPs and the associated Test Data Registers (TDR) a class that extends the ovm_object was used. This is called the SoCTapNWRegModel that contains various members of different data types such as static System Verilog structures, arrays of queues, integers, helper functions that do basic search in these data structures and modify/read/update struct members with relevant data.

In the previous methodology, the test writer had to present a manual pre-calculated IR/DR vector to the API and the BFM would drive it during the ScanIR/DR states. This caused the test case writer to be aware of the active TAPs within the network hierarchy. This caused test case writing to be error prone and unproductive due to debugging one's test repeatedly. In the new paradigm, the data structures maintain an active transaction history and perform TAP network management tasks instead of the engineer doing it.
The construction of concatenated IR/DR vectors is done in the members of SoCTapNetwork package. Since this is at the test sequence level, the data structures parse the test intent captured by the APIs and store such information before sending to the JTAG driver. This is where the test intent is understood by the

BFM and why there is a layer of code in between these two components (sequence and driver) that must keep track of the TAP Network status. This includes how many TAPs are enabled on the network, how many are removed, etc. To keep track of these Network management tasks, data structures were defined in the Sequence class. The inherent TAP hierarchy is pre-populated into these structures during the run phase of OVM by having a standard API call (BuildTapDatabase). These structures have been designed to be static objects. Hence hierarchy of OVM sequences will be able to utilize the values in these structures.

Figure 2 shows the OVM class hierarchy. All the new features are present in a class that extends the JtagBfmSeqeunces class. Architecture definition of the TAP and TAP network are captured by engineers entering data on tools like the Chassis TAP Tool and Collage. Tools process the information provided and generates 6 files as shown in diagram. The six files generated are used by JTAG BFM to train and store information in history elements. Following paragraphs explains how hierarchy/TAPs information is used in the BFM.

