Shorten and Simplify SoC Verification using a Generic eVC

David Robinson, Verilab
E: david.robinson@verilab.com

Abstract: This paper describes an e Verification Component (eVC) that has been used in the verification of four separate bus based SoCs with only minimal modification. The user simply describes the topology of the design and testbench to the eVC, and the eVC uses this information to instantiate the correct bus eVCs, to create the required scoreboards, to generate customised functional coverage, to control the default stimuli so that only realistic transactions are generated, and to perform full bus infrastructure verification.

Individual testbenches can define a working topology to reduce the scope of the testbench, improve performance, and reduce third party license usage. The eVC greatly increases the user's productivity by automating many of the trivial but time-consuming aspects of an SoC testbench. It has proven to be so generic that certain tests can be moved between SoCs *without modification*. The "self healing" nature of the testbench makes mid-project design changes trivial to deal with.

Once a design's bus infrastructure has been verified, the eVC provides an advanced platform on which to build the rest of the testbench.

Introduction

IT IS A TIME CONSUMING TASK TO PREPARE A testbench to verify an SoC. Integrating all of the required eVCs, connecting them to the design and making them play together is complex enough, but that's just the start. The bus eVCs required to monitor the bus infrastructure of the design need to be instantiated and configured. Not only will you need to know several bus protocols and their corresponding bus eVCs, you will also need to know their quirky ways - those unexpected little features that make life so interesting.

Once the bus eVCs are in place, you will need to write bus scoreboards, functional coverage groups and stimuli for the SoC. Those buses and components that do not deal with the full bus protocol will have to be dealt with as special cases. You will need to integrate all of the register sequences that are required to control the peripherals, instantiate eVCs in the testbench to communicate with the design, build a system-wide address map and so on.

Of course, such a testbench may be slow and memory hungry due to the number of eVCs loaded for the buses and components, and it may also be license hungry. Those bus and peripheral eVCs don't always come for free. A common solution to this is to build several additional testbenches, each dealing with a sub-system of the design.

Once you have all of this in place, it can be reasonably assumed that one or more of the following will occur:

- Someone will want to use one of the testbenches to functionally verify a component
- Someone will change the specification, and you'll find components moving to new buses, bus protocols changing, bus protocol limitations appearing, new peripherals added, etc.
- Someone will want to use your testbenches on a derivative or next-generation design, when "all" that is changed is the address map, the number and types of the buses, the position and number of components, and the number of bus interfaces

This paper reports on a generic SoC eVC we have created for a customer that takes some pain out of SoC verification. It provides near automatic bus infrastructure and bus connectivity verification for any bus-based design, and provides a platform to be used for the remaining verification.

Background

Despite the fact that all SoC designs are unique, when viewed in a certain way many start to look suspiciously similar. Most SoCs are bus based, and can be viewed as a connected graph of buses, components, bus interfaces and bridges. Slave bus interfaces will have at least one address range, and there is a high probability that some buses and bus interfaces will not support the full bus protocol. For instance, RETRY responses may not be supported, and instruction fetch buses won't have write capabilities.

In addition to the structure of the SoC, the verification process for different designs is similar. One of the first tasks in verifying such an eVC is to comprehensively check the bus infrastructure using constrained random stimuli. If the verification environment is set up correctly, then these tests can also be used to verify the connectivity between the bus and the bus interfaces [1]. Directed or semi-directed tests will be needed to sanity check a particular component, to do use-case testing or to generate characterisation patterns.

The requirements of a verification environment for an SoC are also similar between disparate designs. The verification environment(s) you create for a design will be expected to deal with both the entire SoC, sub-systems and possibly even individual components. You can expect to be asked to make it work on derivative designs and the brand new architecture - WhizzoSoC2005 - when it arrives.

