Using Formal Applications to Create Pristine IPs

Lee Burns
Elec Engr Principal
Cypress Semiconductor
425-787-4461
Lynnwood, WA 98087
lbrn@cypress.com

David Crutchfield
CAD Engr Sr Principal
Cypress Semiconductor
859 977 7555
Lexington, KY 40507
daac@cypress.com

Bob Metzler
Elec Engr Principal
425-985-6221
Lynnwood, WA 98087
technobob@foxinternet.net

Hithesh Velkooru
CAD Engr Sr
Cypress Semiconductor
859 977 7581
Lexington, KY 40507
hitr@cypress.com

***Abstract*- The search for a technology that can have the benefits of rapid and exhaustive verification led us to Formal verification methodologies. This paper will focus on the use of such methodologies within Cypress to improve IP quality throughout the design and verification cycle. At that point of time we did not have Formal expertise in our team. Fortunately, a shift in the industry to automate Formal verification techniques into applications was occurring at the same time. These applications give the benefits of Formal verification without forcing a steep learning curve, enabling non-experts to take advantage of exhaustive Formal analysis. In this paper we will describe how we leveraged Formal-based applications to expedite some high-value verification tasks and how we brought down the verification cycle time to hours from weeks. Additionally, we will present how we integrated Formal applications within our existing Verification Management System to seamlessly enable these tools for our engineers saving countless hours of training.**

I. Introduction

With any library of reusable IP, it is imperative to fully verify each configuration of an IP without sacrificing time to market or quality. Traditional functional simulation methods are unable to satisfy quality requirements within all configurations along with delivery requirements. At Cypress the desire was to leverage Formal applications to achieve quality in different areas of the design cycle and reduce development effort. The targeted areas for Formal applications were register behavioral verification, intra-IP and inter-IP connectivity verification, mining unreachable code, and coverage closure. We evaluated Formal verification tools in the market, chose the tool that would fit efficiently into our existing digital verification infrastructure, and provided an immediate impact for our verification teams.

Before integrating Formal applications, simulation based verification was required for verifying pin-constrained IO pad multiplexing and on-chip bus connectivity. This typically took 7-8 weeks to get reliable simulation results on

a design consisting of approximately 50K logic gates. The Formal based application we leveraged requires as input a CSV formatted file containing connectivity information. This CSV specification typically takes 1-2 days to create. However, through automation, this specification can be generated from our existing design database that contains all sorts of information including connectivity. Overall, with automation, and by leveraging Formal applications, verification run times can be reduced to a few minutes.

Coverage closure is an important indicator in our IP verification process and is required. Mining out unreachable RTL code can be very time consuming and involves a substantial amount of manual work. Usually, for a small to medium IP, one week is required for identifying dead code. Through a Formal application we were able to generate exclusions within few minutes to hours of time. This application also generates information that could help in writing tests for uncovered RTL. By integrating this application into our Verification Management System we are able to automate generation of exclusions for unreachable items from coverage criteria in conjunction with functional simulation regressions in a single run.

For register verification we previously relied exclusively on UVM test benches using UVM_REG. Typically this would take four to six weeks for writing tests and debugging the test bench for an IP with approximately 50K logic gates. The Formal application that aids in this verification task takes only a few days for setup including creation of a CSV formatted file describing the registers and their access policies and configuration variables. This application tests to make sure that there are no corner-case bugs hiding between configuration schemas and access policies. The results are impressive considering run time is less than one day. We are able to find issues that would take a very long time for a constrained random simulation to find.

Along with these applications we found a Formal application that could point out critical problems in the RTL early in the design process and also point out to the verification team where potential problems might exist. This application complements the functions of a Linting tool with checks for coverage closure issues, functional impact of 'X's and common design errors.

We experienced significant benefits once these Formal applications were integrated into our existing Verification Management System making usage seamless to our verification engineers. We are able to leverage existing design database and automation software to create inputs for Formal applications and integrate outputs of Formal verification and simulation based verification. Implementation of each Formal application discussed will be detailed below along with results and issues encountered to this point.

II. Definition of Terms

CPU	Central Processing Unit
CSV	Comma Separated Value. A text file format wherein fields are delineated by commas.
FPGA	Field Programmable Gate Array
IP	Intellectual Property, refers to reusable design
IP-XACT	XML format that defines and describes designs in a standard way
LSF	Platform Load Sharing Facility – Workload management platform, job scheduler, for distributed HPC environments
PLD	Programmable Logic Device
Questa VM	Questa Verification Management – Functionality within Questa SIM that offers a wide variety of features for managing the regression. These features are built upon the UCDB
Questa VRM	Questa Verification Run Management – Mentor's tool for launching verification tasks within a regression
RTL	Register Transfer Level. Design abstraction which models a synchronous digital circuit with regards to digital signal flow between hardware registers
SOC	System-on-a-chip
UVM	Universal Verification Methodology
VMS	Verification Management System – Internally developed tool for gathering design and test bench file information, compilation arguments, simulation arguments and test information, launching each task, and collecting regression information and status
XML	Extensible Markup Language

III. Verification Management System

Years ago Cypress developed VMS (Verification Management System) to manage logic verification regressions across the entire company. Along with the VMS tool, the project created a standard for design and test bench organization, specification of tool arguments, test list creation, regression status reports, and coverage information. VMS leverages Mentor Graphics' Questa VRM (Verification Run Management) tool to launch compilation and

simulation jobs through a load sharing facility (LSF), and to generate and merge functional coverage information. Having a standardized verification environment is important in cases where new techniques are desired. These techniques or methodologies can be developed and propagated easily throughout the company without burdening verification engineers who are focused on design and product delivery. In the context of this paper the intent was to introduce Formal verification applications within an existing infrastructure to improve IP quality. Each section will highlight the steps through VMS used to take advantage of Formal applications.