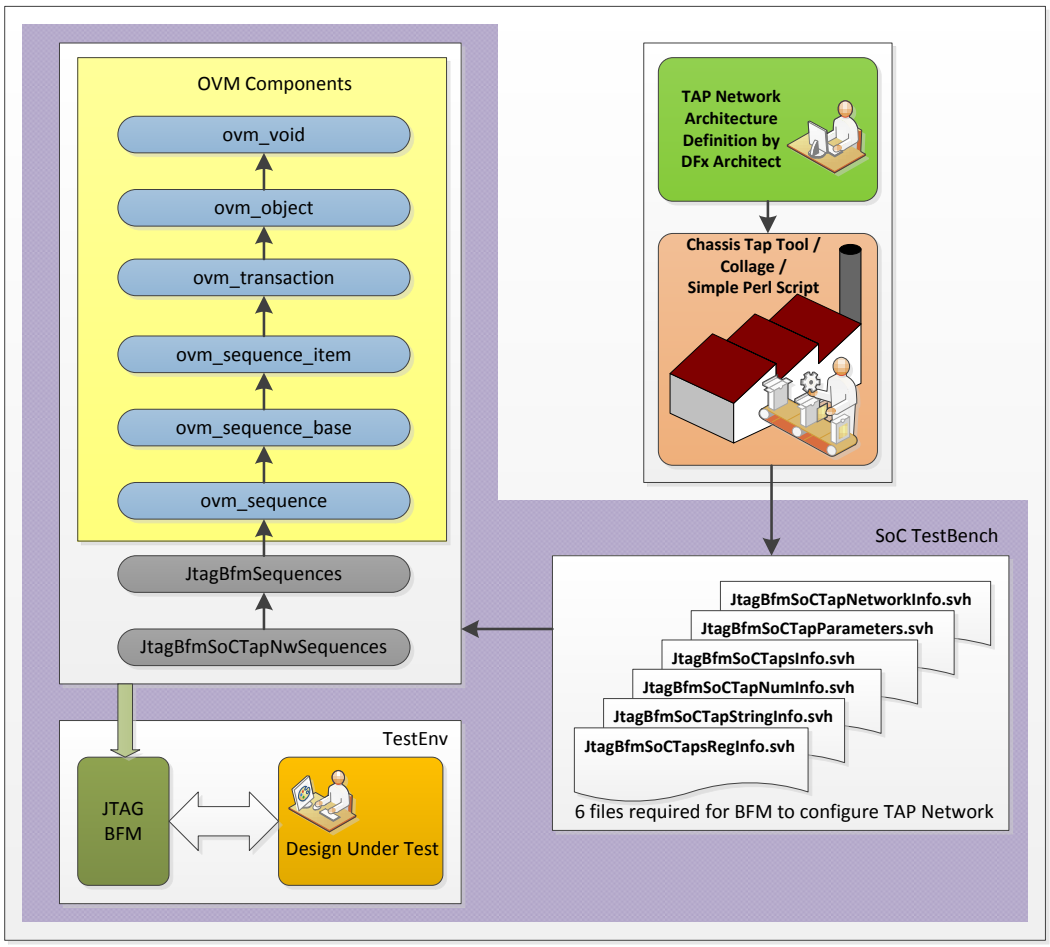

Figure 2. OVM Class hierarchy and files for BFM TAP network

2.3 Chassis TAP Tool

Originally, the Chassis TAP Tool was developed to provide the slave id codes (SLVIDCODE) that are applied to each TAP so that it can be uniquely identified in post-silicon. However, it was quickly realized that since the tool already contains all the information necessary to describe the network if it was extended to include register definitions then it can be offered to a wider audience. The casual user does not need to dig through countless specs to obtain useful information. Once the information is inputted into this tool the user can then direct the tool to output text files for various downstream tools to enhance productivity for pre-silicon integration and validation. The following files are currently supported:

- SIP JTAG TAP BFM files (six specific .svh files)
- SystemRDL files for Nebulon (<tap_name>.rdl)
- TAP RTL include files (<tap_name>.inc)
- Collage design files for the network TAPs (<project_name>.param and <tap_name.tdr>)
- TAP/TAP network attribute list output to various output formats (.txt, .csv, .xml)

We had a summer intern develop the foundation to edit and view test data registers. We developed our own proprietary set of SystemRDL UDP properties since none had existed before and we required extensive details to define all RTL parameters. The TAP tool is written in C# .NET using Microsoft Visual Studio.

The SIP TAP BFM .svh files were the first downstream validation tool supported since most of the network information was readily available and the content was well defined. We added register support later when the BFM required it. When we started on SystemRDL format for the registers using the tool it provided several advantages; one was the fast turn-around time in updating fields for Nebulon and another was implementing the parsing algorithm to import SystemRDL files back into the TAP tool to verify the overall process. We had a two phased plan with the first to implement just the registers to import and export files that are SystemRDL format for TAP. The next phase will include the TAP network information. After some stable SystemRDL files were available, members of the Nebulon team developed the Control Register Interchange Format (CRIF) register file descriptions and the SAOLA register validation collateral to fully test out the entire flow from the tool to working verification tests.

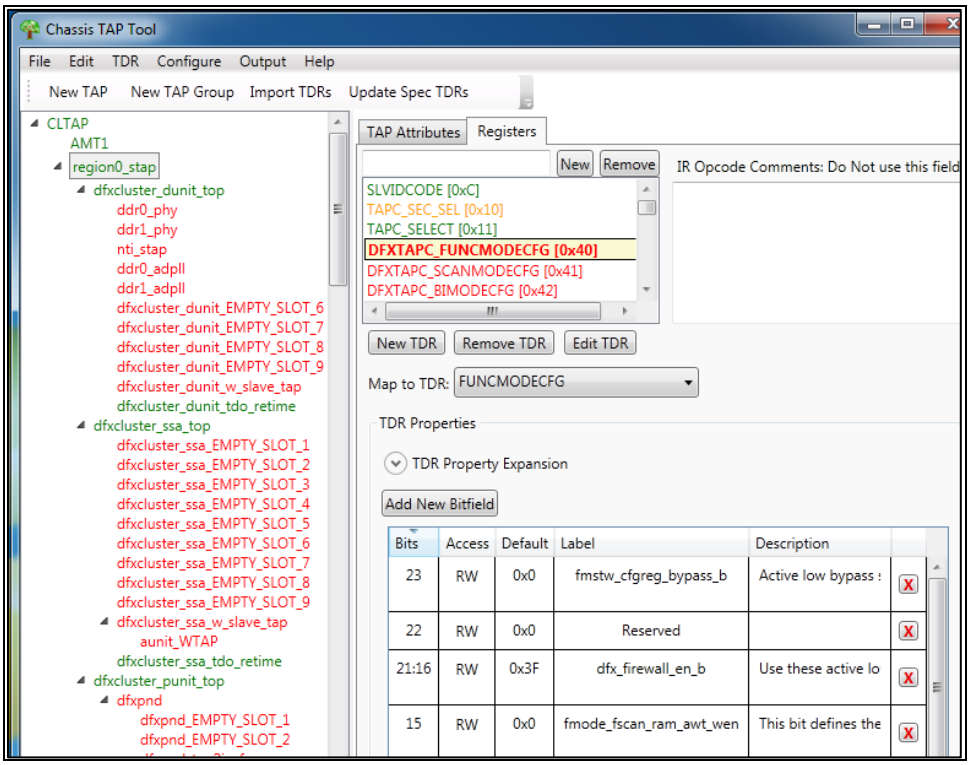

Figure 3. Screen shot of Cherryview TAP network in the Chassis TAP Tool

A screenshot of the tool is shown in Figure 3 is constructed with three primary panels to input the TAP information. A left side panel shows a hierarchy of TAPs in a familiar tree view representation that is similar to the Windows Explorer application to manage folders and files. One should notice that the TAP names and the IR opcode are color coded to match the security level. This gives the user multi-dimensional information about child TAPs, their security levels at the network level and individual TDR level. This immediate feedback to an expert user assists in the identification of potential issues even before the JTAG BFM verification collateral is ran on this topology. Additionally, since it is a graphical tool it allows easy organization of TAP topologies for derivative SoCs by simply dragging and dropping cluster TAPs with its children intact from one region to another.

