2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

# Break the SoC with Random UVM Instruction Driver

Bogdan Todea, Microchip Technology, Inc., Bucharest, Romania
(*Bogdan.Todea@microchip.com*)

Pravin Wilfred, Microchip Technology, Inc., Bangalore, India

(*Pravin.Wilfred@microchip.com*)

Madhukar Mahadevappa, Microchip Technology, Inc., Bangalore, India

(*Madhukar.Mahadevappa@microchip.com*)

Diana Dranga, Microchip Technology, Inc., Bucharest, Romania

(*Diana.Dranga@microchip.com*)

***Abstract*— Today's System On a Chip (SoC) is made up of numerous peripherals, usually multiple processor cores and in-house and third-party IP's. Verifying such complex designs is even more challenging, due to various scenarios that come with such multitude of highly dependent IP's.**

**Integrating both C tests (containing the software instructions) and the Universal Verification Methodology (UVM) code (containing the rest of the testbench components aimed to verify the hardware) into a single verification testbench is a main methodology for SoC level verification. However, this approach brings limitations for the control, reusability and randomization of tests.**

**This paper demonstrates a technique that allows random generation and driving from an UVM Verification Environment of Central Processing Unit (CPU) instructions for SoC Verification, using a black-box approach for the design. The solution accommodates any design with one or more CPU cores, it is flexible for design changes and even for CPU Instruction Set Architecture (ISA) changes. The solution is fully compatible with UVM and non-UVM System Verilog (SV) based Verification Testbenches. Examples show how this method is implemented in a testbench that automatically adapts and works with any design configuration.**

***Keywords*— *SoC Verification, UVM, CPU instructions, Randomization***

## I. INTRODUCTION

There are different methodologies for SoC verification. The main challenge is that we should consider the software and hardware working together as the full Device Under Test (DUT) [1]. For this reason, usually, the code for the CPU core is written in C, then processed by a compiler, copied into the Flash Model and executed by the CPU during the simulation.

However, C tests solve only a part of the problem - the stimuli for the CPU instructions. There are other stimuli driving peripheral communications (I2C, SPI, UART etc.). Since the Universal verification components (UVCs) are already working at a module level, a natural solution is to reuse the UVM components from module to system. In this way, C tests contain the software code, while the SV test contains the stimuli for the hardware, including monitors, checkers and other components. There could be a communication protocol between the SV testcase and the C code creating more complex scenarios.

There are major disadvantages to using C tests together with SV code, like the challenge to communicate efficiently between SV and C, which decreases the controllability of the stimuli, becoming harder to implement more complex scenarios. There are limitations for randomization and reusability of tests as well.

There are several industry approaches which try to solve the problem of driving the CPU instructions with a System Verilog driver.

## II. INDUSTRY APPROACHES FOR DRIVING CPU INSTRUCTIONS WITH SYSTEM VERILOG

### A. *CPU Bus Functional Model (BFM) Approach*

As shown in Figure 1, The CPU is replaced by the SV model called CPU BFM and the test interacts directly with the model. For example, if the application wants to configure a register, the write task in the test sends the write command to the CPU model and the BFM in turn writes into the specific register.

The disadvantage of this model is that several design blocks are excluded from verification at SoC level: Flash Model, Non-Volatile Memory (NVM) Controller, CPU RTL and replaced with the Verification Testbench models. They are represented with RED in Figure 1.

While replacing the Flash Model by a verification testbench model is not a high risk, excluding from verification the NVM Controller and especially the CPU (the most important block of the design) is increasing the risk of missing important bugs (decreases the quality of verification).

