Developing a Portable Block Testbench and Reusable SOC Verification Scenarios Using IP-XACT Based Automation.

Taejin Kim[1,2] Minjae Kim[2] Yonghyun Yang[2]
Hyundon Kim[2] Daewoo Kim[2] Tae Hee Han[1] Seonil Brian Choi[2]

[1]Department of Semiconductor and Display Engineering, Sungkyunkwan University, Suwon, Korea.
(kimrainy@g.skku.edu, than@skku.edu)
[2]Samsung Electronics, Hwaseong, Korea.
({mjstyle.kim, yh1597.yang, hyundon.kim, dw0912.kim, seonilb.choi}@samsung.com)

***Abstract*- This paper presents a portable block testbench architecture for the development of reusable SOC verification scenarios, and automated testbench generation by using IP-XACT. We explain how the portable block testbench is composed to improve the reusability of verification scenarios from the sub-system level to the SOC level. Next, we describe a way to automate portable block testbench generation for entire blocks in the SOC using IP-XACT. We applied this framework to our verification flow of premium SOC design. As a result, we can achieve an average simulation run time reduction of 65%~95%.**

I. INTRODUCTION

SOC design complexity and size are continuously increasing to meet market demands such as high performance and various use cases. Research for functional verification trends by Wilson Research Group from 2014 to 2016 shows that the percentage of IC/ASIC projects with design size over 40M gates was growing to 20%, 31%, and 38% in 2012, 2014, and 2016, respectively. Furthermore, the products over 500M gates were growing to 17%, 19% of 2014, 2016, as well [1], [2]. In 2018, although the percentage of projects with design size over 40M and 500M gates has reduced to 33% and 17%, respectively, due to an increase of very small designs (less than 100k gates) for IoT or automatic devices. These trends in design size show that the semiconductor industry continues to move in the direction of larger designs [3].

In the event-driven simulation for design function verification, which is currently widely used, the number of events that occurred during the simulation process determines the simulation running time (wall clock time). This relationship means the more events during simulation, the more computing needs and, eventually, the longer the simulation run time. Two main conditions, design size and length of verification scenario, determine the number of events that occur during simulation under the same time precision, which are proportional under the same clock frequency condition. In this view, developing a verification scenario using SOC design, which is a very large scale, is the worst choice in terms of the development time of design verification scenarios.

Verification scenario development time is closely related to the overall verification schedule. According to the research in the 2018 Wilson Research Group, 69% of projects were behind the original schedule in terms of design completion time, and the total verification time spent in the project was 53%. Besides, developing testbenches and verification scenarios, including simulation run time, accounts for 40% of the work of verification engineers [3]. Thus, shortening the development time for verification scenarios is an important factor in the overall verification schedule.

In this paper, a portable block testbench architecture based on UVM (Universal Verification Methodology) is proposed to reduce development time for verification scenarios and reuse those scenarios from a block-level to an SOC level testbench. Verification scenarios are developed through the portable block testbench, which merged block testbenches into the SOC testbench. Then, the developed verification scenarios are reused in the SOC testbench to reduce development time for verification scenarios. This testbench is automatically generated based on SOC design information, which is described in IP-XACT [4], to reduce maintenance overhead.

The rest of this paper is organized as follows: The next section briefly reviews the SOC verification scenario development time issue and SOC/block testbench architecture to motivate the considered work. Afterward, the proposed portable block testbench architecture and IP-XACT based testbench automation are described in Section III and Section IV. Finally, Section V presents our experimental evaluation, and Section VI concludes the work.

II. BACKGROUND

A. SOC Verification Scenario Development Time Issue

In general, a bottom-up verification approach is used, where IP level verification scenarios are reused at an upper-level verification such as a sub-system (or a block) level as well as an SOC level [5], [6]. However, a top-down verification approach is required when both IP level and upper-level verifications are conducted simultaneously, where IP level verification scenarios are not available in advance for reuse. Besides, a top-level verification such as SOC level one typically uses complex testbench structures and requires a significant amount of simulation run time.