IV. Connectivity Checks

The Questa Formal connectivity checker is an application that converts a CSV or XML connection specification into System Verilog assertions. These assertions are fed into Questa Formal Property checking along with the design for validate the assertions are true. The assertion generator supports simple wired connections, conditional connections, and connections with clock-cycle delays. So it is really more than just a connectivity checker, as it can be used to:

- Check signal connections at any level of implementation intra-module, inter-module, or inter-IP.
- Check the function of any combinatorial logic with a truth table.
- Check the function of a state machine, pipeline logic, etc.

Of course we can check all of these things with simulation, but Formal may be a better choice if:

- Your testbench is not ready for use.
- You don't have time to develop a reference model/scoreboard.
- Your specification is written to emphasize how functional blocks are connected; this is likely true at the chip level, and may apply at the IP level. It really depends on what the IP does and how the RTL code is implemented.

A. Implementation

The Questa Formal connectivity flow consists of generating property assertions from a user provided input specification, in the form of a CSV file, followed by execution of Questa Formal property verification on the generated assertions. If all assertions are properly verified to pass then all connections are considered valid. As VMS is responsible for design management and compilation for all verification tasks, the system was modified to include Questa Formal tool execution. Fig. 1 provides the implemented flow.

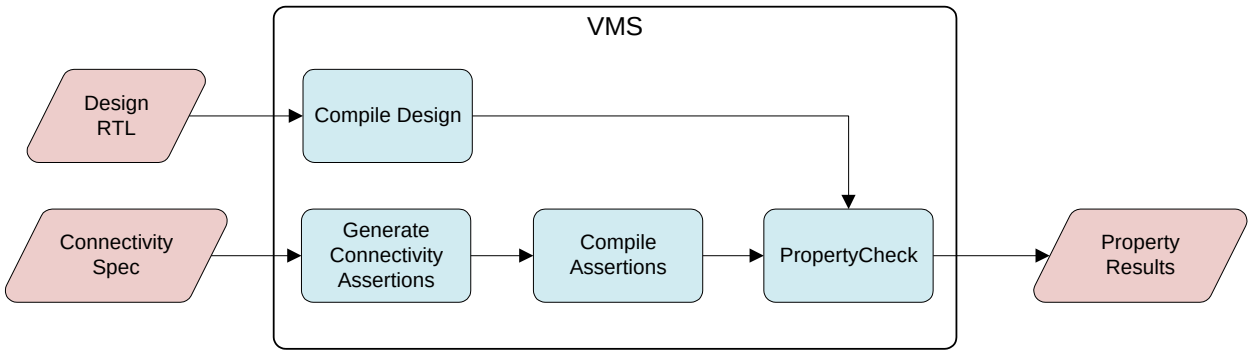

Figure 1: VMS Connectivity Check Flow

For connectivity verification, VMS provides an input mechanism for the connectivity specification and uses a Questa Formal application, qconnect, to generate connectivity assertions. Once created, the assertions are compiled for usage in Questa Formal as properties to be verified. The process of compiling design RTL is the same as that of functional verification and can simply feed the Questa Formal step. Note that a test bench is not required in this flow.

B. Results

VMS, using the Questa Formal connectivity checking application, was targeted for connectivity checks at the chip level, as well as, verifying portions of a PLD-like IP. Only results from IP connectivity verification are covered

here. The IP in this case contains islands of programmable logic, joined by programmable interconnect. This type of IP is difficult to verify with the traditional UVM approach, as it would require a complicated reference model and loads of time to write tests. Writing UVM tests to exhaustively verify an FPGA would be a monumental task. Given the nature of the IP, the block functional specification is largely written in structural terms. Thus it was straightforward to get from functional specification to connectivity specification. This connectivity specification was used to verify:

- the user-programmable routing blocks
- the connections between the routing block instances and the programmable logic blocks.

A typical CSV routing specification looks like this (#s define comments):

```
# Format specifier: type,src,dest,cond,delay are MG keywords
type,src,dest,cond,delay

# top_block.route_in[0] --> top_block.route_out[0] (when route is enabled, async mode)
cond,top_block.route_in[0],top_block.route_out[0],(top_block.route_cfg && !top_block.sync)

# top_block.route_in[0] --> top_block.route_out[0] (when route is enabled,1 clock cycle delay)
cond_dly,top_block.route_in[0],top_block.route_out[0],(top_block.route_cfg && top_block.sync),1

# 'b0 --> top_block.route_out[0] (tied low when routing is disabled)
cond_tied_low,top_block.route_out[10],top_block.disable[10]
```

Above, 'route_cfg' enables the routing path and 'sync' determines if pipelining is enabled for the path. Other connection types are supported but were not used for this IP. The complete list is: connect, connect_dly, connect_inv, connect_allsame, cond, cond_inv, cond_allsame, cond_dly, mcond_dly, mutex, cond_mutex, tied_high, cond_tied_high, tied_low, and cond_tied_low. Once the CSV specification is written, it is simply provided to VMS as input for the connectivity checking flow.