This is exactly the situation we found ourselves in a few projects ago, and we decided to do something about it. Starting with the following requirements, we built a generic SoC eVC that jump starts our customer's SoC verification projects. We decided that the new eVC must:

- automatically perform random bus infrastructure verification of the design
- be usable for directed and constrained random verification of individual modules, sub systems and the entire design
- allow the design's topology to be easily described
- use the topology description to automatically generate scoreboards, coverage files and stimuli, and to instantiate and configure the eVCs required to deal with the design's busses and bridges
- deal with buses and bus interfaces that do not support all protocol features
- deal with any changes to the design's topology occurring during the project
- be able to control the loading of component eVCs to increase performance and reduce license usage
- be able to control the working topology of the design. By this we mean that each testbench using the eVC should be able to specify which subsystems are or are not included in the testbench, and which components are black boxed in the RTL to allow bus-infrastructure testing of them
- as much as possible, isolate the test writers from changes to the specification, the RTL or the eVC itself

Key Concepts

Before we discuss what the eVC actually does, there are some key concepts that need to be understood:

- The topologies
- The Topology Database
- The APIs

- Bus protocol limitations

Design, Environment and Working Topology

It is commonly assumed that the topology of a design is fixed and constant. In reality, this is not always true. The topology of an SoC design can change in two ways. One change in topology is between designs in the same family. For instance, between WhizzoSoC2004 and WhizzoSoC2005, or between WhizzoSoC2004 and WhizzoSoC2004-LowPower, the topology will change even though much of the overall functionality remains the same. If we can somehow encapsulate the changes, we can get some reuse from the testbenches used to verify earlier members of the family.

A design's topology can also change within its development cycle. We do not mean that the specification changes[1], but instead that we define our own topologies based on the verification task at hand. At one end of the scale we want a testbench that deals with just a single peripheral, and at the other, a testbench that deals with all peripherals, all buses, and all system wide features such as debug, test and clocking. For speed, memory and license reasons, we normally desire a selection of testbenches that deal with different topologies within these bounds. For instance, if we are trying to verify the clocks on the AHB_COM_BUS1 sub-system, why include the multi-layer AHB bus, the DSP sub-system, the other peripheral buses and the AXI buses?

The topology seen by any particular testbench is known as the **working topology**. This is simply a subset of the full topology. By programmatically altering the design topology that the eVC can see, we can use the same eVC to implement all of the smaller testbenches normally required for an SoC. For instance, the working topology can be used to reduce the visible design to a single sub-system, or a cluster of sub-systems, for the testbench that is charged with verifying them.

The working topology does more than just change the number of design items seen by the eVC. It is also used to control the black boxing of HDL components. When we do bus infrastructure testing and bus interconnectivity testing, we need to generate random bus traffic. We cannot do this if the HDL for the components attached to the buses are present. Not only can we not control the stimuli from the masters and the responses from the slaves, but writing random data to a random address in an SoC can quickly cause a variety of interesting simulation failures. For instance, there's a good chance you'll switch the clocks off or put the design into reset.

The approach we use to deal with this is called the skeleton approach [2]. HDL masters and slaves on the bus or buses we are testing are black boxed and replaced with active verification models. This is only required for certain testbenches, so this information forms part of the testbench specific working topology.

As well as reducing the number of separate testbenches that need to be created for an SoC, the working topology also improves the performance of the testbench by allowing us to remove surplus HDL and verification components. This also reduces the number of licenses required for a simulation.

Of course, the overall topology seen by a simulation does not just include the design. Testbench components also form part of the topology. For instance, if your SoC is designed to be a slave on an AHB bus, then your testbench will need to have a model of an AHB bus and an AHB master as well. The eVC has to know about these so it can control stimuli generation from the master, implement a scoreboard between the master and the SoC slave interface, generate the coverage groups, etc. We therefore split the idea of a topology into a design and environment component. The eVC itself makes no distinction between these - it just sees a large SoC connected to other components, but the topology information is captured separately to allow easy reuse of the eVC in different testbenches.