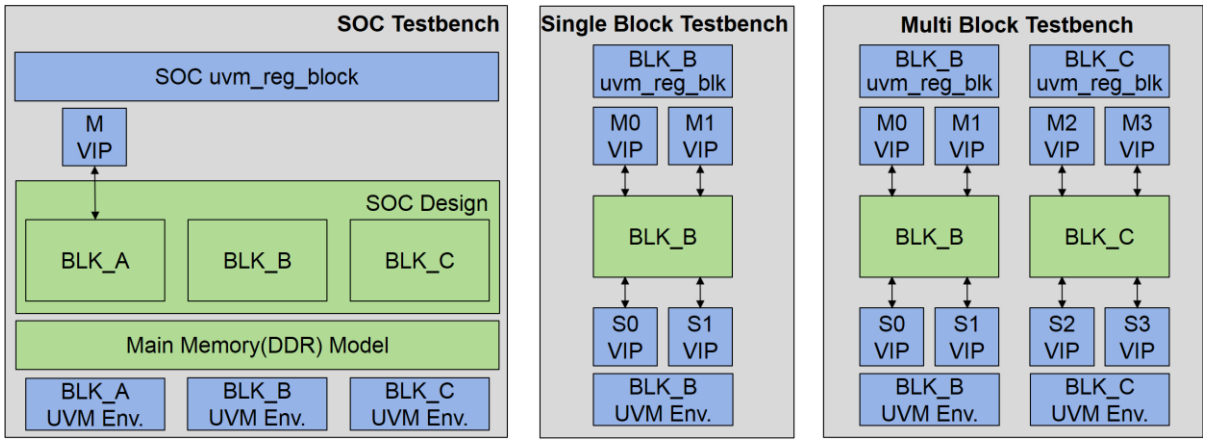
A stub-out approach, where unused blocks are stubbed out in the SOC level testbench, is used to reduce its long simulation time. However, the design blocks that must be included to use basic SOC systems, such as a backbone bus and memory controllers, cannot be replaced by the stub-out module. Therefore, there is a limit to shorten the development time for verification scenarios by reducing the design size while using the basic systems of the SOC.

Also, a single/multi-block testbench approach that uses only the required design blocks for verification purposes can further improve simulation performance. This approach reduces the debugging time compared to the testbench using an SOC design by narrowing and isolating the scopes of various issues, such as design/testbench bugs and register settings, that occur in the design under verification. However, this approach requires each verification engineer to build the appropriate testbench components, such as deployment/setup of verification IP (VIP) and connection with the design, for their design configuration of interest. Building testbench components demands a significant amount of manual effort. Moreover, further efforts are also needed to implement the verification scenarios developed at the single/multi-block testbench to the original SOC testbench.

B. SOC Testbench and Single/Multi Block Testbench

As shown in Fig. 1 (a), the UVM-based original SOC testbench consists of a multi-block SOC design and various verification components such as an SOC uvm_reg_block which models the internal registers of the SOC design, a master VIP which abstracts host CPU core, SOC main memory models like LPDDR memory, and UVM environments for block/function verification. In comparison, the testbenches using the single/multi-block design of Fig. 1 (b) and Fig. 1 (c) only use the design blocks required to verify the specific functions in an SOC design. Except for the VIPs for each design block, block-level testbench components, such as uvm_reg_blocks and block/function-specific UVM environments, can be used as part of the SOC testbench. Therefore, proper partition and separation of the verification components for SOC testbench enable the development of block and SOC level verification scenarios in the single merged testbench. Besides, UVM environments and verification scenarios in a block-level testbench can be vertically reused at an SOC level testbench.

As described earlier, keeping both block and SOC level testbenches together on a single testbench has the disadvantage of maintaining testbench components and verification scenarios but the advantage of reusability and portability. Besides, developing time for the SOC verification scenarios can be reduced by using the block-level testbench and the SOC level testbench in stages.



