DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 2024
MUNICH, GERMANY
OCTOBER 15-16, 2024

A Innovative Approach to Verify the SoC Integration using the Formal Property Verification

David Vincenzoni, STMicroelectronics s.r.l., Agrate, Italy (*david.vincenzoni@st.com*)

Marcello Dusini, STMicroelectronics s.r.l., Agrate, Italy (*marcello.dusini@st.com*)

Abstract— **Formal verification is a widely used technique for verifying digital Intellectual Properties (IPs) at block level. Formal Property Verification (FPV) and its various applications, such as linting, formal register checks, Clock Domain Crossing (CDC), and code coverage, are commonly used in the industry. This paper presents a novel approach which allows to extend the traditional scope of the FPV to verifying the IP integration within a System-on-Chip (SoC); starting with a simple description of the SoC in a spreadsheet, SystemVerilog Assertions (SVAs) are generated and then applied to SoC. This flow was tested on two of our products based on microcontroller architecture. The results clearly show that this approach is effective in detecting integration issues early in the design cycle, improving overall design quality and reducing development time.**

Keywords—Formal Verification; Design Verification; System on Chip Integration Verification

I. RELATED WORKS

As the complexity of System on Chip (SoC) designs continues to increase, the verification process becomes more challenging. The top-level verification of a SoC is a significant task that requires a considerable number of resources to ensure that the SoC is functional and meets the required specifications. In the SoC verification flow, it is assumed that all the integrated IPs are 100% verified at the block level: with the advent of formal methods, achieving some verification goals became more efficient; in fact, one could consider that FPV and other useful apps such as register check, linting, and formal coverage are now widely used for IP verification as standalone unit; and some teams even sign-off their IPs relying on formal flow only; however, verifying the integration of the same IPs at the top-level remains a crucial step in ensuring the overall functionality [1]. This means that:

a) IP integration: it is necessary to verify that all the blocks are connected correctly to the bus and accessible by the masters. This process enables the detection of integration connection bugs.

b) System functionality: it is essential to ensure that all the IPs are interacting properly with the system's resources (e.g., interrupt controller, pads, Direct Memory Access (DMA) devices). Doing so helps to identify system resources connection bugs.

c) Application functionality: it is necessary to confirm that the main application is working properly; therefore, some tests will be developed to involve multiple IPs working together to execute a simplified version of the application itself. This approach could be useful for catching system bottleneck bugs.

Traditionally, these tests are conducted using simulations that rely on a Universal Verification Methodology (UVM) test bench, representing the system being tested; within this environment, the CPUs run C code designed to exercise the aforementioned points [2]. A significant amount of the effort goes into the creation of the software component: initially, each peripheral requires the development of a specific low-level driver; then specialized tests are crafted to check reset values and the accuracy of various read and write operations. These tests reveal some weaknesses: since they typically use fixed values (e.g. 0xAAAAAAAA, 0x55555555), sometimes they do not match the sophistication of the registers design and do not cover them very well. These facts can make the verification process challenging and time-consuming; this complexity could result in overlooked flaws, such as the accidental swapping of data or address lines.

II. Goal

In this paper, we propose to extend the application of the FPV by using it at the SoC level. Our objective is to verify the IP integration (a) by creating targeted SVAs, aimed at proving that the peripherals are accessible by the bus master.

The methodology described should not be confused with formal register checks at the SoC level; it is not even intended to verify the whole formal correctness of the connection-matrix. On the contrary, it is specifically designed for verifying IP integration, i.e. simply that any transaction issued by any master completes successfully and correctly to its destination. This flow does not prevent the use of SoC dynamic simulation for verifying the system and application functionalities as described in b) and c) points on page 1.

We will demonstrate that our approach based on FPV will lead to more efficient and effective verification of SoCs memory space integration, reducing the time and resources required for top-level verification.