[1] although that happens too!

Figure 1 shows an example topology for a design. This simple SoC has two buses with different protocols, two primary masters (CPU and DMA), and both the DMA and the Bridge components have multiple bus interfaces. All of this information is given to the eVC so it can build its view of the design.

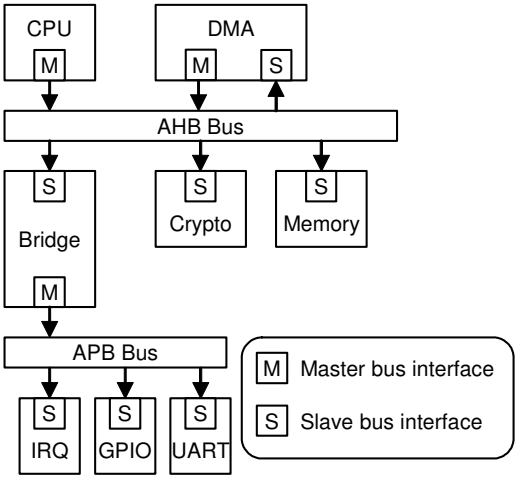

Figure 1: The design topology

In Figure 2, we can see an example of the working topology in action. This working topology is for a bus infrastructure test of the AHB bus sub-system. The entire APB bus is removed from the topology, and all of the masters and slaves on the AHB bus are black boxed in the RTL. The SoC eVC instantiates active agents on the AHB bus interfaces to inject random transactions and responses.

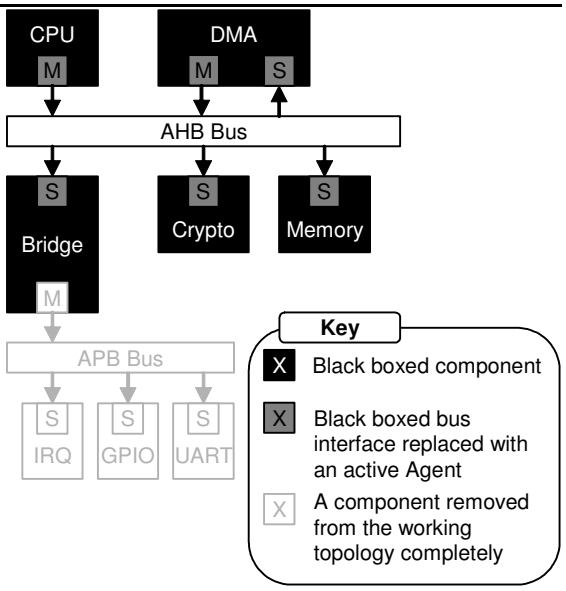

Figure 2: The working topology for bus infrastructure testing of the AHB sub-system

Figure 3 shows a more complex example of a working topology. This one has been set up to do a use-case performance analysis. The DMA controller transfers data from the memory to the Crypto unit and then from there to the UART for off-chip transmission. The transfer is controlled by interrupts. Both subsystems are needed for this (the AHB and the APB), and the RTL for the DMA, the bridge, the IRQ and the UART are needed, so they are included in the working topology (they are not black boxed). The master and slave interfaces of these components have passive agents instantiated to allow us to monitor the bus traffic and to do functional coverage.