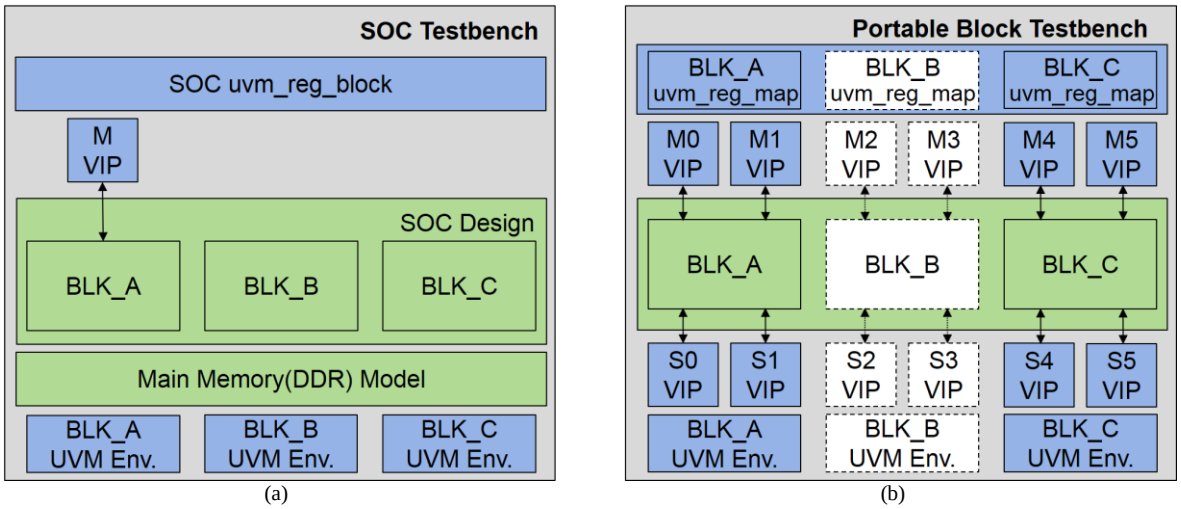
(a) (b) (c)

Figure 1. SOC and Single/Multi Block testbench architecture

III. PROPOSED PORTABLE TESTBENCH ARCHITECTURE

This section presents the proposed portable testbench architecture. This section also describes how to solve the reusability issue for verification scenarios and the portability of testbench components between the portable block testbench and the SOC testbench.

Fig. 2 show the original SOC testbench and the portable block testbench architecture, respectively. The portable block testbench automatically configures the necessary testbench components, such as master/slave VIPs and block-level uvm_reg_maps, which takes IP level uvm_reg_maps as sub-map, according to the design configuration.



(a) (b)

Figure 2. SOC/Portable block testbench architecture

A. *Compiler Directives Based Testbench Component Selection*

In the portable block testbench, all design blocks that are not subject to verification according to the design configuration are replaced by stub-out modules. Design configurations consist of `define statements or +define options, which indicates real or stub-out design modules. In the portable block testbench, the design configuration automatically determines the components that will be included in the testbench with compiler directives, such as `ifdef and `ifndef.

The design configuration in Fig. 3 shows the target and exclusion blocks for verification. The BLK_A and BLK_C are the target blocks, which are represented with {BLKNAME}_RTL_MT, and the BLK_B is the exclusion block, which is represented with {BLKNAME}_FAKE_MT. This design configuration is applied to the portable block testbench and the BLK_B related testbench components, such as the M2/M3 and the S2/S3 VIPS and the uvm_reg_maps of the BLK_B, are automatically excluded from the portable block testbench, as shown in Fig. 2 (b).