The right panel is tabbed to switch between the TAP attribute panel and the register definitions for the selected TAP. TAP attributes define the slave ID code and parameterized features such as boundary scan or programmable reset support. For example, there is a checkbox for boundary scan support. If it is checked the tool will include the boundary scan registers compliant to the SoC TAP HAS with a single empty bit for the boundary scan register that the JTAG BFM can exercise as part of its register access APIs. The register panel is where the opcodes and register bits are manually inputted into the tool. Data entry is required for any new IP-block. it is more efficient for humans to work with organized data fields than to attempt parsing large text files manually. Manual data entry is only necessary for small updates once the majority of the registers are finally inputted into the tool. This tool supports exporting (creation of RDL) and

importing (assembling leaves to build a branch of TAPs) TAP SystemRDL. Once all the soft and hard IPs have entered their registers and delivered as part of the collateral then the SoC integration team can simply import all IP .rdl files and output the necessary JTAG BFM files at the click of a button.

Figure 4 shows the TAP tool's Output pull-down menu for the choices of output file formats that are available. Selection of "Print to file: TAP BFM verification files" will export the files based on all information available at the time of execution. Verification can begin in stages if the SoC team implements the TAP network first and then imports the registers later.

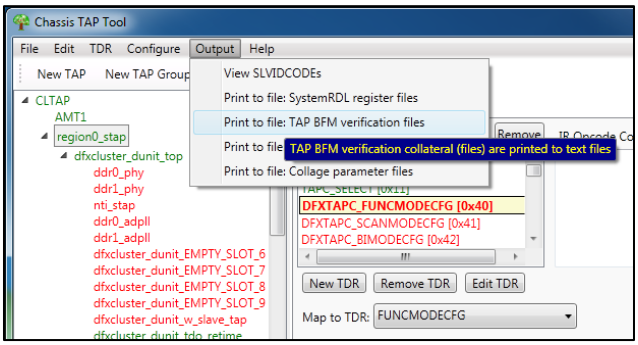

Figure 4. Chassis TAP Tool screenshot of BFM file generation

2.4 Files needed for TAP Awareness

Architecture definition of TAP-Network along with TAP details are captured by entering data in tools like Chassis TAP Tool or Collage. These tools process the network information provided and generates 6 files as shown in Figure 5. The six generated files are used by JTAG BFM to train and store information in history elements. These six files should be generated by every IP consumer of the BFM. The information regarding the underlying network will vary depending on the usage at IP standalone or cluster or partition or at fullchip level. The BFM can be located at a central place and the usage at each level can be governed by a specific set of six files that can pointed to by a predefined tick define that the BFM understands. This is documented more in the Integration Guide.

An example topology that explains the hierarchy is shown in Figure 3. CLTAP is at the root. It has two child TAPs namely STAP1 and STAP7. They both have two more child TAPs and finally, STAP3 has three child TAPs.