III. Application

A. *Prepare the SoC for verification*

The system-under-verification is to be set up, as depicted in Figure 1, swapping the Register Transfer Level (RTL) description of the bus masters with Assertion-Based Verification IPs (ABVIPs, a.k.a. Formal VIPs or Assertion IPs as referenced in [3] and [4]); this is done to supply the subsystem under test with assumptions that prevent illegal bus transactions. Black boxing of some parts is often necessary; behavioral parts of memory-models must be treated as black boxes because formal tools work on synthesizable components only; parts of the interconnection-matrix may also be black boxed, if they do not affect the logic that interacts with the peripherals (doing so can simplify the process of solving the set of formal equations). Black boxing operation is straightforward and performed automatically by any formal tool by avoiding the compilation of the block description file.

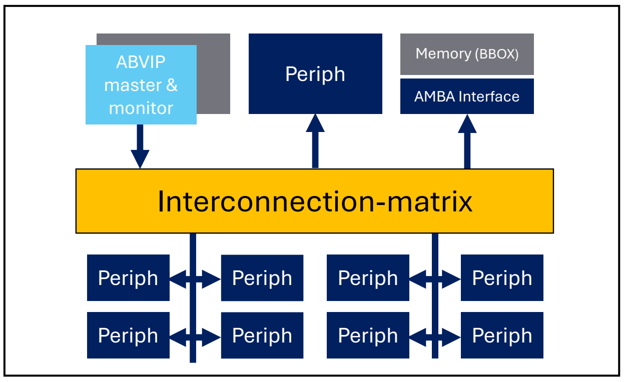


Figure 1. Example of a SoC architecture where the Master has been substituted with ABVIPs and the non-synthesizable parts have been black-boxed.

B. *Targeted Assertions*

The custom assertions created to achieve the goal are tailored for verifying peripheral integration by checking registers during reset, read, write and read-after-write operations. To maximize the capability of the assertions the notion of *undetermined data* is used; in formal verification, this term refers to data whose value is arbitrarily defined by the tool allowing it to represent any value. Doing so makes the tool capable of exploring all possible scenarios and verifying the correctness of the subsystem under any possible input conditions.

AHB and APB protocols (see [5] and [6]) were concretely used to develop this methodology, although it can be extended to a broader range of bus standards, in the future.

Now, an overview of the custom SVAs is presented:

1) Check reset value assertion

This assertion reads the reset value of the peripheral register to ensure it matches the expected value.

Equation (1) illustrates an example of SVA assertion for an AHB master driving an APB peripheral. The assertion remains disabled until the peripheral's reset is de-asserted. The APB_READ_RST_SEQ represents a sequence that defines the APB read access for reading data immediately after the reset, utilizing *parameters* to set the bus address of the relevant register to be verified. As the APB bus may introduce waiting cycles on AHB, the assertion checks the master HREADY signal for a low state. Once it transitions to a high state, the data is verified in the next cycle.

```
periph_i2c1_Reg_A_RST_VAL: assert property (disable iff (`DISABLE_RST_I2C1)
@(posedge `CLK_AHB)  APB_READ_RST_SEQ(`parameters)                              (1)
      |-> !`hreadyout_sig[*0:16] ##1 `hrdata_s_sig == `I2C_REG_A_RST_VAL);
```

2) Read access assertion

This assertion involves a reading sequence of peripheral's registers to verify their accessibility and the correctness of their values. It can occur at any time, not only upon exiting the reset.

Equation (2) provides an example of the SVA assertion for an AHB peripheral. The AHB_READ_SEQ is a sequence that defines the AHB read access based on *parameters* setting the bus address of the relevant register to be verified. Upon completion of the sequence, the data is checked in the same cycle: the data read on the master data bus must match the data generated at the peripheral's data bus.

```
periph_ahb_ram_MEM_RD: assert property (@(posedge `CLK_AHB)                      (2)
  AHB_READ_SEQ(`parameters) |-> `hrdata_s_sig == `AHB_RAM.hrdata);
```

Typically, this assertion is utilized to check the integration of RAM, ROM, and Flash, for which no memory array is available. The read data originates from the black box and is arbitrarily chosen by the formal tool.

3) Write access assertion

This assertion checks that peripheral registers can be written to and that their values are updated correctly by performing write-sequences on them.