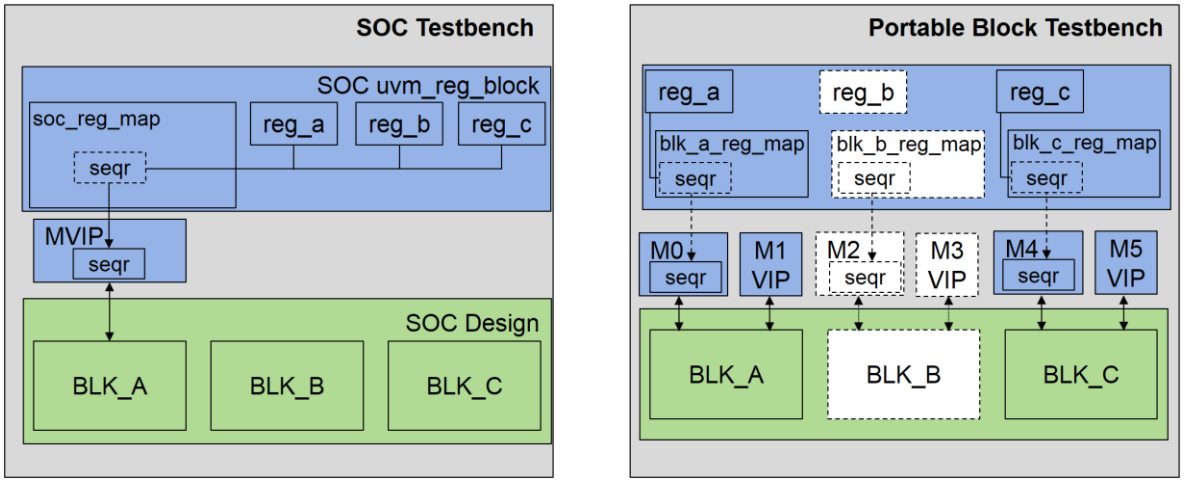
```
`define SOCDV_PORTABLE_BLKSIM   //Design configuration example
`define BLK_A_RTL_MT            //*RTL_MT  : original RTL module
`define BLK_B_FAKE_MT           //*FAKE_MT : stub-out module
`define BLK_C_RTL_MT
class tb_c extends uvm_env;
   …
   `ifdef SOCDV_PORTABLE_BLKSIM  //Component selection with compiler directives
      `ifndef BLK_A_FAKE_MT
        vip_env_c blk_a_m0_env;
        vip_env_c blk_a_m1_env;
      `endif
      `ifndef BLK_B_FAKE_MT
        vip_env_c blk_b_m0_env;
        vip_env_c blk_b_m1_env;
      `endif
      `ifndef BLK_C_FAKE_MT
        vip_env_c blk_c_m0_env;
        vip_env_c blk_c_m1_env;
      `endif
   `else
      vip_env_c host_cpu_m_env;
   `endif
   …
endclass
```

Figure 3. Example of design configuration and VIP declaration

B. Multiple Block uvm_reg_maps

The UVM register layer class is a useful abstraction mechanism that models memory-mapped registers and memories of hardware design, and it can be easily reused from IP level testbenches to SOC level testbenches. This UVM register layer class is widely used when developing testbenches and verification scenarios. In particular, accessing registers using the uvm_reg class tasks, such as read(), write(), update(), and mirror(), are most frequently used to develop verification scenarios. In a verification scenario, accessing registers using uvm_reg class tasks is issued from the master VIP through the associated sequencer of the uvm_reg_map, in which the uvm_reg class belongs to it.

In the case of the SOC testbench, as shown in Fig. 4 (a), a single consolidated uvm_reg_block has a single uvm_reg_map, and all IP-specific uvm_reg_maps construct sub-maps of the SOC uvm_reg_map. Therefore, register access tasks using the uvm_reg class are issued through the sequencer of the host CPU VIP, which is the associated sequencer of the single uvm_reg_map.



(a) (b)

Figure 4. uvm_reg_map structure in SOC and portable block testbench

In contrast, in the portable block testbench, each design block has its own master VIPs, so the register access tasks using the uvm_reg class must be issued on the master VIP of the appropriate design block. Besides, considering the reusability of verification scenarios between both testbenches, the register access tasks using the uvm_reg class should be able to be reused without any modification. To address the reusability issue, we applied multiple uvm_reg_maps structure to the portable block testbench, considering that a single uvm_reg_block can have multiple uvm_reg_maps, and each address map can have a different associated sequencer.

Fig. 4 (b) shows the multiple uvm_reg_maps structure described earlier. Each design block has its own uvm_reg_map, and uvm_reg_maps for all IPs inside the block are added as sub-maps. In the case of the sequencer in the master VIP, which is connected to the control bus interface, it is set to the associated sequencer for the corresponding design block. Therefore, when the register access tasks are executed in the verification scenarios, the appropriate VIP is automatically activated. This multiple uvm_reg_maps structure in the portable block testbench enables that register access tasks of uvm_reg class can be reused at the SOC testbench without any modification.