Below is one of the connectivity checks from a routing CSV file:

```
type,src,dest,cond,delay
cond,s40udbtk_route.ext_route_left_horz_in[0],
     s40udbtk_route.udbpair_route_left_horz_out[0],
     (s40udbtk_route.csr_route_lho_cfg0_lho0sel==0)
      && !s40udbtk_route.csr_private_pipeline_md
```

The generated SV code is:

```
// ------------------------------------------------
// Row Number: 15
// type: cond
// src: s40udbtk_route.ext_route_left_horz_in[ 0]
// dest: s40udbtk_route.udbpair_route_left_horz_out[ 0]
// cond: (s40udbtk_route.csr_route_lho_cfg0_lho0sel==0) &&
//        !s40udbtk_route.csr_private_pipeline_md
// ------------------------------------------------
check_cond #( .width(`WidthParam_0) )
CHECK_COND_FROM_s40udbtk_route_ext_route_left_horz_in_0_TO_s40udbtk_route_udbpair_route_left_ho
rz_out_0 (
      .cond( (s40udbtk_route.csr_route_lho_cfg0_lho0sel == 0) &&
             !s40udbtk_route.csr_private_pipeline_md),
      .src( s40udbtk_route.ext_route_left_horz_in[0] ),
      .dest( s40udbtk_route.udbpair_route_left_horz_out[0])
);

// Signal Name : s40udbtk_route.ext_route_left_horz_in[ 0]
`define WidthParam_0 1

module check_cond #(parameter width = 1) (
    input logic [width-1:0] src , dest ,
    input logic cond
);
```

```
    connectivity_assert : assert property ( @($global_clock)
        cond |-> src == dest
      );
endmodule : check_cond
```

Using the Formal connectivity application we completed the routing verification in one week. This was a savings of four person-weeks compared to our estimate of the simulation approach, which likely would not have yielded full coverage in the allotted time. Using the Formal connectivity application we discovered a major bug that would have been difficult to detect using IP-level simulation. Further, given the projects schedule constraints it is unlikely that we would have identified the bug before tapeout had we only used simulations.

TABLE I
CONNECTIVITY CHECK SUMMARY FOR PLD

Block	Route Inputs	Route Outputs	Control Register Bits	# Lines in CSV File	Formal Runtime (minutes)
1	408	308	962	5266	2
2	408	308	962	5266	2
3	312	256	786	5127	2
4	312	256	786	5127	2

The strengths of using the Formal connectivity check are:

- Short runtimes
- Short learning curve: the user does not need to know anything about Formal assertions to get started.
- No testbench required!
- Through using Formal methods, verification starts sooner, and shortens the overall schedule. Also, bugs are found sooner, which may prevent hours of debug through functional simulations.
- Filling in simulation coverage gaps by targeting regions of code with Formal techniques.

The main areas of concern for Formal connectivity are:

1. Knowing when to use it: Blocks that are defined structurally (like PLD IPs or SOCs) and complex combinatorial logic blocks are good candidates.
2. Understanding how you will combine your Formal results and your simulation results to decide when your coverage goals are achieved: There is currently no standard way of combining Formal and simulation coverage. In simulation we chose to exclude module code coverage for each routing block instance because we had all paths tested by Formal. This may not be the right choice for every project.
3. Creating the connectivity specification without reading the RTL code: We always test to the Cypress specification. As we increase our use of Formal we are finding that we need to write better IP specifications.
4. Maintaining strict code modularity: We also need to write better RTL code. Modules should only contain the code necessary to implement a specific behavior. This allows us to use Formal techniques to divide and conquer.
5. Automating the generation of the connectivity specification:
    a. For IPs, design churn leads to verification churn. Automating verification tasks will minimize the schedule impact of design changes. We chose not to automate the CSV file generation for the PLD IP because we expect that the routing blocks would be the same in future revisions.
    b. For SOCs, it is possible to achieve a push-button flow from chip specification to proven IP integration, which eliminates the need for many chip-level simulations.

V. COVERAGE EXCLUSIONS

Coverage closure is a critical component in IP verification. With any IP, it is not unusual to exclude coverage for lines or portions of code that cannot be reached due to configuration constraints placed on the design. In such cases unreachable RTL code must be indentified and coverage exclusions created and passed to the simulator for calculating accurate coverage. Mining out unreachable code is a significant pain point if done manually. It can take

a week or more initially, with additional time to vet and maintain manual exclusions every time the code changes to ensure that all exclusions are still valid. An automated process is required.

A. Implementation

Mentor's Questa Formal tool provides the CoverCheck application, which takes in design files and automatically generates the applicable exclusions for unreachable code within a relatively short amount of time. While this is a very important step forward, it is necessary to apply exclusions on final coverage in a second step. As the VMS tool is responsible for managing design compilation, launching all functional tests, and merging coverage, it follows that it should seamlessly integrate coverage exclusions to achieve the highest end coverage possible, without user interaction. Through integration of this application into VMS, several flows have been identified that users within Cypress can use for generation and application of coverage exclusions.