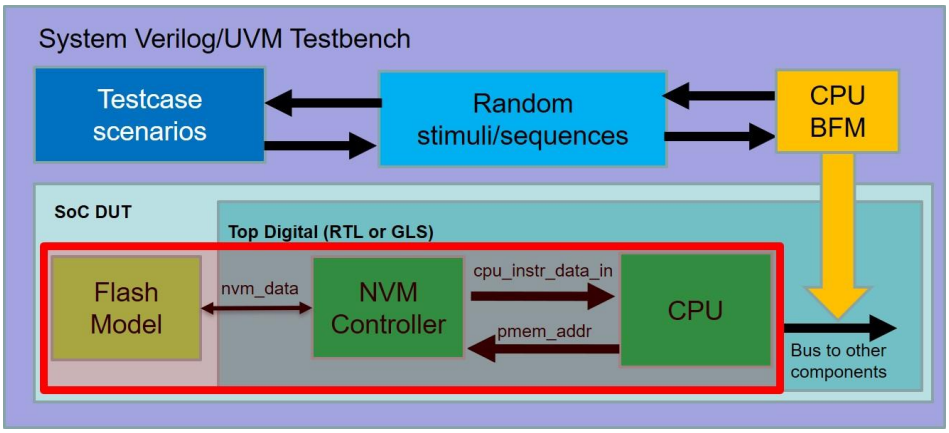


Figure 1. CPU BFM Approach

### B. *Instruction loader approach*

As shown in Figure 2, Instruction loader method does not replace the CPU with a model, but it bypasses the NVM controller on the instruction path, just before the CPU, by adding an "Instruction loader".

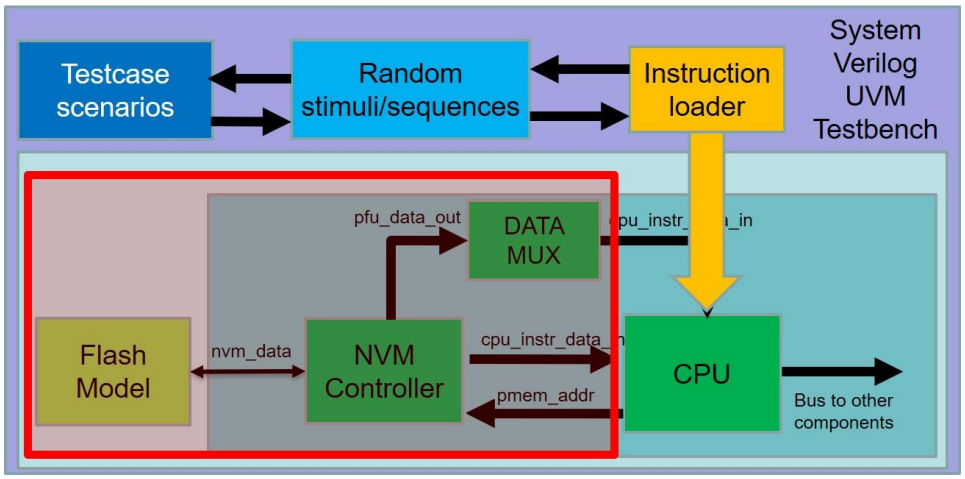


Figure 2. Instruction loader approach

The disadvantage with this approach is that there is no verification coverage for the interaction between the flash memory and the "replaced" module (NVM controller) because they are cut off as shown in Figure 2 (design blocks inside the red line are excluded for verification at SoC level).

Both cases represent a low robustness for Gate Level Simulation (GLS). It is very difficult to handle the case when DATA MUX doesn't have its output registered. In this case we need to drive data from DATA MUX with a constant delay to mimic the combinational delay which might vary from netlist to netlist.

## III. PROPOSED APPROACH FOR THE CPU INSTRUCTION DRIVING

The methodology proposed by this paper aims to complement the already existing C based tests verification.

The paper acknowledges that C based test verification is of critical importance for every SoC verification project, since we must verify in the same time the hardware and the software as a single DUT at SoC level. However, the gaps in verification of the C based methodology that were discussed in chapter I. can be covered as a complementary approach by the new proposed method.

In the proposed method, the CPU instruction code will be written directly and dynamically into the memory, using an instruction driver written in System Verilog, called the Instruction Transactor (IX). The IX is formed of two components:

- IX Driver – this component implements the translation of stimuli from the UVM testbench into instructions, based on the Instruction Set Architecture (ISA) of the CPU. This means that the IX is dependent on the ISA.
- IX2NVMIF – this component implements the tasks in charge of writing the instructions received from IX Driver into the NVM Flash Model. This block is dependent on each Flash Model.

By developing IX Drivers for different CPU Architecture types and IX2NVMIF for different Flash Model types, a library will be developed for different types of the IX Transactor.