Equation (3) illustrates an example of the SVA assertion for an AHB peripheral. The AHB_WRITE_DIRECT is a sequence that defines the AHB write access based on *parameters* setting the bus address of the relevant register to be verified and on the *undetermined data* (as described above on page 2). Upon completion of the sequence, the data is checked in the same cycle against the *undetermined data* itself.

```
periph_ahb_ram_MEM_WR: assert property (@(posedge `CLK_AHB)                      (3)
AHB_WRITE_DIRECT(`parameters, und_data) |-> `AHB_RAM.wdata == und_data);
```

This kind of assertion is crucial for checking the integration of RAM's, particularly in cases where no memory array is present.

4) Read after write:

This assertion writes a value to a peripheral's register and then reads it back to verify the success of the write operation and the accuracy of the value update.

Equation (4) illustrates an example of SVA assertion for an APB peripheral. The APB_WRITE_SEQ is a sequence that defines the APB write access; it is specialized thanks to *parameters* setting the bus address of the relevant register and relying on the *undetermined data*. After the idle cycle, the APB_READ_SEQ defines the APB read access; it uses the same *address* as the write sequence. As the APB bus may introduce waiting cycles on AHB, the assertion facilitates the check of the master's HREADY signal for a low state; once it transitions to a high state, the data is checked in the following cycle using the write-mask of the register.

```
periph_i2c_REG_A_WR2RD: assert property (@(posedge `CLK_AHB)
  APB_WRITE_SEQ(`parameters^W, und_data) ##1 APB_READ_SEQ(`parameters^R)  |->      (4)
!`hreadyout_sig[*0:16] ##1 (`hrdata_s_sig & 'mask) == (und_data & `mask));
```

5) Assertions summary

Table I. Targeted assertions for verifying IP integration within SoC.

SVA assertions			
Check reset value	***Read Access***	***Write Access***	***Read after Write***
Read of peripheral register reset value	Read sequence of peripheral register	Write sequence to peripheral register	Write and then read of peripheral register

C. *Automatic generation of the assertions*

In the context of verifying IP integration within a medium-complexity SoC, the number of potential SVA assertions can escalate to hundreds. Although it is possible to manually write all these assertions using the methodology outlined in the 'Targeted Assertions' section on page 2, this approach is both error-prone and time-consuming. To achieve the goal efficiently, a Python application was developed to generate the required SVAs (see Figure 2).

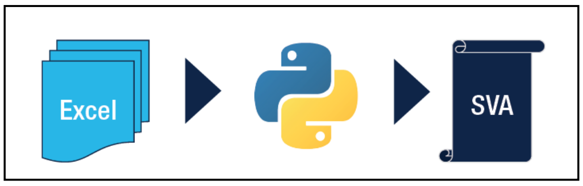


Figure 2. Automatic generation flow.

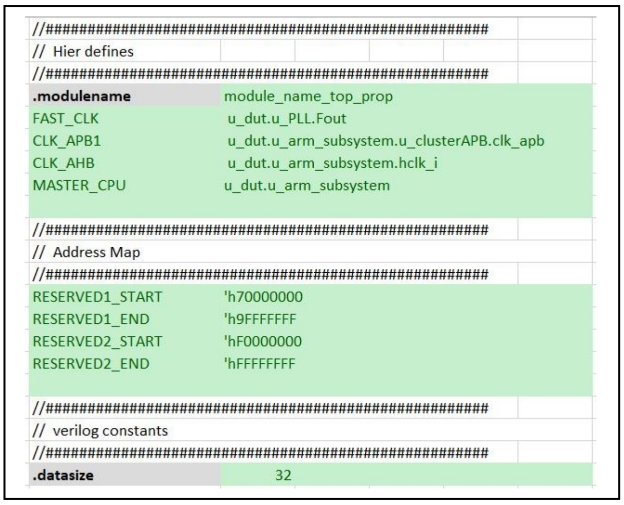
As a first step, this software must build an internal representation of the register map together with its constraints. The model will help identify relevant information necessary to create the assertions which must prove the correct connection between any master and its slaves. An Excel file must contain the description of memory space, clocks, resets, relationships between masters and slaves including information about paths and peripheral registers. The software takes this file as an input and elaborates it (with the aid of open-source libraries [7]) in order to build the model of the subsystem; based on the type of buses and bus-protocols which are detailed in the spreadsheet, the application generates proper assertions.