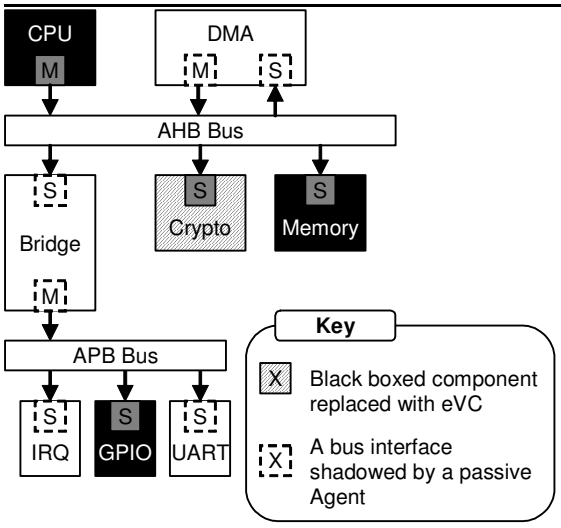

Figure 3: The working topology for a specific use-case

The Crypto component is treated differently. We want to include its functionality in this simulation, but we don't want the speed penalty of letting the RTL perform the functionality. What we have done here is to black box the RTL for the Crypto component, and load up an eVC in active mode to get its functionality at a behavioural level [1].

Topology Database

Most of the above information is stored in a database in the eVC, called the Topology Database, which forms the heart of the eVC. Almost everything else in the eVC either uses or supports the database. The information in the database forms a graph which tells us what "things" have in the design, what attributes these "things" support, and how these "things" are linked to each other. "Things" are either buses, components or bus interfaces.

Buses form the communication channels in the database. The database contains a list of bus descriptors, and each descriptor contains information such as the name of the bus, its protocol, whether it is multi-layer or not, its routing table if it is, a list of components attached to the bus, whether it is black boxed and the protocol features the bus doesn't support. Each bus entry has an associated bus eVC instantiated in the testbench (assuming it is in the working topology).

Components are used to group bus interfaces together. The database contains a list of component descriptors, and each descriptor contains information such as the name of the component, whether it represents a peripheral or a bridge, a list of all the bus interfaces the component has, and whether it is black boxed.

Bus interfaces are the main element in the Topology Database. A bus interface entry must be part of a component, and it describes one of the component's bus interfaces. These can currently be masters or slaves. A component can have many bus interfaces on many different buses. Each bus interface entry has an associated agent instantiated in the testbench (assuming it is in the working topology). A bus interface entry contains information such as its name, the kind of interface (master or slave), the name and protocol of the bus it connects, the features of the protocol that it doesn't support, and whether it is black boxed.

Slave bus interfaces store the slave's address ranges, and master entries store the register sequence driver for the master. Specific information relevant to the corresponding agent can also be stored in a protocol extension of the entry.

When the eVC is ported to a new design, someone has to fill in the Topology Database to describe the structure of the design. Once this is done though, almost everything else, including certain tests, will work automatically on the new design.

The APIs

The eVC has two APIs (Application Programming Interfaces) which are used to both make the eVC easy to use and to provide a layer of encapsulation against changes. The Test API is provided exclusively for test writers. This provides named access to the various sequence drivers in

the eVC. For instance, a test writer can make the system CPU perform a sequence without knowing where the CPU is in the design, or what bus protocol it uses:

```
do seq keeping{
  .driver == driver.test_api.
         get_bus_sequence_driver(CPU);

  // Other constraints removed for clarity

};
```

If the SoC's topology changes, and the CPU moves to another bus, or if it moves off chip, or if it changes bus protocol, then this test will continue to work *without modification.* As long as there is a master bus interface called CPU in the topology, no modification have to be made. A factory pattern is used to create the correct type of master sequence.

The Topology API provides an interface to the Topology Database. The Topology API exports many different methods, some of which are heavily used by the eVC itself, and some of which are of great use to test writers. The Topology API is extendable, so new methods can be added as required. Some examples of methods are:

- **get_active_flag**: This method tells you whether a bus interface is active of passive
- **get_address_for_slave**: This method returns a random address belonging to the named slave
- **is_bus_interface_black_boxed**: This method returns TRUE if the specified bus interface is black boxed
- **get_all_reachable_slaves**: This method returns all slave that can be reached by the specified master bus interface. This method searches through bridges, so it returns every slave in the entire topology that can be reached by this master. Routing tables on multi-layer buses are also taken into consideration. Complex filters can be applied to restrict the results to slaves that meet certain criteria, such as "only return black boxed slaves on buses with the AHB and AXI protocols"

Bus Protocol Limitations

One other thing that the eVC needs to know about is to what degree the buses and bus interfaces in the design implement their bus protocols. For example, there is normally not much reason for the CPU instruction interface to support write accesses. You'll find that certain masters in the design can only generate 32-bit transfers or bursts of a certain type, and that slaves do not support responses other than OK or ERROR. These limitations may just apply to individual bus interfaces, or they may apply to the entire bus.