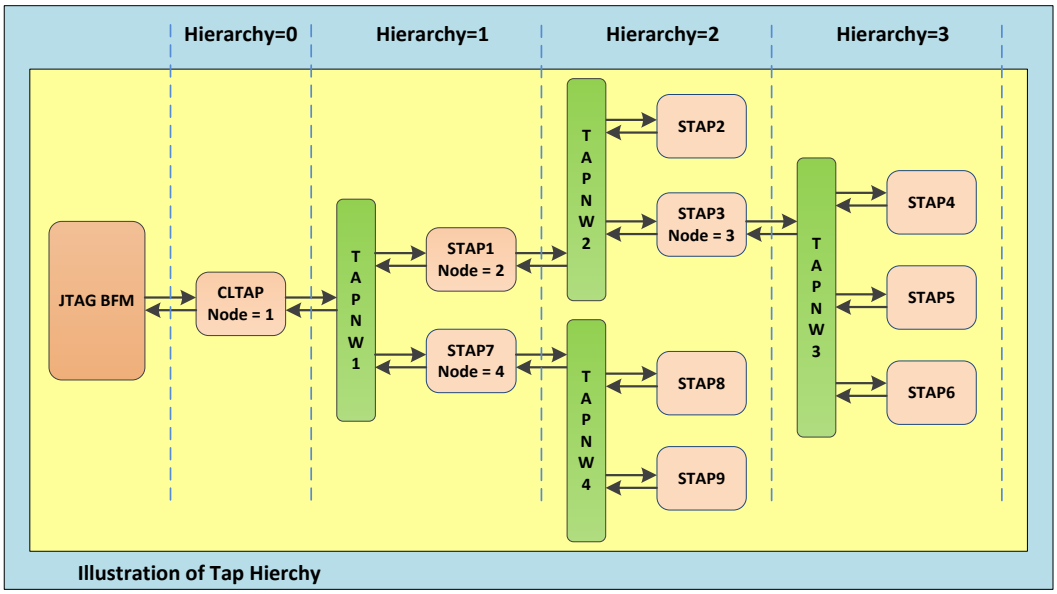

Figure 5. Simplified TAP Network Topology

Next we explain the contents of the various files. A code snippet of these is shown in Figure 6.

1. The file "JtagBfmSoCTapParameters.svh" contains parameters for the number of hierarchies, total number of Slave TAPs and the number of tertiary ports in the entire SoC TAP Topology.
2. The file "JtagBfmSoCTapsInfo.svh" contains a task that creates a lookup table containing various characteristics of the TAP attributes such as enumerated TAP name, length of IR, the Slave

IDcode along with the IR opcode used to read it out, whether it is a parent TAP that has children, the hierarchy level at which it exists, its position with respect to its parent's TAPC_SELECT register, etc. The task Create_TAP_LUT is executed when BuildTapDatabase() API is called in the run phase of the test. It then creates a queue of structs to store all this information which is retrieved when the various APIs in the test case are called.

3. The file "JtagBfmSoCTapRegInfo.svh" contains all the TDRs of every TAP and the width of each TDR. The task Create_Reg_Model is executed when BuildTapDatabase() API is called in the run phase of the test. It then creates a queue of structs to store all this information which is retrieved when the various APIs in the test case are called.
4. The file "JtagBfmSoCTapNetworkInfo.svh" contains the relation between a parent TAP and its child TAPs. A parent TAP will have a non-zero "Node" number. The node assignments can be done in any order as long as the numbers are unique. TAPs having no children below it will have a zero number assigned to the node.
5. The file "JtagBfmSoCTapNumInfo.svh" contains enumerated TAP names and the integer mapping for creating a typedef.
6. The file "JtagBfmSoCTapStringInfo.svh" contains the string mapping to the enumerated integer TAP name. This makes it easy to access a TAP by its enumerated name.

Figure 6. Code snapshot of six header files required for BFM

The fields of tasks in files JtagBfmSoCTapsInfo.svh and JtagBfmSoCTapRegInfo.svh are further explained here. Tap_SlvIDcode is the 32-bit vector generated by Chassis TAP Tool/collage. The same ID will be strapped in RTL. This is a very useful field to compare Slave ID read from RTL.This field is used in conjunction with Tap_VendorIDcode. Tap_SlvIDcode is used only when IsVendorTap field is set to 0. When the IsVendorTap field is a 1 then the user provides their own IDCODEcode. This field serves two purposes; the obvious one is when the TAP is from an external vendor and they only supply the IEEE1149.1 ID code and the other is to fix problems with the Slave ID code on projects that are using an older version of the TAP HAS. IR_Width indicates size of IR for TAP of Interest. This value can be programmed to any non-zero value as the BFM does not constrain this, however, for SoC TAP spec compliance this value should be between 8 and 16 bits. External Vendor TAPs will have a wider range so the BFM must be flexible. The Node filed is an enumeration assignment for the TAP parents. A non-zero number indicates presence of a TAP network IP and a zero value indicates it's a leaf TAP. For example, CLTAP hosts a network and the value of Node is 1. Similarly, values for STAP1, STAP3 and STAP7 are non-zero. There is no rule to assign particular numbers. But care should be taken to use the same numbers when network info file is created. SecondaryConnectionEnable indicates secondary and tertiary connections are available and can be connected to external pins in the SoC. By default CLTAP is enabled with secondary and hence its value is set to 1. HierarchyLevel field is used to represent the hierarchy level of each TAP with respect to CLTAP. Hierarchy level numbering starts with 0 and increments as each TAP is access linearly from the CLTAP to

the last leaf TAP. This is equivalent to finding the deepest node in a depth-first search algorithm. For example, CLTAP has value of 0. Similarly, STAP1 and STAP7 have a value of 1, STAP2, STAP3, STAP8 and STAP9 have values of 2. Also, note there is only one network at hierarchy level 3, i.e., STAP4, STAP5 and STAP6.