D. *Spreadsheets description*

The Excel file must adhere to specific rules to ensure parsing by the application. A variable number of additional sheets for masters and peripherals can be included. The currently supported protocols are AHB3 Lite, APB2 and APB3.

1) Macro and Reset Sheets

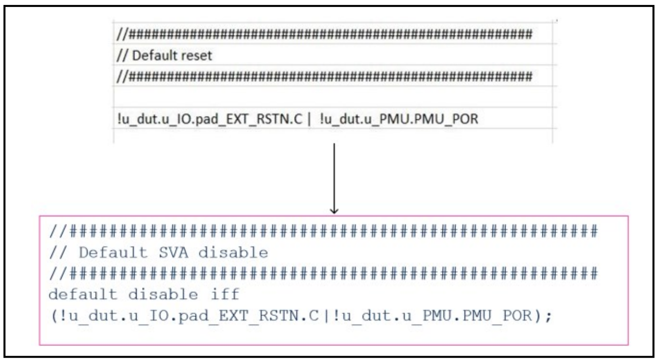
The Macro Sheet contains all relevant constants for the SoC being tested, such as hierarchical paths, the global memory map, and constants that improve the legibility of the assertions created. An example of this sheet is illustrated in Figure 3.



```
//########################################################
// Hier defines
//########################################################
.modulename        module_name_top_prop
FAST_CLK            u_dut.u_PLL.Fout
CLK_APB1            u_dut.u_arm_subsystem.u_clusterAPB.clk_apb
CLK_AHB             u_dut.u_arm_subsystem.hclk_i
MASTER_CPU         u_dut.u_arm_subsystem

//########################################################
// Address Map
//########################################################
RESERVED1_START    'h70000000
RESERVED1_END      'h9FFFFFFF
RESERVED2_START    'hF0000000
RESERVED2_END      'hFFFFFFFF

//########################################################
// verilog constants
//########################################################
.datasize                 32
```

Figure 3. Example of Excel Macro Sheet. Notice that **.modulename** and **.datasize** are *dotted keywords* which identify specific tokens.

The Reset Sheet allows for the definition of resets. In a SoC, reset expressions can be complex. The Python application translates this expression into the SVA syntax, as illustrated in Figure 4.



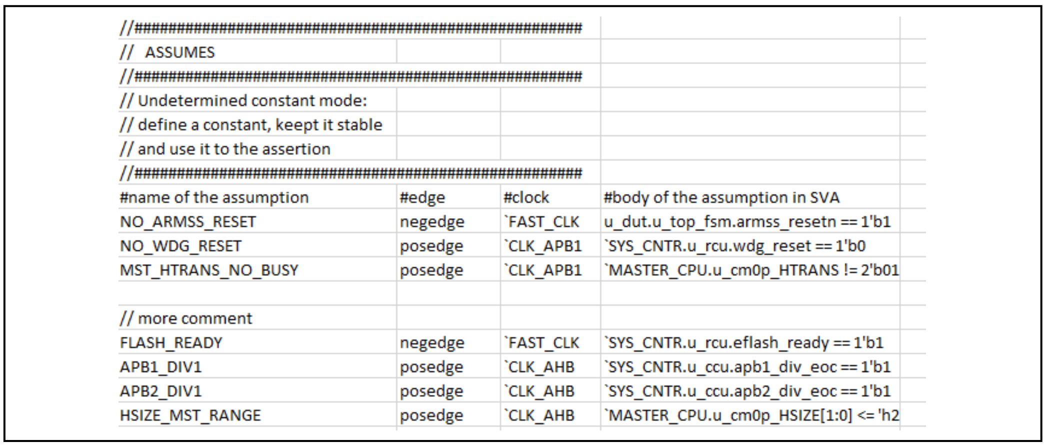
```
//########################################################
// Default reset
//########################################################

!u_dut.u_IO.pad_EXT_RSTN.C |  !u_dut.u_PMU.PMU_POR
```

```
//####################################################
// Default SVA disable
//####################################################
default disable iff
(!u_dut.u_IO.pad_EXT_RSTN.C|!u_dut.u_PMU.PMU_POR);
```

Figure 4. Example reset conversion from Excel sheet to SVA.