The interface between the IX Driver and the UVM testbench will be standard, so we can reuse the verification stimuli and tests by carefully selecting the IX Driver based on the ISA and the IX2NVMIF based on the flash type.

The SV test will generate randomly constrained sequences that will be transmitted to the IX. The IX will convert it to CPU instructions, read the current program memory address and write it to the first flash location from where the CPU will fetch the next instructions in the next cycle.

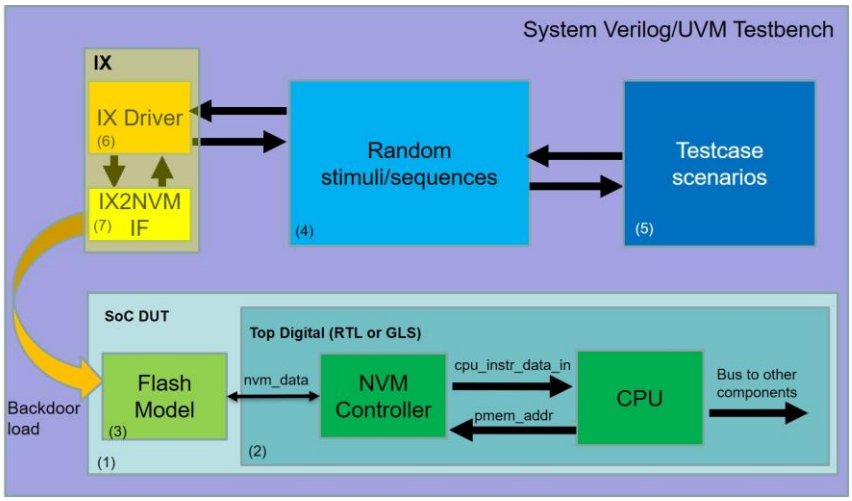


Figure 3. Backdoor load through the CPU Instruction Driver

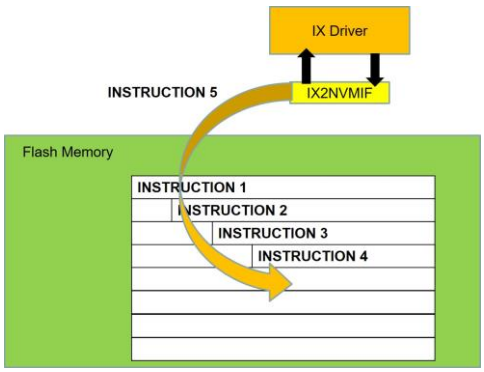


Figure 4. Detailed representation of IX loading opcode/instructions to flash

This new approach can be visualized as loading of binary file to flash dynamically when we want to perform any operation with CPU. IX driver would be monitoring the location from which the current instruction is fetched. Whenever it receives a new transaction from the test, it backdoor loads the flash. CPU is fetching the instruction from flash and executes it.

It is not necessary that all instruction in the ISA should be supported by the IX Driver. For SoC level verification, the requirement is to implement in IX Driver the instructions which are necessary to execute SoC level modes (SFR read/writes, any debug, sleep/idle instructions, CPU instructions taking a long time etc.). Most of the non-CPU tests ran at SoC level do not need all the instructions to be implemented.

## IV. Advantages of using a System Verilog CPU Instruction Driver

### A. *Ensuring the complete integrity of the Design under test (DUT)*

As can be seen in Figure 3, the proposed methodology does not require excluding any component of the design from the SoC verification. The proposed approach will treat the Top Digital as a black box. The only "intrusion" of the UVM Verification Environment inside the DUT is by writing the instructions within the Flash Model. This ensures the verification of connectivity for all the DUT components, from the NVM to the CPU and to the blocks communicating with the CPU (peripherals, Interrupt controller, DMA, RAM etc.).

### B. *High reusability of tests and testbench*

This methodology presents a high flexibility for design change and high reusability between different projects with different memory technologies, CPU ISA types and numbers of CPU cores. Using a layered hierarchy, the UVM testbench can be completely decoupled from the design architecture, flash technology and even from the CPU architecture/type.