PostionOfTapInTapcSelReg: This field is very important as it specifies position of the TAP within TAPCSEL register. The value of '0' indicates LSB position and increments thereafter. For example, STAP4 is assigned with the value of 0, STAP5 assigned with the value of 1 and STAP6 is assigned with the value of 2. This is applicable to all the TAPs in any given TAPNW. IsVendorTap: This field is used in conjunction with VendorIdOpcode. The value of '1' indicates IDCODE is imported externally and the internally generated IDCODE is not used. VendorIdOpcode: This field is used in conjunction with Tap_VendorIDcode. From the task Create Reg Model, the field Tap ID indicates the TAP of Interest that all arguments are intended.Example, CLTAP, STAP1, etc. Opcode: A hexadecimal value indicates the presence of a Test Data Register in a given TAP. For example, CLTAP has opcodes like 8'hFF, 8'h2, 8'h5, etc. DR_Width: This field indicates the width of the register corresponding to the OPCODE of interest. For example, 8'hFF is the BYPASS opcode and its width is 1.

2.5 Dynamic Network awareness and History Element

During the OVM run phase, the first API that has to be executed is the BuildTapDatabase. This call allocates system memory to create static objects and fill them with the underlying hierarchy of TAPs coming from the six files that are generated for the BFM using tools like Collage or Chassis TAP Tool. These objects are data structures that constitute what we call in the BFM terminology as "History Element". These elements get dynamically updated based on the API calls that the user makes. Also this element is common and shared across multiple instances of the JtagBfm. For example, if a SoC has Primary, Secondary and two Tertiary JTAG ports, then there will be four instances of the JtagBfm in the testbench. All four will share this History Element. This allows keeping track of topology changes dynamically when a TAP is moved to the Tertiary or Secondary JTAG ports. The history element assists the BFM to inject only the cycles necessary for the transaction. For example, subsequent accesses to the same TAP and to the same TDR on the network will skip the IR scan path of the TAP FSM thus saving validation time. This will be extremely helpful when a particular register is polled before taking any further action.

```
typedef struct {
   Tap_t       TapOfInterest;
   Tap_t       ParentTap;
   int         HierarchyLevel;
   Tap_t       NodeQueue [$]; // Contains all the parent TAPs of a given TAP.
   int         DrLengthOfTapcSelectReg;
   int         PostionOfTapInTapcSelReg;
   bit [1:0]   TapMode; // SoC TAPNW Modes.
   int         IsTapEnabled;
   int         IsTapDisabled;
   int         IsTapShadowed;
   bit [255:0] TapcSelectRegister;
   bit [127:0] TapSecondarySelect;
   int         TapBeforeTapOfInterestArray [];
   int         TapAfterTapOfInterestArray  [];
   int         TapBeforeTapOfInterestQueue [$];
   int         TapAfterTapOfInterestQueue  [$];
   int         ChildTapsArray []; // List of all the Childs for a given TAP. This info is made for every TAP in the design.
   int         IsTapOnSecondary;
   int         IsTapOnPrimary;
   int         IsTapOnRemove;
   bit [API_SIZE_OF_IR_REG-1:0] PreviousAccessedIr;
   bit         IsTapOnTertiary[NUMBER_OF_TERTIARY_PORTS-1:0]; // Indicates on which Tertiary port a given Tap is placed.
   Tap_t       TertiaryParentTap; // Need a function to update this value from CTT's SecondaryConnectionEnable.
   int         IsVendorTap;

} HistoryElements;
```

Figure 7. Code snapshot of the elements of History Table

TAP Aware APIs heavily use the history element. APIs are implemented using queues with each queue updated depending on the value of the history elements. For instance, referring to the example in Figure 5, when STAP2 is accessed, it involves enabling STAP1. As soon as STAP1 is enabled, the history element is updated to reflect this enable. When the concatenated IR/DR chain is constructed for the TAP of interest, the BFM will refer to the history table and concatenates appropriate values for enabled TAPs. Any disabled TAPS are excluded from the active queues. Active queues are updated upon each access for all the APIs. A portion of the history element is shown in Figure 7.

2.6 Sequence Porting from IP level to SoC

One of the key features that the BFM enables is the as is reuse of IP level sequences at SoC level. This is possible due to the enumerated TAP names that are used to address each TAP. Therefore, as long as the same enumerated TAP name that is used at the IP level is used at the SoC level, the sequence can be reused. Table 1 shows how an IP level sequence can be reused at SoC.