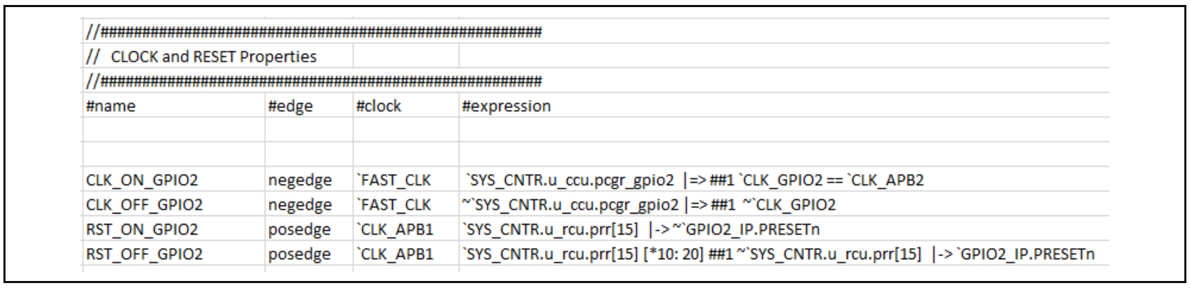
2) Global sheets

Formal verification often requires global assumptions and global assertions. An example of these sheets are reported in Figure 5 and Figure 6.



//##			
// ASSUMES			
//##			
// Undetermined constant mode:			
// define a constant, keept it stable			
// and use it to the assertion			
//##			
#name of the assumption	#edge	#clock	#body of the assumption in SVA
NO_ARMSS_RESET	negedge	`FAST_CLK	u_dut.u_top_fsm.armss_resetn == 1'b1
NO_WDG_RESET	posedge	`CLK_APB1	`SYS_CNTR.u_rcu.wdg_reset == 1'b0
MST_HTRANS_NO_BUSY	posedge	`CLK_APB1	`MASTER_CPU.u_cm0p_HTRANS != 2'b01
// more comment			
FLASH_READY	negedge	`FAST_CLK	`SYS_CNTR.u_rcu.eflash_ready == 1'b1
APB1_DIV1	posedge	`CLK_AHB	`SYS_CNTR.u_ccu.apb1_div_eoc == 1'b1
APB2_DIV1	posedge	`CLK_AHB	`SYS_CNTR.u_ccu.apb2_div_eoc == 1'b1
HSIZE_MST_RANGE	posedge	`CLK_AHB	`MASTER_CPU.u_cm0p_HSIZE[1:0] <= 'h2