Fig. 5 shows the example code for the single uvm_reg_map structure and the multiple uvm_reg_maps structure according to the SOC and portable block testbench. It also shows an example of the associated sequencer setting for each uvm_reg_map.

```
class reg_blk_c extends uvm_reg_block;
   rand reg_a_c reg_a;                 //uvm_reg_block declaration
   rand reg_b_c reg_b;
   rand reg_c_c reg_c;
   …
   uvm_reg_map soc_reg_map;            //uvm_reg_map declaration and creation
   uvm_reg_map blk_a_reg_map;
   uvm_reg_map blk_b_reg_map;
   uvm_reg_map blk_c_reg_map;
   …
```

```
   reg_a = reg_a_c::type_id::create("reg_a"); //uvm_reg_block creation and add_submap
   reg_b = reg_b_c::type_id::create("reg_b");
   reg_c = reg_c_c::type_id::create("reg_c");
   `ifdef SOCDV_PORTABLE_BLKSIM
      `ifndef BLK_A_FAKE_MT
         blk_a_reg_map.add_submap(this.reg_a.default_map,`REG_A_BASE_ADDR);
      `endif
      `ifndef BLK_B_FAKE_MT
         blk_b_reg_map.add_submap(this.reg_b.default_map,`REG_B_BASE_ADDR);
      `endif
      `ifndef BLK_C_FAKE_MT
         blk_c_reg_map.add_submap(this.reg_c.default_map,`REG_C_BASE_ADDR);
      `endif
   `else
     soc_reg_map.add_submap(this.reg_a.default_map,`REG_A_BASE_ADDR);
     soc_reg_map.add_submap(this.reg_b.default_map,`REG_B_BASE_ADDR);
     soc_reg_map.add_submap(this.reg_c.default_map,`REG_C_BASE_ADDR);
   `endif
endclass
class tb_c extends uvm_env      // Associated sequencer setting for each uvm_reg_map
    `ifdef SOCDV_PORTABLE_BLKSIM
      `ifndef BLK_A_FAKE_MT
        reg_blk.blk_a_reg_map.set_sequencer(blk_a_m0_env.master.sequencer, blk_a_m0_reg_adp)
      `endif
      `ifndef BLK_B_FAKE_MT
        reg_blk.blk_b_reg_map.set_sequencer(blk_b_m0_env.master.sequencer, blk_b_m0_reg_adp)
      `endif
      `ifndef BLK_C_FAKE_MT
        reg_blk.blk_c_reg_map.set_sequencer(blk_c_m0_env.master.sequencer, blk_c_m0_reg_adp)
      `endif
   `else
     reg_blk.blk_c_reg_map.set_sequencer(host_cpu_m_env.master.sequencer, host_cpu_reg_adp)
   `endif
   …
endclass
```

Figure 5. Example code for single and multiple uvm_reg_maps structure

C. Slave Memory Mirroring

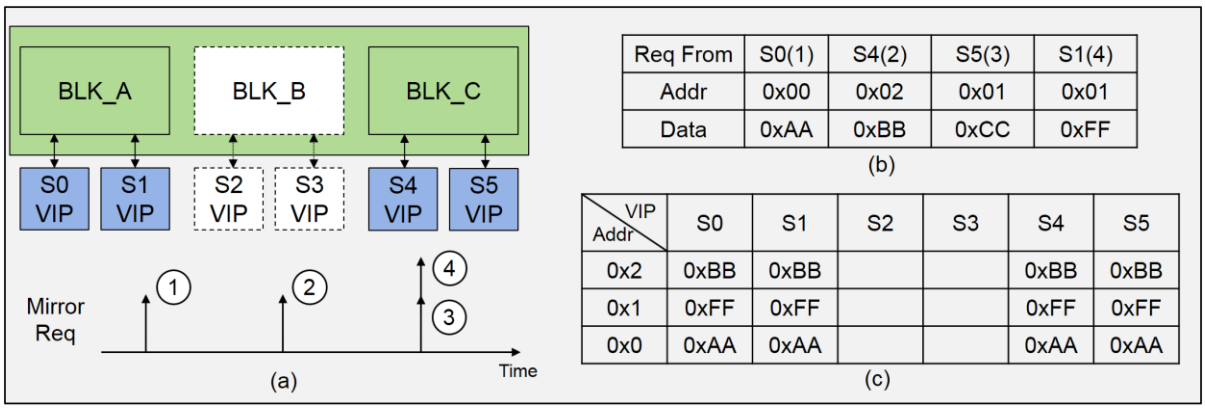
In the SOC design, data flow from each block to the main memory requires access to a bus network and memory controllers. In contrast, slave VIPs in the portable block testbench abstract the behavior of the bus network and memory controllers, so they act as sparse memories. However, multiple slave VIPs and sparse memories cause each block to have partial data only in their sparse memory space in case of the write transaction. To overcome this partial data sparsity, we configure the testbench to mirror the written data to the sparse memory of all slave VIPs. In this way, those data can be shared in all blocks. We call this slave memory mirroring.