This BFM is integrated in the Chassis Sandbox verification environment and assuming this continues into the future there will be SoC to SoC porting that can be leveraged due to the architectural consistency and standardization across Regional/Cluster DFx units. For reusing the IP level sequences at SoC, the IP provider has to make the reusable sequence part of ip_pkg_lib and export it through ACE. SoC in their test

cases will make an instance of the IP sequence and plays it on the JTAGBFM sequencer declared at SoC level. First, this avoids copying of the IP level sequences in the SoC sequence library. Second, this avoids adding a preamble or a postamble to the IP level sequence.

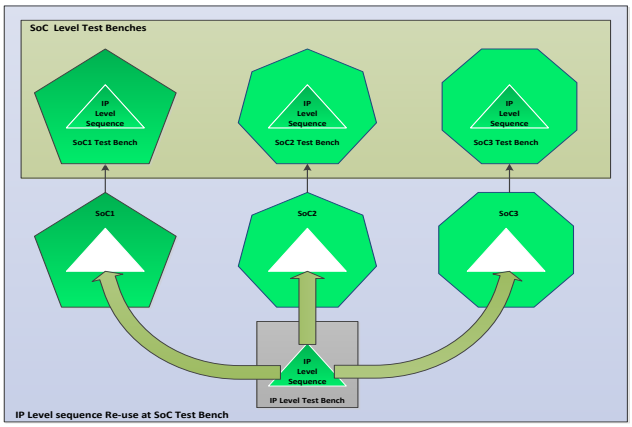

Figure 8. IP level Sequence Re-use at SoC level.

IP Provider – CTT files will point to indicate only single TAP in the hierarchy.	**IP Consumer -- CTT files will point to indicate the entire SoC TAP hierarchy with "IP_STAP" enumeration type indicating a leaf level TAP.**

`//-----------------------------------------------------------------`
`// Seqeunce Class containing the usage of BFM API's`
`class IP_level_TapSeq extends JtagBfmSoCTapNwSequences;`
`...`
`task body();`
`...`
`TapAccessSlaveIdcode(IP_STAP);`
`...`
`endtask`
`...`
`endclass : IP_level_TapSeq`
 	
`//-----------------------------------------------------------------`
`// Above class added to Env Pkg`
`package IP_EnvPkg;`
`...`
`include "IP_level_TapSeq.svh"`
`...`
`endpackage`
 	
`//-----------------------------------------------------------------`
`// Env lib exported thru ace ip_hdl.udf`
`...`
`models => {`
`ip_model => {`
`export => {`
`libs => [ "ip_env_lib", "ip_rtl_lib" ],`
`},`
`...`
	`//-----------------------------------------------------------------`
	`// SoC imports and uses the IP lib in soc_hdl.udf`
	`models => {`
	`soc_model => {`
	`import => {`
	`libs => [ "ip_env_lib", "ip_rtl_lib" ],`
	`},`
	`...`
	
	`//-----------------------------------------------------------------`
	`// SoC adds IP to thier package`
	`package Soc_EnvPkg;`
	`...`
	`import IP_EnvPkg::*;`
	`...`
	`endpackage`
	
	`//-----------------------------------------------------------------`
	`// SoC Sequence Class using containing an instance of IP seq`
	`import Soc_EnvPkg::*;`
	`class SoC_level_TapSeq extends JtagBfmSoCTapNwSequences;`
	`...`
	`IP_level_TapSeq inst_IP_level_TapSeq = new();`
	`task body();`
	`...`
	`inst_IP_level_TapSeq.start(SocEnv.inst_JtagBfmAgent.Sequencer);`
	`...`
	`endtask`
	`...`
	`endclass : SoC_level_TapSeq`

Table 1. Code snippets of how an IP sequence can be reused at SoC

Multiple instantiation of the same IP throws a challenge in addressing content reuse from IP to SoC. For example let's say there are 4 instances of PCI Express cluster at fullchip. Let's assume the IP provider will provide the enumerated name – PCIE_STAP as a parameter as he may not know how many instances will be used at fullchip. Now let's say SoC uses four instances and chooses to have PCIe_STAP1, PCIe_STAP2, PCIe_STAP3, PCIe_STAP4 as the enumerated names in fullchip. Further topology checks in CTT prevent the same name appearing multiple times. Now when test content to any one of the TAPs is addressed, then it's the prerogative of the IP provider to provide parameterizable sequences so that the name can be overridden. This allows content reuse across instance.

3 Results

Figure 9 shows a complex SoC TAP hierarchy as a test case to cover nearly all possible TAP topology arrangements. The root node of the hierarchy is Chip Level TAP controller. It has TAP Network IP attached that hosts TAPs on the first level of hierarchy. These typically would be the Regional DFx unit TAPs. Here STAP1 and STAP20 are parent TAPs that host a sub-network. This extends down until all the TAPs are covered in the SoC. Here STAP17 is a leaf level TAP that is of interest for this discussion. In order to access a TAP register in this TAP, STAP1, STAP8, STAP13 have to be configured to be Normal with the BFM. Normal is a term used by the IEEE1149.7 T0 spec which this is based upon and defines a TAP that is in series with the master node controlling it. In terms of the SoC TAP HAS it is a parent that is in series with one of its children. This requires many IR/DR scans to program each Select register throughout the hierarchy by using the LoadIR/LoadDR APIs of the BFM.