Figure 5. Assumption Sheet to constraint the design

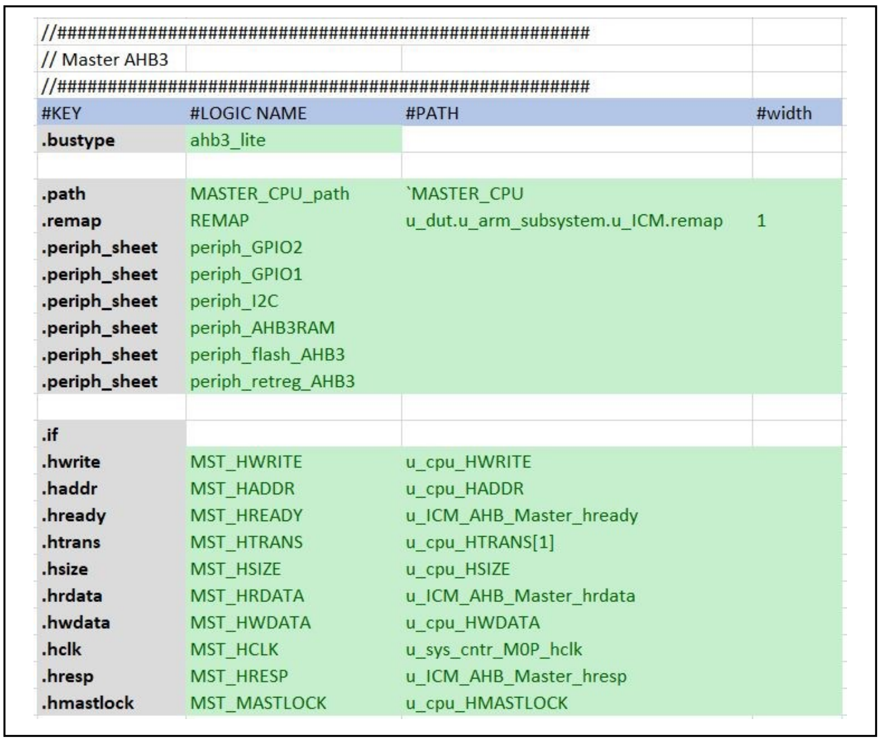


//##			
// CLOCK and RESET Properties			
//##			
#name	#edge	#clock	#expression
CLK_ON_GPIO2	negedge	`FAST_CLK	`SYS_CNTR.u_ccu.pcgr_gpio2 |=> ##1 `CLK_GPIO2 == `CLK_APB2
CLK_OFF_GPIO2	negedge	`FAST_CLK	~`SYS_CNTR.u_ccu.pcgr_gpio2 |=> ##1 ~`CLK_GPIO2
RST_ON_GPIO2	posedge	`CLK_APB1	`SYS_CNTR.u_rcu.prr[15] |-> ~`GPIO2_IP.PRESETn
RST_OFF_GPIO2	posedge	`CLK_APB1	`SYS_CNTR.u_rcu.prr[15] [*10: 20] ##1 ~`SYS_CNTR.u_rcu.prr[15] |-> `GPIO2_IP.PRESETn

Figure 6. Assertion Sheet for SoC level checks

3) *Masters Sheets*

The Master Sheets contain details including the specification of the bus type, the memory remapping and the addressed peripherals. Any signals of the bus master interface must be reported. It's important to remember that this section includes all the peripherals for which integration needs to be verified.