The slave memory mirroring is performed on all slave VIPs whenever the final event of a data transaction, such as a burst end of AXI, occurs for each slave VIP in all blocks. When a mirroring request(MirrorReq) event occurs, the data is stored as a backdoor in the sparse memory across all slave VIPs by referring to the transaction information from the slave VIP that has received the actual one.

Slave memory mirroring occurs in the order in which the event occurred, and the real latency of the SOC bus network and the memory controller is not reflected. In addition, because the slave memory mirroring operates according to FCFS (First Come First Served) scheduling policy, it ensures that the same data is mirrored in the sparse memories of all slave VIPs based on the last event that occurred on the time axis. This means that the slave memory mirroring occurs in the order that the simulator accepts the MirrorReq event, even if the events occur simultaneously.

Fig. 6 shows an example of the slave memory mirroring. As shown in Fig. 6 (a), it is assumed that the MirrorReq events occurred in the order of VIP S0, S4, S5, and S1. It is also assumed that the VIP S5 and S1 raise MirrorReq events at the same time.

Also, if the address and data in the write transaction are issued as shown in Fig. 6 (b), the data by VIP S0 and S4 will be stored in the order of the sparse memory of all slave VIPs. Although the MirrorReq events from VIP S5 and S1 occurred at the same time, the data are mirrored in the sparse memory of the slave VIP, as shown in Fig. 6 (c), in the order in which the simulator accepted the event.



Req From	S0(1)	S4(2)	S5(3)	S1(4)
Addr	0x00	0x02	0x01	0x01
Data	0xAA	0xBB	0xCC	0xFF

VIP / Addr	S0	S1	S2	S3	S4	S5
0x2	0xBB	0xBB			0xBB	0xBB
0x1	0xFF	0xFF			0xFF	0xFF
0x0	0xAA	0xAA			0xAA	0xAA

Figure 6. Example of slave memory mirroring

D. Accessing Registers and Memories Using VIP Sequences

In addition to the access tasks of uvm_reg or uvm_mem, there are other methods to access the registers or memories inside the design from a master VIP by using VIP sequences. Since VIP sequences are also frequently used to access registers and memories in verification scenarios, reusability between an SOC and a portable block testbench should be considered.

In the original SOC testbench, a master VIP covers all accesses from a host CPU core to registers and memories in each block. Accessing registers and memories from the host CPU VIP is reached to the destination of each design block based on the address decoding information while passing through the bus network.

In the portable block testbench, in contrast, multiple master VIPs are used since each block has its own master VIPs and those VIPs model the behavior of the host CPU core and the bus network. However, this portability causes the routing information that the host CPU core and bus networks decode to be lost. To address this routing information loss, we develop a VIP sequence wrapper, which decodes the base address, and activates an appropriate master VIP. This VIP sequence wrapper enables VIP sequences to be reused in both the SOC testbench and the portable block testbench.

Fig. 7 (a) and Fig. 7 (b) show the example of the VIP sequence wrapper (rw_vseq) and its connections in the SOC and portable block testbench. In the SOC testbench, as shown in Fig. 7 (a), the VIP sequence in rw_vseq is connected directly to the sequencer in the host CPU VIP. So the host CPU VIP is always activated when rw_vseq is executed. In the portable block testbench, as shown in Fig. 7 (b), the VIP sequence is executed on the appropriate master VIP sequencer according to the input address of rw_vseq because the master VIPs in each block have their own address space. Below Fig. 8 shows the example code of rw_vseq as described earlier.