There are different levels of reusability:

*1)* Reuse testcase scenarios from module to system

The methodology used in our case for SoC verification is based on reuse from Module to System not only of the verification testbench, but also the testcases. The reuse of testcases is done through a System Abstraction Layer (SAL), a system that will assure the right integration between the test and the testbench used for.

This means that any test for any peripheral can be reused, if written according to the SAL rules. For example, at module level, the SFR access can be done through the UVM reg map component, directly accessing the module level SFR access bus (the peripheral bus). For SoC level, the same SFR access will be through the IX. The testcase will contain just the specific task of writing or reading the SFR and the SAL will take care of handling the right method of SFR access, depending if the testbench is a module or system level one.

*2)* Reuse testcase scenarios and testbench from project to project

Testbench and testcase scenarios would be reused from project to project with minimal impact.

There are several possible differences between projects:

*a)* Different Flash technology

The dependency of different NVM types (depending on size, technology, model provider etc.) is resolved by creating an additional layer between the IX and the NVM model – the IX2NVM interface. One such interface can be created for each memory type, with a common interface to the IX driver.

*b)* Different CPU cores architecture or different ISA

In case of a different CPU Instruction Set Architecture, we select the corresponding IX driver, which is based on the ISA type.

The IX driver is a reusable component, created specifically for each CPU instruction set architecture, but with a common interface with the IX2NVMIF block and with the rest of the testbench.

The IX Driver can be automatically generated from the ISA Documentation.

*c)* Multi-core projects

In case of multi-core projects, the CPU instruction driver components would be instantiated one time for each core: one instruction data path component, consisting of one IX Driver (6) and one IX2NVMIF (7) for each core.

### C. *Seamless integration for Gate Level Simulations (GLS)*

A main advantage is that the IX is seamlessly integrated for gate level simulations. Since this methodology is a black-box approach of the DUT, no components of the design were excluded from verification (like in Figure 1 or Figure 2).

The gate level netlist should not require any change in order to have the IX driver plug-and-play. The Top Digital would be RTL or Gate Level netlist, depending on the type of the simulation. As can be seen in **Error! Reference source not found.**3, the type of the top digital component (RTL or netlist) is transparent to the Verification Testbench.

### D. *Randomization and Coverage*

The coverage that makes sense to define for chip level verification will cover just a small space of the total randomization space, since chip verification is more concentrated on verifying features across multiple blocks, versus module verification, which is concentrated more on randomization of features of each block.

There is a multitude of characteristics that can be randomized within a set of instructions at chip level. The biggest space for randomization is found with mathematical instructions: value of operands and working registers (WREGs) used for the operands (randomly choose any of the WREGs). Another randomization space is the randomization between the consecutive transactions, to create complex scenarios with different kind of hazards, random sequences of consecutive transactions etc.

The space of randomization for instructions can be covered very well at module level (through simulation based, formal verification or a mixed). For chip level, it makes sense to cross the instruction set coverage with cross-functional coverage of other blocks. This cross-functional coverage cannot be collected at module level.

The instruction set coverage is crossed at SoC level with interactions with other blocks (DMA, RAM, Interrupt Controller, peripherals).

One of the most suitable candidates for randomization were the CPU mathematical instructions, like Divide (DIV) and Multiply (MUL).

Tests have been developed, containing scenarios for a couple of mathematical instructions, each containing a cover group of its own.

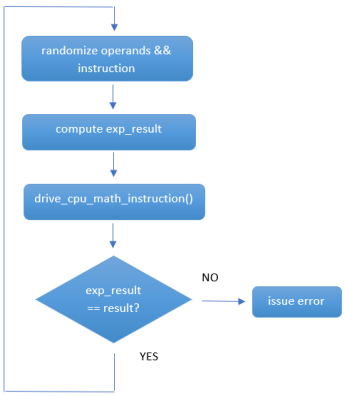


Figure 5. Simplified flow diagram for stimuli generation and checking

Below is a randomization example of the operands:

```
assert(randomize(Wm) with { Wm dist {[lower_part_Wm : (upper_part_Wm / 2)] := 50, [(upper_part_Wm / 2) : upper_part_Wm] := 50 }; });
assert(randomize(Wn) with { Wn dist {[lower_part_Wn : (upper_part_Wn / 2)] := 50, [(upper_part_Wn / 2) : upper_part_Wn] := 50 }; });
```