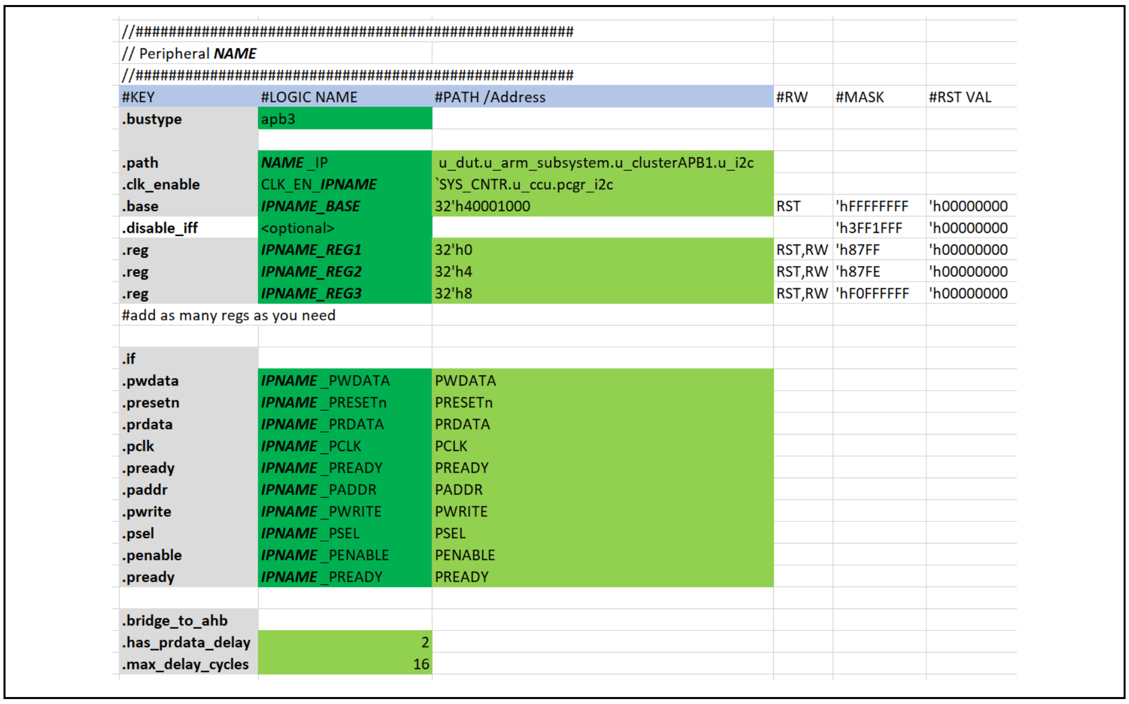


//##			
// Master AHB3			
//##			
#KEY	#LOGIC NAME	#PATH	#width
.bustype	ahb3_lite		
.path	MASTER_CPU_path	`MASTER_CPU	
.remap	REMAP	u_dut.u_arm_subsystem.u_ICM.remap	1
.periph_sheet	periph_GPIO2		
.periph_sheet	periph_GPIO1		
.periph_sheet	periph_I2C		
.periph_sheet	periph_AHB3RAM		
.periph_sheet	periph_flash_AHB3		
.periph_sheet	periph_retreg_AHB3		
.if			
.hwrite	MST_HWRITE	u_cpu_HWRITE	
.haddr	MST_HADDR	u_cpu_HADDR	
.hready	MST_HREADY	u_ICM_AHB_Master_hready	
.htrans	MST_HTRANS	u_cpu_HTRANS[1]	
.hsize	MST_HSIZE	u_cpu_HSIZE	
.hrdata	MST_HRDATA	u_ICM_AHB_Master_hrdata	
.hwdata	MST_HWDATA	u_cpu_HWDATA	
.hclk	MST_HCLK	u_sys_cntr_M0P_hclk	
.hresp	MST_HRESP	u_ICM_AHB_Master_hresp	
.hmastlock	MST_MASTLOCK	u_cpu_HMASTLOCK	

Figure 7. Example of Master Sheet. Notice the *dotted keywords* to identify the relevant parts of the design.

4) *Peripheral Sheets*

Similarly to the Master Sheets, the Peripheral Sheets include attributes such as the bus type, the Verilog/VHDL hierarchical path of the block and the base address. In the upper part, all the peripheral's registers, for which we want to generate the properties, are listed; for each of them several parameters are established: the address offset, the access policy (such as read-write, read-only or write-only), a bit mask where '1' represents the bits in the register that can be written to, and the reset value (nevertheless specific actions on peculiar flags, such as set-to-clear registers are not covered by the application, yet.). The signal names of bus peripheral interface shall be reported in the lower part of the sheet.



//##

// Peripheral ***NAME***

//##

#KEY	#LOGIC NAME	#PATH /Address	#RW	#MASK	#RST VAL
.bustype	apb3				
.path	***NAME*** _IP	u_dut.u_arm_subsystem.u_clusterAPB1.u_i2c			
.clk_enable	CLK_EN_***IPNAME***	`SYS_CNTR.u_ccu.pcgr_i2c			
.base	***IPNAME_BASE***	32'h40001000	RST	'hFFFFFFFF	'h00000000
.disable_iff	<optional>			'h3FF1FFF	'h00000000
.reg	***IPNAME_REG1***	32'h0	RST,RW	'h87FF	'h00000000
.reg	***IPNAME_REG2***	32'h4	RST,RW	'h87FE	'h00000000
.reg	***IPNAME_REG3***	32'h8	RST,RW	'hF0FFFFFF	'h00000000
#add as many regs as you need					
.if					
.pwdata	***IPNAME*** _PWDATA	PWDATA			
.presetn	***IPNAME*** _PRESETn	PRESETn			
.prdata	***IPNAME*** _PRDATA	PRDATA			
.pclk	***IPNAME*** _PCLK	PCLK			
.pready	***IPNAME*** _PREADY	PREADY			
.paddr	***IPNAME*** _PADDR	PADDR			
.pwrite	***IPNAME*** _PWRITE	PWRITE			
.psel	***IPNAME*** _PSEL	PSEL			
.penable	***IPNAME*** _PENABLE	PENABLE			
.pready	***IPNAME*** _PREADY	PREADY			
.bridge_to_ahb					
.has_prdata_delay	2				
.max_delay_cycles	16				

Figure 8. Example of Peripheral Sheet.