The user of the eVC can capture this information when he defines the topology of the design. The information is stored using the types that the underlying bus eVCs will use.

The eVC

Now we know what information the eVC has about the design and testbench, what does it do with this information? Well, it does quite a lot actually:

- It automatically instantiates and configures the eVCs needed to verify the design
- It automatically creates and connects the scoreboards required for bus infrastructure testing
- It creates the functional coverage required for bus infrastructure testing, taking the protocol limitations and design connectivity into account
- It controls the random bus traffic generated, taking the protocol limitations and design connectivity into account
- It uses the information to verify the design's bus infrastructure
- It sets up the register and memory maps for the design

These are all tasks that are otherwise time consuming and cause problems when the design changes.

Scoreboards

The SoC eVC provides automatic scoreboarding for all bus traffic. To reduce the complexity of the code required, the eVC uses a simple segmented point-to-point scoreboard structure as shown in Figure 4. The transfer between a master and a slave is broken down into its individual segments, where each segment is a single bus or a bridge. The overall transfer is therefore treated as a series of smaller transfers, which keeps the code complexity down and makes it easy to pinpoint where a transfer is failing.

For each master on a bus, the eVC works out which slaves on the bus that the master can communicate with and creates a scoreboard for each path. For example, if a master can communicate with three slaves on its bus, the master will have three scoreboards - one going to each slave. The eVC also identifies bridges and adds a special scoreboard across the bridge that checks traffic across the bridge. Using this scheme, simple scoreboards can be used to check that any access from any master to any slave is correct.

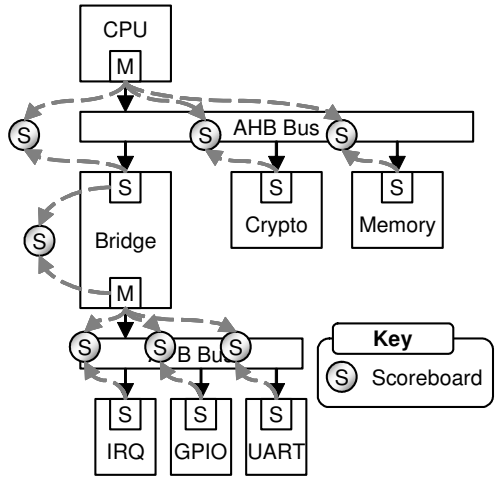

Figure 4: The scoreboard structure

Functional Coverage

In order to have coverage definitions that match the design, we do not use the coverage supplied with the bus eVCs. Instead, we define our own coverage groups which are similar to the original coverage but with modifications to make things easier for the rest of the eVC.

As discussed earlier, each master and slave bus interface can have protocol limitations which means that they will have coverage holes if the default coverage is used. There are also limitations on the slaves a master can reach. This may be because the slave is on a completely independent bus system, or that the routing table on a multi-layer specifies that the master can't see the slave.

Whatever the reason, we need to create individual coverage groups for each bus interface in the design. In the e language, coverage groups must be valid at compile time which is before the Topology Database has been initialised. We deal with this by having the eVC operate in a special mode where it uses the Topology Database to write out the customised coverage files. These can then be loaded by a later simulation.

Intra-bus Traffic Generation

It would normally be easy to generate bus traffic in an SoC. Black box the master or slave, attach an active interface agent (an AHB master, an APB slave, etc) and let it generate random transactions. Most agents do this by default, so what can we offer here?