Figure 6. Randomization of the operands used

Below is an example of coverage for the randomized instructions scenarios:

```
covergroup cpu_mul_rand_cg with function sample(bit[31 : 0] Wm, bit[31 : 0] Wn, bit[63 : 0] result, ix_cpu_data_math_33a_c::cpu_instruction_type_e instruction);
   option.name = "cpu_mul_rand_cg";
   option.per_instance = 1;

   Wm_mul : coverpoint Wm {

      bins lower_half = {[min_value : (max_value / 2 - 'h01)]};
      bins mid        = {(max_value / 2)};
      bins upper_half = {[(max_value / 2 + 'h01) : min_value]};
   }

   Wn_mul : coverpoint Wn {

      bins lower_half = {[min_value : (max_value / 2 - 'h01)]};
      bins mid        = {(max_value / 2)};
      bins upper_half = {[(max_value / 2 + 'h01) : max_value]};
   }

   result_mul : coverpoint result {

      bins lower_half = {['h0000000000000000 : 'h7FFFFFFFFFFFFFFF]};
      bins upper_half = {['h8000000000000000 : 'hFFFFFFFFFFFFFFFF]};
   }

   Wm_mul_x_Wn_mul_x_Result : cross Wm, Wn, result, instruction;
endgroup: cpu_mul_rand_cg
```

Figure 7. Cover group used CPU instruction coverage

By using this approach, several RTL issues have been discovered in scenarios with the CPU DIV and CPU MULT instructions. Using randomization, the test was able to detect corner cases in which the instructions did not work correctly.

### *E. Execute Custom code*

We want to have a hookup in our verification environment to execute any instruction or series of instructions from the test case. In this mode, the user can specify the decoded instruction to be executed. This can be viewed as a function written in C or assembly and called from System Verilog code. This could be also helpful in simulating validation reported issues using the same binary file used for validation.

To reproduce the validation issue, it is very difficult to reuse the application code (C code) as it is, since there are no place holders for configuring the testbench components.

With the CPU IX feature, we can load the binary as it is to configure the DUT and the SV testbench can drive external stimulus and monitor the response.

### *F. Golden Model usage at chip level*

Randomization is requiring using some sort of golden model which can automatically compute the expected data, based on the driven stimuli and check it versus the returned data.

Golden Model can be modeled for each instruction aside, with relatively simple System Verilog functions. The example below shows a very simple check that replaced thousands of lines with hardcoded expected values, that were previously written in C or assembly testcases.

```
exp_result = (instruction inside { instr_A }) ? (Wm[15 : 0] * Wn[15 : 0]) : (Wm * Wn);

if(exp_result == Wresult) begin

   cpu_mul_rand_cg.sample(Wm, Wn, exp_result, instruction);

   `uvm_info("CPU_MUL_RAND: ", $sformatf("randomize_transaction(): results are equal: %0h, instruction: %0s", exp_result, instruction.name()), UVM_LOW)
end
else begin

   `uvm_error("CPU_MUL_RAND: ", $sformatf("randomize_transaction(): results are different, exp_result: %0h, Wresult: %0h, instruction: %0s", exp_result, Wresult, instruction.name()))
end
```

Figure 8. Example model for CPU instruction

## V. CHALLENGES IN IMPLEMENTING THE SYSTEM VERILOG CPU INSTRUCTION DRIVER

### *A. Interrupt handling*

IX supports Interrupt Service routine (ISR) handling for both SV based ISR and Assembly (ASM) based ISR code.

*1)* SV based ISR:

IX has two main channels to receive instructions from the test:

1. Software main channel
2. ISR channel

User code is sent to IX through software main channel and ISR code through ISR channel. By default, IX will be processing code from software main channel and when interrupt happens, IX will switch to ISR channel

and process instructions from this channel. Once ISR execution is complete, IX will switch back to Software main channel.

*2)* ASM based ISR:

Inside the Flash, the initial space can be allocated for Interrupt Vector Table (IVT) and ASM ISR code. User space will start after the ISR code placement, so there is no risk of over-writing the IVT or the ISR code in flash. When Interrupt happens, the CPU code execution jumps to ISR code and after completion it returns to the execution code in user space. With this approach, the ISR code will be pre-loaded in the flash model during compilation stage and so there is no backdoor loading of ISR code by IX. IX will only stop processing instructions in main channel and waits for ISR execution to complete.