The first method, as shown in Fig. 2, is to run CoverCheck on the compiled design, generate exclusions and then apply them to coverage results from the regression executed. This can be considered the general purpose flow as user interaction is not required.

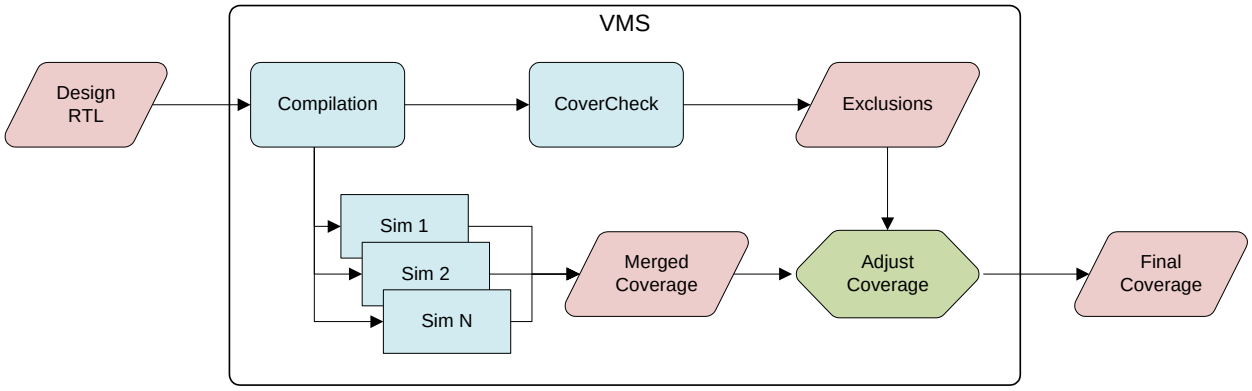

Figure 2: General Purpose Coverage Exclusion Flow

In this flow CoverCheck is being executed without any prior coverage information being provided. Without coverage input CoverCheck must exhaustively search all coverage space mining for unreachable code, which can be time consuming. However, this step may only need to be executed once and exclusions applied on subsequent runs to close coverage. Fig. 3 highlights the slightly modified flow of providing exclusions for subsequent runs.

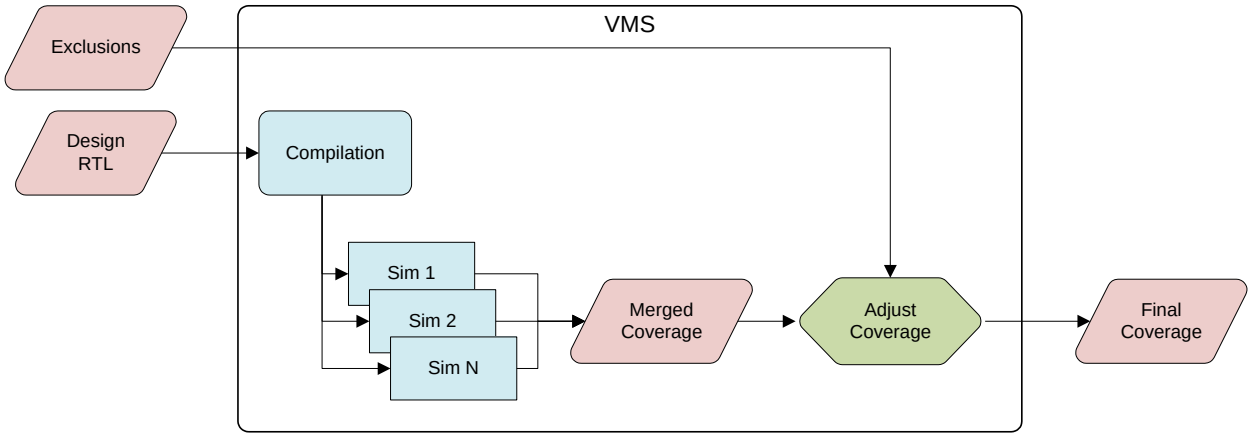

Figure 3: Coverage Exclusion Without CoverCheck

Here, exclusions are generated in a previous run and applied in subsequent runs eliminating the need for additional executions of CoverCheck.

The last flow proposed switches the order slightly by seeding CoverCheck with known coverage. Fig. 4 presents this flow.