Well, we can offer several things. The eVC supplies default master and slave sequences that take the topology of the design into account, which is essential on buses with a routing table. On many designs, just being a master on a bus doesn't let you see all of its slaves.

The eVC also takes the bus protocol limitations into account. If you told the eVC that your CPU instruction interface cannot performs writes, then you don't particularly want the default master sequence randomly generating writes. If your AHB bus doesn't support SPLIT responses, then you don't want to see these either. Due to the nature of the bus eVCs we have used, and

their desire to be really helpful[2], it can actually be quite difficult to stop them doing certain things.

One other feature provided by default is that the eVC will bias the default transaction generation towards hard to reach areas that we define in the eVC's functional coverage [3]. For example, many bus master sequences will generate random transfers that are bounded by the user-supplied minimum and maximum addresses. For functional coverage, we split each address segment into four sections - base address, lower range, upper range and top address. The lower and upper ranges are really easy to hit during simulations.

Without our biasing, hitting the lower address of a slave address range was a rare occurrence, and after running 1 million bus bursts (each burst accessed multiple addresses) we never once hit any of the upper addresses.

Bus Infrastructure and Interconnect Testing

Bus infrastructure and interconnect testing are fairly identical on every design. Or at least, they are once you know what the eVC knows. Tucked away in its Topology Database it knows the entire bus connectivity of the design and all protocol limitations. It knows the number of address ranges each slave has and the bounding addresses of each. It knows which of these address ranges can be accessed randomly without side effects, and which ones are unsafe to do so. It knows which bus interfaces have not been black boxed and can omit them from random testing.

[2] The helpfulness of some commercial eVC writers has in fact been the biggest problem we have had to overcome to create this eVC. In their desire to be super helpful, many have made it almost impossible to use their eVCs in a flexible manner. For instance, while defining slave memory maps through macros is nice, it also prevents you doing it programmatically, and causes people to create inflexible verification environments.

The default bus infrastructure sequence shipped with the eVC can be congigured as follows:

- **Number of paths:** A path is a master to slave transaction, which may be a burst
- **Number of threads:** This controls how many paths will be run in parallel. Each thread will run the full number of paths. So if you specify 100 paths and 10 threads that the eVC will do 1000 paths in total

And that's it. From here, the sequence will automatically deal with everything else. If you move it to a new working topology, it will work. If you move it to a new design, it will continue to work. If you change bus protocols, add new bus interfaces or bus limitations, it will continue to work without modification.

Code Sample 1 shows the bus infrastructure test required for any bus based SoC. This will generate random traffic between 20,000 master and slave combinations, running four masters in parallel. Of course, custom reset and initialisation sequences will also have to be added.

```
<'
extend MAIN vlb_soc_virtual_sequence {
 !bus_inf: PARALLEL_RANDOM_BUS_TRAFFIC
                vlb_soc_virtual_sequence;

 body() @driver.clock is {
  do bus_inf keeping{
    .number_of_paths   == 5000;
    .number_of_threads == 4;
   };
 };
};
'>
```

Code Sample 1: The bus infrastructure test for any bus based SoC

Example Topology

ALL A USER NEEDS TO DO TO USE THE EVC ON THEIR project us to define the topology of the design and testbench. These topologies are stored separately, but they are done identically, so we will not make any further distinction here. The

following code samples are taken from a real topology definition.

```
#ifdef USE_SUBSYSTEM_MAIN_BUS{                          1
  add bus {
    name         : AHBM;
    protocol     : AHB;                                 2
    multi-layer  : TRUE;

     <limitations>;
       add size : TWO_WORDS;
       add size : FOUR_WORDS;
       add size : EIGHT_WORDS;
       add size : SIXTEEN_WORDS;
       add size : K_BITS;
                                                        3
       // Do not allow splits or
       // retries on this bus
       add slave_responses : RETRY;
       add slave_responses : SPLIT;
     </limitations>;
   };
 // Instantiate the bus eVC
 ahb bus MAIN_BUS "rst_n" "clk" "";
};                                                      4
```