With the introduction of TapAccess API, any register of a TAP can be accessed just as if it is being addressed directly. The key here is to provide the TapName argument. The BFM then enables all the TAPs in the path to the TAP of interest. To access STAP17 in hierarchy level 4, only one line of code is required (single API used once) if using the new TapAccess API instead of the using MultipleTapRegiserAccess five times the old way (one API call for each level of hierarchy).

Several APIs such as EnableTap, DisableTap, ShadowTap, and RemoveTap MultiTap were added that perform dynamic network management functionality where the TapName is the key argument that they take. The enhanced capability of the BFM was successfully deployed first in Cherryview SoC. Their effort in discovering bugs, issues in deployment and enhancement requests have made the BFM robust. It took a total of two weeks' time to generate/maintain the CTT xml file for the network, familiarize with the new BFM APIs and work through all the various bugs of the BFM. The auto setup of a leaf TAP onto the TAP network provided by the APIs saved two weeks of test writers' time from writing mundane/trivial code to program TAPC_SELECT registers to bring up a TAP. The BFM shields test writers from any changes in the TAP configuration change such as TAP relocation and addition/deletion of TAPs which is common in any SoC. The effort savings from this is estimated at two weeks. The same test content could be easily reused to access TAPs on the secondary TAP network through a simple configuration API thus saving an estimated week of effort. In total there is a net savings of two weeks of fullchip validation time for the lead project. Subsequent projects will see about four weeks of savings as a result of this BFM usage.

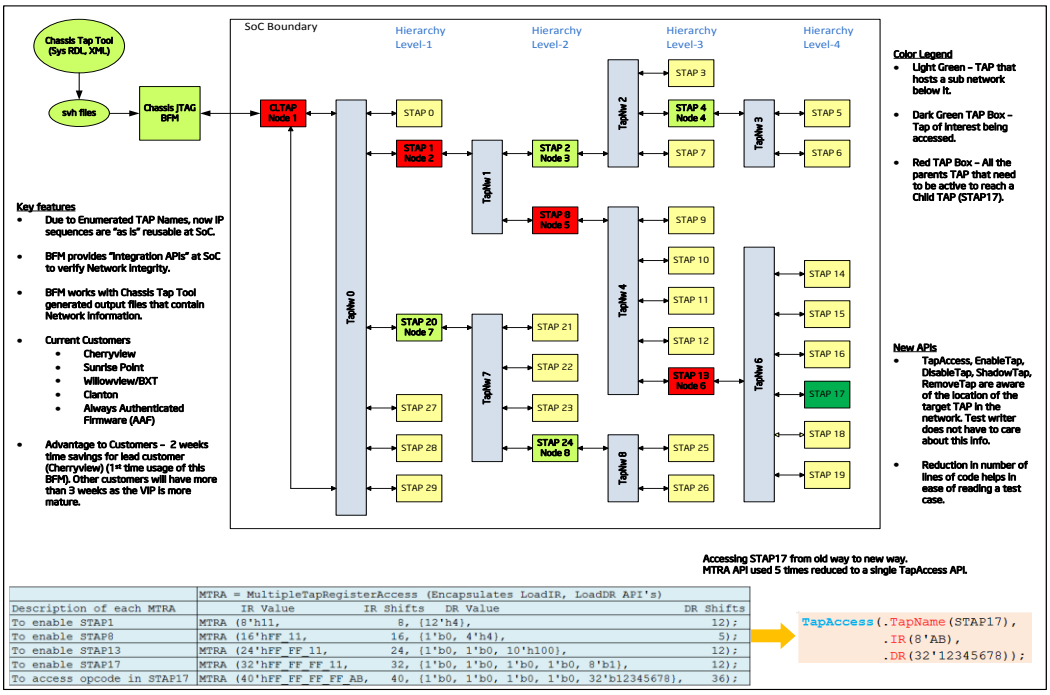

Figure 9. Example view of SoC TAP Hierarchy

The amount of time gained here is quantified for the usage of the BFM for TAP network validation. More substantial benefits will be gained if the BFM is used for porting of IP level validation tests to fullchip level for future projects. The modular approach of the BFM eased integration in the Broadwell-DE environment. In five hours they could get the first test to pass.

Next, Table2 outlines the key metrics between the original version of the BFM used in Berryville SoC of 2009 and the enhancements done to the BFM used in 2012 SoC- Cherryview.