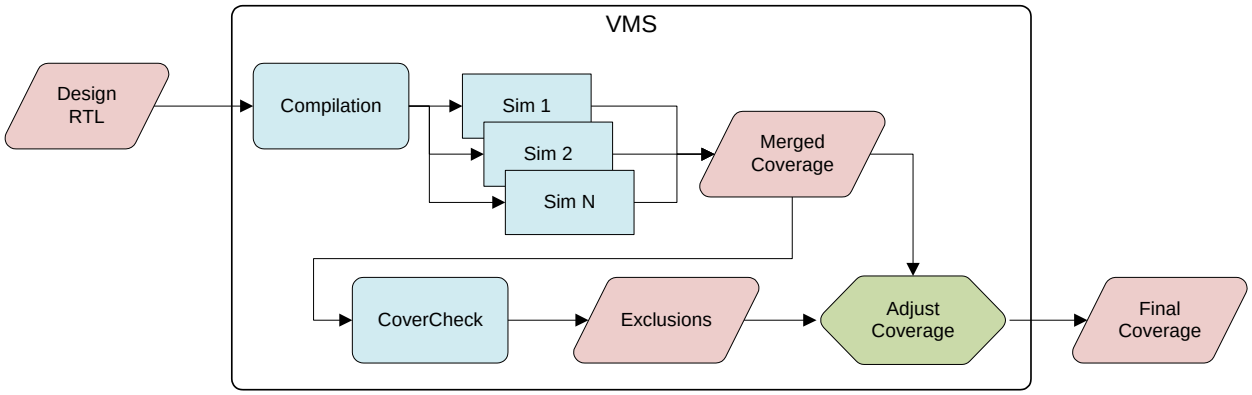

Figure 4: Post Simulation Exclusion Generation

By executing CoverCheck after an initial regression run, unreachable code space can be minimized by code found to be covered through previously written tests. This can greatly reduce run times for CoverCheck as mining is not as exhaustive. As with Figs. 2 and 3, once exclusions have been identified they can be applied directly through VMS to eliminate the need for executing CoverCheck in subsequent runs. Fig. 4 presents the preferred flow when tests are available. In cases where coverage investigation is just beginning and tests are few then the flow of Fig. 2 is useful for eliminating unreachable code before developing a regression suite.

B. Results

Below are results from executing CoverCheck on one IP. There were a total of 17783 unreachable items with regards to code coverage. Manually detecting each of these would take several weeks of functional simulation iterations. In this case the total elapsed time was approximately 1 hour.

```
# CoverCheck Summary
# ------------------------------------------------------------------------------
# Coverage Type          Active          Unreachable     Witness         Inconclusives
# ------------------------------------------------------------------------------
# Branch                 96525           8489            82998           5038
# Condition               1082           22              400             660
# Expression             31714           1517            14991           15206
# FSM                    1114            0               10              1104
#   States               388             0               4               384
#   Transitions          726             0               6               720
# Statement              91522           7577            80343           3602
# Toggle                 72074           178             58715           13181
# Coverbin                   0           0               0               0
# ------------------------------------------------------------------------------
# Total                  294031          17783           237457          38791
# ------------------------------------------------------------------------------
#
# --------- Process Statistics ----------
# Elapsed Time (s):              3619
# -------------- kblue01:0 --------------
# Total CPU Time (s):            3011
# Memory Used (MB):              12372
# ---------------------------------------
```

While the tool is useful for quickly identifying unreachable code, we did discover a fairly significant limitation. As mentioned previously, the desire is to create quality reusable IP. Typically, for an IP to be reusable, it must also be highly parameterized, enabling multiple configurations, depending on the application. Therefore, exclusions generated for a highly configurable IP, must be valid for all legal combinations of parameters. For functional simulations, parameters are "floated" during compilation and set at simulation time through parameter options.

CoverCheck currently only uses default parameter values during compilation unless overridden with the –G switch. The parameters are fixed for a given execution of CoverCheck. It is unable to accept valid ranges for parameters and generate exclusions for unreachable RTL that is common for all settings. Without automation, a user is required to manually launch CoverCheck with each parameter option, merge results once all combinations have been Formally verified or manually categorize exclusions based on configurations. This is too cumbersome for a user to manage and reduces the benefit of Formal coverage exclusion generation significantly. After understanding the need, Mentor Graphics is investigating an enhancement to CoverCheck for allowing parameter ranges as an input to the tool. This is not a trivial issue to overcome and Mentor has not promised to resolve it. The need exists for Questa Formal to automatically iterate runs across all parameter ranges and combinations. Fig. 5 attempts to capture the intent.

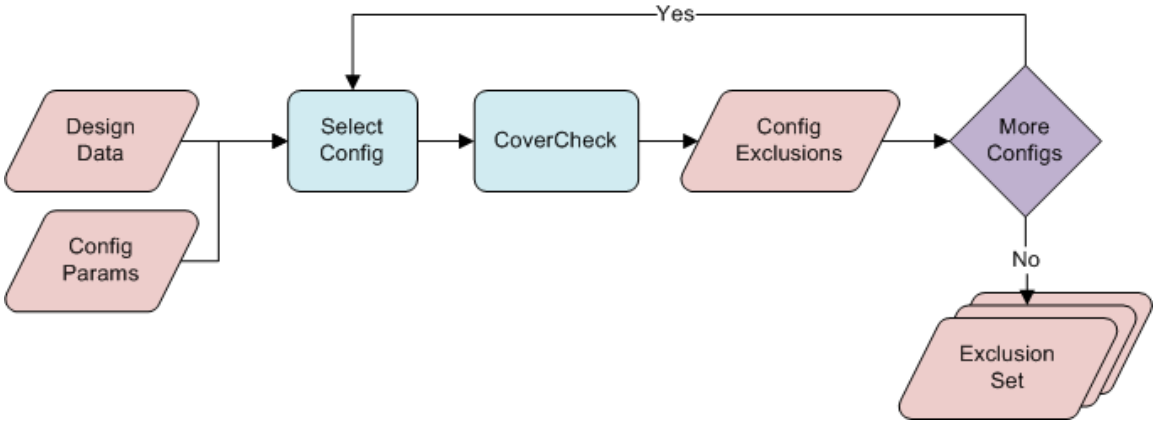

Figure 5: Iterative Exclusion Generation Through Configuration Parameters

Shown here is an enhancement to Questa Formal that would automate the selection of configurations for mining unreachable coverage for each. After every configuration is checked for unreachable code, the exclusions generated from each run would need to be categorized into configuration groups or into a set of shared exclusions. These exclusions would then be applied as shown in Figs 2-4 for each individual configuration run before merging results.