Code Sample 2: Adding a bus to the topology

Code Sample 2 shows how to add a bus to the topology. Points 2 and 3 add the bus entry to the Topology Database and point 4 instantiates the third party bus eVC. In this case, it is the AHB eVC from Cadence. This particular AHB bus doesn't support transfers wider that 32 bits, retry responses or split responses. This is specified at point 3. The code at point 1 is used to control the working topology. If the user specifies that the MAIN_BUS subsystem is not to be used, then this bus will not be added to the topology.

```
#ifdef USE_SUBSYSTEM_MAIN_BUS{
  add component_new{
    name : CPU;
    kind : PERIPHERAL;                                  1
  };
};
```

Code Sample 3: Adding a component to the topology

Code Sample 3 adds a component to the topology. It does this at point 1. There isn't much else to do with a component, although you may chose to load a third party eVC from here.

In Code Sample 4 below we add a master bus interface to the Topology Database. This is the master interface for the CPU component defined in Code Sample 3. The bus interface entry is added at 1, and the interface agent is added at 2. One task we leave for the user is to define the signal map for each agent. This, unfortunately, can't be automated.

In fact, automating this from the same description used to generate the design itself would be a bad idea. If a signal was missed from the common connectivity definition, the testbench would necessarily find the bug, because it would not expect the signal to exist.

```
#ifdef USE_SUBSYSTEM_MAIN_BUS{

  add master bus_interface_new{
    name      : CPU_IF;
    component: CPU;
    bus       : MAIN_BUS;
    protocol  : AHB;
    <limitations>;                                      1
      set no_lock       : TRUE;
      add transfer_kind : BUSY;
    </limitations>;
  };

 // Instantiate the Master agent
 ahb master vlb_soc_main_bus_u CPU_IF
                0 FALSE "rst_n" "clk";

 // Connect up the master
 extend CPU_IF VLB_SOC                                  2
             vr_ahb_master_signal_map {
  // Signal connects removed
  // for clarity
 };
};
```

Code Sample 4: Adding a bus interface to the topology

The last code we need to show is the code required to define the working topology for a testbench. This is specified in a text file and processed by a Perl script. The output of this script is an e file with #defines and an e file to configure the active flags in the Topology Database.

```
<use>
 AHB    t // Use the AHB eVC
 CIPHER f // Don't use the GPRS eVC
</use>
```

```
<subsystem>
 MAIN_BUS t // Include the MAIN_BUS
            // subsystem
 SECURITY f // Don't include the
            // SECURITY subsystem
</subsystem>

<black_box>
 bus f MAIN_BUS; // Don't black box the
                 // MAIN_BUS
 bus t SEC_BUS;  // Black box the
                 // Security bus
 com t CPU;      // Black box the CPU
                 // component
 bif t CPU_IF;   // Black box the CPU's
                 // master interface
                 // Other entries removed
</black_box>
```

Code Sample 5: Specifying a working topology

Future Work

THE EVC AS IT EXISTS TODAY SATISFIES ALL THE uses our customer currently has for it. It is being successfully used on several projects, and we are waiting until these are complete before planning any new features. Three candidate possibilities have emerged, but we are still looking for justification to make the changes.

A mapping layer: The eVC can encapsulate the architecture of the design and some of its capabilities, but what it's can't do, unless you define multiple topologies, is encapsulate the technology mapping of the design. By default, the eVC assumes that buses and bus interfaces are mapped to VHDL. If one is mapped to Verilog or SystemC, then this has to be specified somewhere, and the topology definition is the most obvious place. In some larger projects there may be the need to change this mapping for certain testbenches. For some, a bus interface may be implemented in Verilog, but for others it might be in SystemC for performance reasons. In other testbenches it may be on a hardware accelerator.