Metrics	Old BFM usage in Berryville (2009 SoC)	Enhancements done to BFM described in this paper used in Cherryview (2012 SoC)	Remark
BFM Integration time	2 days	1/2 day	Usage of Ace reuse methodology for integration.
Setup/enable of all TAPs on the network	2 weeks	0	API provides the needed abstraction. In old methodology, each user must study the TAP Network topology in order to program the TAPC_SELECT registers properly to enable a TAP of interest. For the new BFM, the user just has to know the assigned name of the targeted sTAP and use EnableTap() API.
Re-setup of a leaf TAPs due to changes in SoC hierarchy.	> 1 week	1 day	This step depends on how many times changes are introduced. In the older methodology, there was a need for all the tests to be rewritten if there were changes to the network topology. Now, by using CTT, the user just needs to change the single XML and regenerate the BFM files using the CTT.
Familiarize with the new BFM APIs	0	3 days	Adequate training material is available such as Integration guides, BKM documentation, training videos, etc. This excludes the additional time required due to bugs discovered by lead customers.
generate/maintain the CTT	1 day	1 day	One day is for entering TAP network info. Will need more time to enter register information of each TAP. In the older methodology where the CTT was not yet introduced, the same amount of time was needed to enter register and TAP network info into HAS documents instead of CTT tool.
TCK saving due to IR optimization	Explicit	Inherent	The BFM automatically address this through the built in data structures.
Network awareness	Explicit	Inherent	The BFM automatically address this through the built in data structures.
Test reuse across IP to SoC & SoC to SoC	No	Yes	BFM absorbs the hierarchy information due to enumerated TAP names and hence makes it possible for reuse.
Portability of tests from primary TAP Network to secondary/tertiary TAP Network	No	Yes	PutTapOnSecondary () and PutTapOnTertiary () APIs enable straight-forward porting of test content from primary to secondary/tertiary JTAG port access.
Code readability	Harder	Easier	Due to the addressing of each TAP by a unique name, writing code using the new BFM is less error-prone and the code is more readable by others. Better readability also means it is easier to debug the test codes.
Auto-generation of test content	No	Yes	IDCODE read test for all TAPs in Cherryview SoC is auto-generated using a simple Perl script. Back in Berryville, all the TAP IDCODE read tests were developed manually and involved a few weeks of time to complete.
Total time estimate	Approx. 4 weeks	1 week	There is easily about 3 weeks of savings as a result of the usage of the new capabilities in the BFM.

Table 2. BFM net resource savings summary

4 Summary

TAP test content development is made independent of the hierarchy in which the TAPs are connected on the network at fullchip. The TAP Network aware APIs provided a layer of abstraction that hides any changes of the TAP network topology from the test writer. This feature significantly saved time of the test writers for maintaining their tests whenever there was late design change. From recent Cherryview SoC experience, there have been several TAP network topology changes due to floor plan changes. By using the TAP Network aware APIs in this BFM, test writers did not have to modify their existing tests or spend time debugging on test failures. The key benefit provided to SoC Integration teams is the APIs that validate the Tap Network Architecture by reading all the IDCODE of all TAPs. Further, test content was reused between primary/secondary/tertiary JTAG ports. This resulted in a net benefit of 3 weeks of validation time savings.

The BFM coupled with support from tools like Collage and Chassis Tap tool provides a significant enhanced TAP validation strategy that can be reused across Intel products.

Acknowledgments

The authors would like to acknowledge several members

Khan, Husnara - Validation Lead for accepting this development into the Cherryview project being the lead customer. Her team provided valuable feedback and caught bugs to make this IP more mature and stable.

Velikandanathan, Balaji - for accepting this development into the SunrisePoint project.

Kulkarni, Kishor from CSG-SIP Team for helping in debugging some of the critical elements related to static class objects where few bugs were found by the lead Customer - Cherryview.

Molchanov, Igor V from the CCDO Team for reviewing the BFM features and suggesting to add the MultiTap capability.

Mishra, Mukesh K for enhancing the capability of Collage to generate the required files needed for the BFM for the Chassis Sandbox framework.

Lange, Pinchas from the Tangier/Broxton team to provide recommendation on most widely used features at SoC Level.

Bibliography/References

[1] “IEEE P1149.7™/D1.21 Draft Standard for Reduced-pin and Enhanced-functionality Test Access Port and Boundary Scan Architecture”, IEEE P1149.7™/D1.21, February 2009

[2] SOC JTAG TAP High Level Architectural Spec – Rev 090 by Mike Wiznerowicz

[3] Bulusu, Shivaprashant, et al, “Configurable and Modular IEEE1149.1/7 Test Access Port Controller Soft IP” – Proceeding DTTC 2011

[4] IEEE Std 1800 – 2009 Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language