Despite the current limitation, we have found the tool useful for implementation specific exclusions by specifying the parameter values that will be assigned at a chip level. Once we can generate the superset of exclusions, we will be able to take this a step further for vaulted IP and be better able to ensure full code coverage can be achieved.

VI. Register Checks

At Cypress, design registers are verified using UVM test benches with UVM_REG. Typically this takes four to six weeks for writing tests and debugging the test bench for an IP with approximately 50K ASIC gates. RegisterCheck, the Formal application in Questa Formal, takes a few days for setup and a couple of hours to generate properties and verify them. The setup includes creating a CSV formatted file that describes the registers and their access policies, as well as, configuring design variables. Questa Formal RegisterCheck uses Formal methods to verify memory mapped registers in the digital design. From a top level, RegisterCheck takes in a register specification file and a configuration file, generates assertions accordingly and formally verifies them. The register specification contains register information, such as, register name, register address, register width, register access, reset value, various fields, and access policy. The config file contains information to configure a particular RegisterCheck run. It may contain details related to interface ports, register specification format, interface type, and base addresses.

A. Implementation

IP-XACT is not used for requirements specifications within Cypress. Instead CSV register specifications are created from an internal specification database. Perl scripts were developed to automate conversion from our internal specification database to the required RegisterCheck UVM format or CSV. An example section from a CSV file containing one register with four fields is shown below.

```
Register Name,Register Description,Register Address,Register Width,Register Access,Register
Reset Value,Register Reset Mask,Field Name,Field Description,Field Offset,Field Width,Field
```

```
Access,Field Reset Value,Field Reset Mask,Field Is Covered,Field Is
Reserved,.memmap_write_internal
botsel_1,"comment",0x00007808,32,RW,0x0,0xffffffff,clk_sel0,"comment",0,2,RW,0x0,0xffffffff,,,
botsel_1,"comment",0x00007808,32,RW,0x0,0xffffffff,clk_sel1,"comment",2,2,RW,0x0,0xffffffff,,,
botsel_1,"comment",0x00007808,32,RW,0x0,0xffffffff,clk_sel2,"comment",4,2,RW,0x0,0xffffffff,,,
botsel_1,"comment",0x00007808,32,RW,0x0,0xffffffff,clk_sel3,"comment",6,2,RW,0x0,0xffffffff,,,
```

The first line of the register specification sets formatting for all register properties provided.

The user must also create a control file that identifies AHB signals, hierarchical location of register variables, and the naming scheme for register variables:

```
-ra
-register        u_csr_bctl.$register_$field
-interface       amba_ahb
-base_addr       0x00000000
-spec_type       uvm
-signal_match    nocase,prefix,postfix

-interface_port  hready_in  = mmio_hready
-interface_port  hselx      = mmio_hsel
-interface_port  hwrite     = mmio_hwrite
-interface_port  haddr      = mmio_haddr
-interface_port  hwdata     = mmio_hwdata
-interface_port  htrans     = mmio_htrans
-interface_port  hsize      = mmio_hsize
-interface_port  hmaster    = mmio_hmaster
-interface_port  hprot      = mmio_hprot
-interface_port  hburst     = mmio_hburst
-interface_port  hmastlock  = mmio_hmastlock
-interface_port  prot_mode  = protection_mode
-interface_port  hrdata     = mmio_hrdata
-interface_port  hready_out = mmio_hready_out
-interface_port  hresp      = mmio_hresp
-interface_port  hresetn    = rst_hf_act_n
-interface_port  hclk       = clk_sys
```

In this example, 'u_csr_bctl' is the instance name of the register block.

Fig. 6 shows the RegisterCheck flow within VMS. An input mechanism exists for providing the register specification (CSV). The typical design compilation flow within VMS is utilized in combination with a user provided register specification to enable Formal register checking.

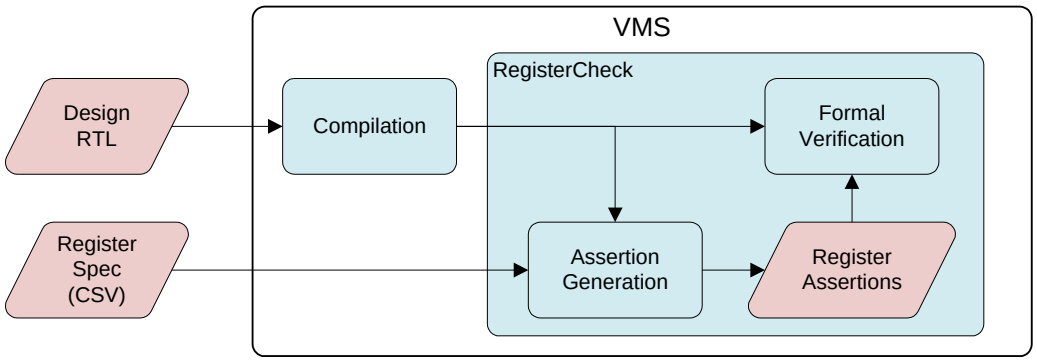

Figure 6: RegisterCheck Flow in VMS

As in the case of connectivity checks, a test bench is not needed to accomplish register verification, providing a much faster quality check on a portion of the design.