IV. RESULTS

This flow has been applied to two different SoCs. The first one is a single master platform with AMBA bus; it has a complexity of around 250K Gates. The second one is a multi-master SoC with security features on interconnect-matrix; its complexity is around 3M Gates. Table II summarizes the results in terms of number of generated assertions and property convergence time.

Table II. Results on two SoC cases.

Description	*Single Master SoC*	*Multi-master SoC with security features*
Design Complexity	250KGates	3MGates
Proved Assertions	More than 40	More than 130
Property maximum convergence time[a]	Few minutes	Some minutes
Property minimum convergence time	Few seconds	Few seconds

[a]. *Convergence time depends on allocated resources in terms of number of licenses and parallel jobs.*

In general, the full formal verification of the interconnect-matrix is a challenging process, and very often most of the properties remain inconclusive after many hours of computing. On the contrary, in this methodology, assertions pass in a few minutes or even seconds because the complexity of the state-space is restricted by properties designed for specific paths. In fact, our primary objective is the successful verification of the IP integration within the SoC.

V. Conclusions

The approach to SoC verification using formal methods, presented in this work, offers significant benefits in terms of efficiency, accuracy, and bug detection.

By verifying IP integration within SoC early in the flow, without the need of a UVM test bench, commonly discovered bugs can be quickly identified and resolved, reducing verification effort, and improving overall design quality; typical bugs that this flow is capable to catch are:

- Wrong memory map description
- Wrong data bus connection
- IP clock and/or reset stuck-at
- Wrong peripheral's reset value

Setting up the formal verification environment is faster than any UVM test bench, making this approach an efficient option for verification teams for testing the SoC IP integration. Notably, knowledge of the SVA language is not strictly required, as the assertions are automatically generated. Additionally, this structured setting formed by RTL list files and clock/reset definition could be reused for other analyses such as CDC, improving the efficiency of the verification process. Hence, the Formal techniques offer a comprehensive and effective way to uncover odd bugs and reduce the time and resources required for top-level verification.

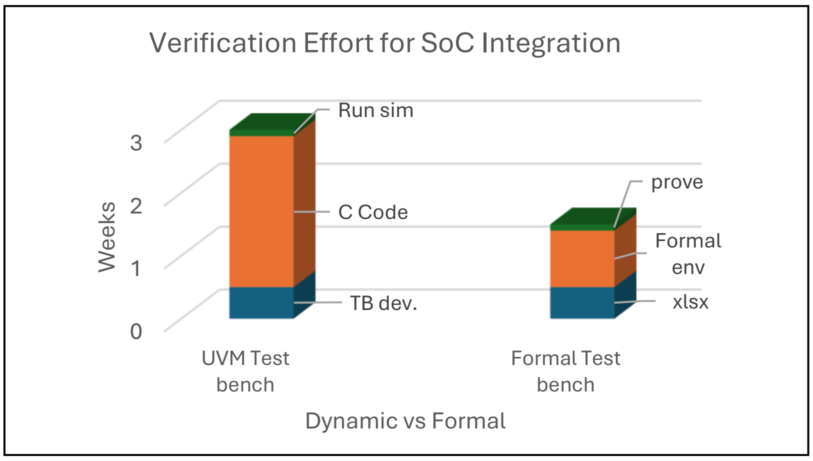


Figure 9. Comparison between Dynamic and Formal flows for the verification of SoC IPs integration. For a UVM testbench, the work involves the construction of the testbench, followed by weeks spent on C coding, and finally, some days dedicated to running and debugging the tests. In contrast, for the formal testbench, a few days are required to complete the Excel file that describes the SoC. Subsequently, approximately one week is needed to set up the formal environment, followed by a few days for proving and debugging the properties. This approach can result in a saving of up to 50% of the effort.

VI. References

[1] G. C. Spear, System Verilog for Verification, third ed., Springer, 2012.
[2] M. Balance, "Creating SoC Integration Tests with Portable Stimulus and UVM Register Models," [Online].
[3] Cadence, "formal VIP," [Online].
[4] Synopsys, "Assertion IP," [Online].
[5] ARM, "AMBA Specification".
[6] ARM, "AMBA 3 Specifications".
[7] "openpyxl," 2024. [Online].