### *B. Handling End of Memory*

In our approach, we are keeping the program control flow in the testcase. When CPU is fetching, the instruction program counter would be incremented linearly. If the testcase scenario is long, program counter may reach end of memory. Two scenarios need to be address here:

*1)* When there are no incoming transactions, CPU would still be executing (NOP instruction) and Program Counter (PC) would be incrementing. At this time, IX would be monitoring the PC and it will issue an instruction (GOTO) to CPU to move the program counter back to the reset vector.

*2)* When the incoming transaction can't be loaded into the remaining memory space, IX inserts an instruction to CPU to move program counter back to reset vector.

We don't see any issue with the insertion of extra instruction to change the program counter, since test case is agnostic about the time at which instruction is executed.

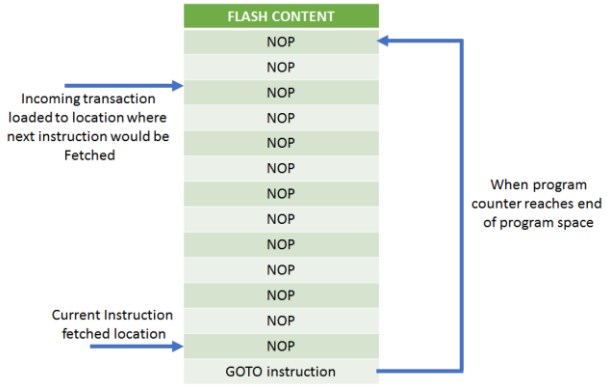


Figure 9. Handling of End of memory

### *C. Burst instructions*

IX supports burst write operation. Burst write transaction from test mentions the size of burst, based on the size, IX will backdoor load opcodes at one shot. As the PC increments, the CPU will execute the loaded opcodes sequentially.

Burst read operations are not supported, since in this case, the IX needs to wait for response from DUT and collect the read data and pass the read data to the test.

For write operations, IX does not wait for any response from DUT, so burst mode can be supported. This is useful for applications which must execute a predefined series of write items to unlock a feature or configure some set of registers in a very specific sequence.

## VI. LIMITATIONS OF THE CURRENT APPROACH

There are some limitations of the current approach:

- There is limited support for cache
- CPU execution for big designs will take a long time at chip level due to high boot times and latency introduced due to CPU execution itself
- The CPU instructions driver will implement only a subset of the complete ISA instructions (the ones which makes sense to be verified/executed at SoC level)
- This is not a complete solution. C based testcases remain a must, since SoC level verification should support simulating application code as well, usually written in C.

## VII. FUTURE IMPROVEMENTS

The list of future improvements contains:

- Automatic generation of the IX driver from the ISA documentation
- Load any sequence of instructions with any length (which can be loaded from a text file).
- Multiple core support

## VIII. CONCLUSION

By using the UVM CPU instruction driver, the majority of the SoC testcases can be completely written in UVM/SV, thus providing a high reusability of tests and testbench components. This approach allows more flexibility in generating complex scenarios for the SoC Verification.

It allows full randomization of the scenarios and easier portability of tests and testbenches between projects.

This methodology presents a high flexibility for design changes and high reusability between different projects with different memory technologies, CPU ISA types and number of CPU cores.

Using a layered hierarchy, the UVM testbench can be completely decoupled from the design architecture, flash technology and even from the CPU architecture type.

The entire SoC design connectivity will be verified, compared to other SoC verification methodologies, which exclude from connectivity verification certain design blocks.

An important advantage is that the IX Driver is seamlessly integrated for gate level simulations.

## REFERENCES

[1] Practical Approaches to SOC Verification by Guy Mosensoson, Verisity Design, Inc

[2] IEEE Std 1800-2012, "Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language"

[3] My Testbench Used to Break! Now it Bends: Adapting to Changing Design Configurations, Jeff Vance, Jeff Montesano, Kevin Vasconcellos, Kevin Johnston, Verilab Inc, DVCon USA 2018