B. Results

Traditionally, functional verification through UVM tests would be used on the PLD IP for register verification. This IP has thousands of registers. With the maximum configuration selected through SV parameters, the PLD

would have 2,781 registers, for a total of 27,105 register fields. In total, around 55,000 AHB accesses would be required to fully verify all register properties, assuming that the average register used 16 of the available 32 bits. Simulation runtime would likely be more than 50 CPU hours.

RegisterCheck was used to verify a small subset of registers in a PLD-like IP. This work was a proof-of-concept exercise to drive development of a fully automated flow. In this case the register CSV specification contained ten register fields. The Questa Formal runtime on a single CPU was 108 minutes, where 32 minutes was consumed for compilation and 76 minutes for proving the properties.

RegisterCheck has the potential to eliminate simulation-based verification of register properties. In order to achieve the full benefit, design and verification teams should consider the following steps before developing IPs:

- The design team must standardize on register variable naming rules. One way to promote standardized naming is through automated register generation from the same specification provided to RegisterCheck.
- Develop a fully automated flow to get from a register specification to a format supported by RegisterCheck. Companies using IP-XACT to define IPs will have no trouble with this step. Others will need to spend a few weeks of coding to work out the kinks.

In the case of the PLD IP several issues were encountered while using RegisterCheck:

- Runtimes were too long. During evaluation 7.6 minutes per register field was seen. Assuming linear scaling it would take 205998 CPU minutes, or 143 CPU days to verify of 27,105 register fields! Likely this could be reduced significantly through performance profiling to understand bottlenecks, but not enough time was allocated for this during the evaluation. Based on other Formal work for this IP, it is likely that clever black boxing could reduce the runtime by approximately 20x. Through efficiency gains it may be possible to verify the IP with the maximum design parameters.
- The PLD block has some ‘unique’ register access modes that involved address aliasing, address ganging, and other access modes intended to reduce the programming time. RegisterCheck was unable to understand these access modes. Further investigation with Mentor on these access modes would be needed.
- Each register RTL module used a different naming convention for the register variables.
- The AHB slave for this IP did not support wait state insertion for write operations. A netlist constraint telling Questa Formal to assume that HREADY was always active was used to work around this issue.
- Constraints were added to disable scan mode, and to prevent certain resets after initialization.
- The AHB implementation in this case had a minor ‘improvement’ over the standard that implemented some enhanced security, causing there to be an extra AHB signal from what RegisterCheck assumed. To work around this issue we edited the generated checker module so that the extra AHB signal was held low.

VII. QUALITY CHECKS

A requirement for developing quality IP is to perform Lint checks to catch minor coding issues and check for coding style directives. Questa Formal AutoCheck certifies quality of a design using Formal methods and is a complementary tool to Linting. While this application cannot replace Linting completely, it can mine out bugs that can go unnoticed through functional verification at a very early stage in the verification flow. This tool can be used by the designer immediately after the design is ready. The setup is very simple and performance can be improved by configuring various options. The application generates assertions automatically and verifies those using Formal methods. A debug data base is generated by default, which can help pinpoint bugs in the source code. It creates a synthesized netlist and does sequential analysis. When used, AutoCheck, performs initialization checks like uninitialized registers and X propagation, functional issue checks like combinational loops, case statement checks, arithmetic checks, bus checks, FSM checks, and coverage reachability checks like unreachable logic, unreachable FSM states and transitions, and register stuck at constant values. These checks can greatly improve IP quality at an earlier stage of design.

C. Implementation

Like the other Formal apps, AutoCheck was integrated into VMS making it easy to launch within an understood environment. Once the design code is ready, the user will simply launch VMS with an additional option to enable AutoCheck. Fig. 7 shows the integration of AutoCheck into VMS.

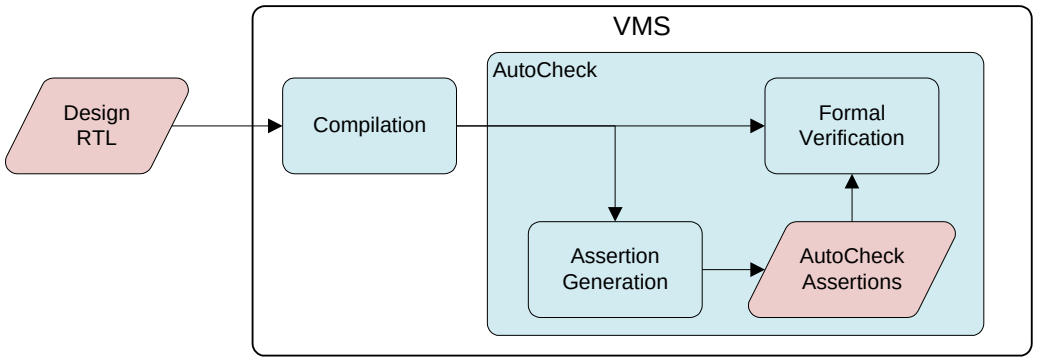

Figure 7: AutoCheck Flow in VMS

VMS utilizes the typical design compilation step followed by AutoCheck. The user can configure the run by enabling and disabling various checks, and improve performance through black boxing, simplifying the clock, and varying hierarchy levels. The tool generates assertions and Formally verifies them. Generated logs files will contain information about all the checks performed.