Port to other languages: The eVC is in e, but there are no reasons why it couldn't be written in Vera or SystemVerilog. However, there are currently no requirements for this.

Alignment with SPIRIT: The SPIRIT consortium [5] are attempting to ease the use of IP by developing an XML standard for describing the physical attributes of the IP. These descriptions can be brought together to form a description of the overall design, which can then be processed by a set of custom tools. The current release (1.0) does not contain enough information to generate a testbench as we have done, but there is a verification working group looking into this. As our eVC is already synergistic with the existing SPIRIT approach, we would consider full alignment an obvious approach. This of course will depend on SPIRIT supporting all of the features that we need, and our customer supporting SPIRIT.

Conclusions

OUR GENERIC SOC EVC BRINGS REAL COST AND time savings to verification projects. Even on large designs, we can run the first tests within days of receiving the RTL for the bus infrastructure. On the latest design, we went from receiving the specification to 100% functional coverage for the bus infrastructure tests in just 4 days, which included waiting for critical RTL fixes. Because the eVC is designed to deal with change, the design can subsequently arrive in stages without having any significant impact on the verification.

No real knowledge of e is required to use the eVC, because the topology definition and control is mainly macro based. Users don't need to know how to instantiate the eVCs required for the busses, how to deal with their strange and unexpected behaviours, or how to set up the design's address maps. No complex programming is required [1], team members don't need to know the eRM or other coding techniques, and good encapsulation is provided by default.

By severely reducing the time it takes to set up a project's verification environment, the number of verification engineers it needs, and the skill level of those engineers, projects using this eVC

can reduce their costs and increase their chances of taping out their designs on schedule.

Related Work

WE ARE NOT THE FIRST PEOPLE TO TRY AND EASE verification by letting users specify the design at a high level and then automatically generating a testbench.

The X-Gen tool [4] is intended to replace the e, Vera and SystemVerilog languages. Users describe the components, the interconnectivity and the transactions between the components, and the tool will generate random stimuli for the design. Self checking and functional coverage have to be provided by other means. As it is not a standard language, it doesn't appear to be possible to make use of existing verification IP.

Beach Solutions sell two commercial tools that take a central description of the design and automatically generate a verification environment. These both appear to be aimed at generating register tests, but will also instantiate the AHB eVC from Cadence. It is unclear if the resulting environment can be manually extended while maintaining the ability to be automatically regenerated.

The SPIRIT consortium, of which Beach Solutions is a founding member, is working on extending their XML schema to include information needed to generate testbenches for a design. As discussed in "Future Work", we will look at aligning with SPIRIT when it becomes publicly available.

Terminology

Bus: a communication channel that connects masters to slaves. This would normally consist of multiple signals, but a point-to-point wire connection also qualifies. A bus would typically have a non-trivial protocol.

Component: an object in the design, components are also known as blocks or modules.

Bus Interface: the interface between a component and a bus. A component can have one or more bus interfaces. For instance, a DMA component may have a slave bus interface for configuration, and two master interfaces per DMA channel.

Bus eVC: An eVC that can monitor and model a bus. An eVC that provides agents for AHB masters, slaves, decoders and arbiters is an example of a bus eVC.

References

[1] G. Mosensoson, "Practical Approaches to SoC Verification", Verisity Design Whitepaper

[2] Dr. M. Ruhwandl, "Functional System Verification Planning and Execution Using Skeleton Approach", ClubV, March 2005

[3] R. Emek, et al, "Quality Improvement Methods for System-Level Stimuli Generation", ICCD 2004

[4] A. Golin et al, "X-Gen: A Random Test-Case Generator For Systems And Socs", HLDVT '02 , Nov 2002

[5] "Structure for Packaging, Integrating and Re-using IP within Tool-flows", SPIRIT, http://www.spiritconsortium.org/index.html