D. Results

Executing AutoCheck on one particular IP tool took approximately 72 minutes and consumed 8GB of memory. Below are results from this execution.

```
# AutoCheck Summary
# ------------------------------------------------------------------------
# Check                          Evaluations          Found          Waived
# ------------------------------------------------------------------------
# ARITH_OVERFLOW_SUB             OFF                  OFF            OFF
# ARITH_OVERFLOW_VAL             OFF                  OFF            OFF
# ARITH_ZERO_DIV                 0                    0              0
# ARITH_ZERO_MOD                 0                    0              0
# ASSIGN_IMPLICIT_CONSTANT       73446                0              0
# BLOCK_UNREACHABLE              8680                 1690           0
# BUS_MULTIPLY_DRIVEN            2                    2              0
# BUS_UNDRIVEN                   2                    2              0
# BUS_VALUE_CONFLICT             2                    2              0
# CASE_DEFAULT                   OFF                  OFF            OFF
# CASE_DUPLICATE                 7023                 0              0
# CASE_FULL                      0                    0              0
# CASE_PARALLEL                  0                    0              0
# CLK_DELAY                      9069                 0              0
# CLK_IN_DATA                    96074                0              0
# COMBO_LOOP                     32806                114            0
# DECLARATION_UNDRIVEN           17495                0              0
# DECLARATION_UNUSED             17495                36             0
# FSM_DEADLOCK                   388                  0              0
# FSM_LIVELOCK                   388                  0              0
# FSM_STUCK_BIT                  146                  0              0
# FSM_UNREACHABLE_STATE          388                  0              0
# FSM_UNREACHABLE_TRANS          388                  0              0
# FUNCTION_INCOMPLETE_ASSIGN     OFF                  OFF            OFF
# INDEX_ILLEGAL                  489                  0              0
# INDEX_UNREACHABLE              0                    0              0
# INIT_X_UNRESOLVED              9067                 9067           0
# LATCH_INFERRED                 OFF                  OFF            OFF
```

# LOGIC_UNDRIVEN	17495	0	0
# LOGIC_UNUSED	786	71	0
# ONE_COLD	0	0	0
# ONE_HOT	0	0	0
# PORT_UNDRIVEN	32547	0	0
# PORT_UNUSED	32547	31	0
# REG_MIXED_ASSIGNS	18495	0	0
# REG_MULTIPLY_DRIVEN	9069	0	0
# REG_NO_RESET	OFF	OFF	OFF
# REG_RACE	OFF	OFF	OFF
# REG_STUCK_AT	14595	0	0
# REG_TOGGLE_VIOLATION	3248	0	0
# REG_VARIABLE_ARESET	18649	0	0
# RESET_HIGH_LOW	0	0	0
# RESET_SYNC_ASYNC	0	0	0
# SLIST_INCOMPLETE	4076	0	0
# X_ASSIGN_REACHABLE	11	2	0
# ---			
# AC Total	424866	11017	0
# ---			

Given that this IP was a proven IP there were no major issues highlighted that were not already understood. However, the RTL quality could definitely be improved in the areas indicated by the tool, enabling a smoother synthesis flow. Based on results, Cypress plans to enhance Linting checks by incorporating AutoCheck within the design and verification flow going forward for all projects.

CONCLUSION

The goal of this effort was to provide design and verification engineers access to Formal applications through an understood environment. This would equip engineers with industry proven verification techniques to provide better IP quality without forcing them to fully understand the Formal techniques being used. While the implementation within Cypress' Verification Management System has been completed, there is still much work to do in proving out the usefulness of these applications. For instance, in generating coverage exclusions through CoverCheck, the obstacle of highly configurable IP through parameters has to be resolved to achieve maximum benefit. The same could be said for all other applications where this applies. Furthermore, execution time of RegisterCheck has to be reduced to make this application viable on large register sets. The task can be broken into smaller groupings of registers and other performance improvements need to be investigated, such as, black boxing and Formal profiling.

ConnectivityCheck looks to be a very beneficial tool for making sure interconnects are honored based on provided specifications. Functional simulations to validate these connections can take weeks or months to create and execute, whereas, Formal can prove them in hours. In an environment where automation can be implemented based on the specification the return can be quite significant.

Lastly, AutoCheck for enhanced Linting of Design RTL can be very beneficial. This is realized in having checks that the designer can execute before handing the IP off for verification. Better quality at the handoff point will eliminate cycles of learning and shorten the schedule to delivery.