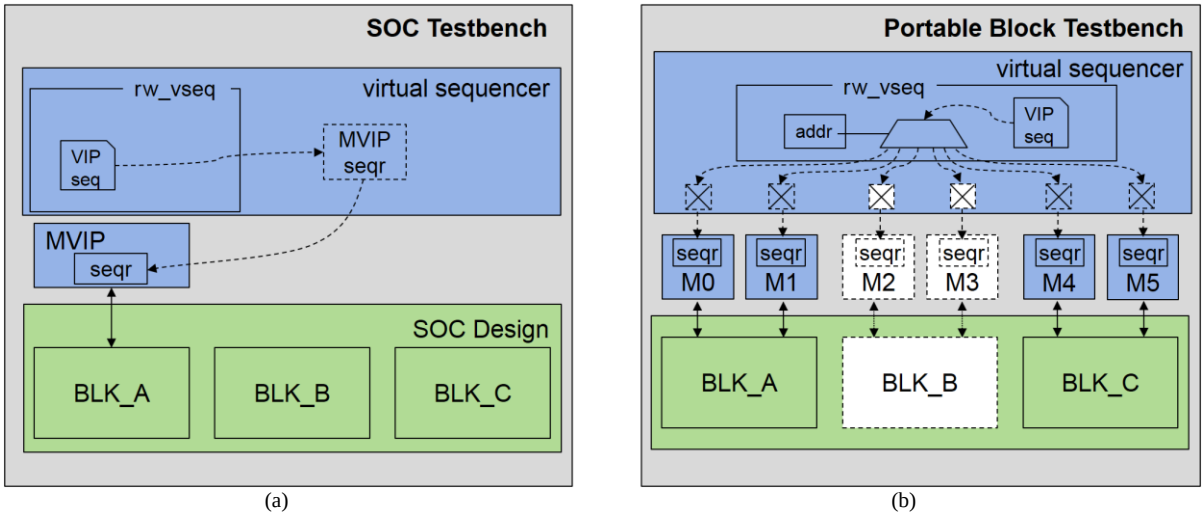


Figure 7. VIP sequence wrapper in SOC/portable block testbench.

```
class rw_vseq_c extends uvm_seq; //VIP sequence wrapper example code
  logic [`ADDR_WIDTH-1 : 0] addr;
  …
  vip_seq_c vip_seq;
  …
  virtual task body();
    …
    `ifdef SOCDV_PORTABLE_BLKSIM
      //each block VIP covers its own address space
     if (addr >= `M0_BASE_START && addr <= `M0_BASE_END)
         `uvm_do_on_with(vip_seq, p_sequencer.m0_seqr, {vip_rw_seq.addr == addr;…})
     if (addr >= `M1_BASE_START && addr <= `M1_BASE_END)
         `uvm_do_on_with(vip_seq, p_sequencer.m1_seqr, {vip_rw_seq.addr == addr;…})
     if (addr >= `M2_BASE_START && addr <= `M2_BASE_END)
         `uvm_do_on_with(vip_seq, p_sequencer.m2_seqr, {vip_rw_seq.addr == addr;…})
     if (addr >= `M3_BASE_START && addr <= `M3_BASE_END)
         `uvm_do_on_with(vip_seq, p_sequencer.m3_seqr, {vip_rw_seq.addr == addr;…})
     if (addr >= `M4_BASE_START && addr <= `M4_BASE_END)
         `uvm_do_on_with(vip_seq, p_sequencer.m4_seqr, {vip_rw_seq.addr == addr;…})
     if (addr >= `M5_BASE_START && addr <= `M5_BASE_END)
         `uvm_do_on_with(vip_seq, p_sequencer.m5_seqr, {vip_rw_seq.addr == addr;…})
      …
    `else
      //host CPU VIP covers all address space
     `uvm_do_on_with(vip_seq, p_sequencer.m_seqr, {vip_rw_seq.addr == addr;…})
    `endif
    …
  endtask:body
endclass:rw_vseq_c
```

Figure 8. Example code for the VIP sequence wrapper

E. Improved Verification Flow

The verification flow is improved using these major testbench changes, as shown in Fig. 9. Two different verification flows are introduced: one using a portable block testbench and the other using an SOC level testbench. In the initial stage of a verification cycle, the majority of verification scenarios are developed by using a portable block testbench. Once those verification scenarios are prepared, they are reused in the SOC level testbench to complete the SOC level verification.

IV. Automated Testbench Generation Based on IP-XACT

Maintaining both testbenches, as mentioned above, requires extensive effort. However, through the metadata described through IP-XACT, maintenance effort can be dramatically reduced through an automated testbench that reflects design information. IP-XACT is an XML schema that defines and describes IPs and design blocks. Metadata of design over IP-XACT enables the implementation of various design automation and testbench/verification automation [7]. The effort presented in [8], [9], has provided SOC design information described in the IP-XACT. Those works are leveraged to generate our testbenches automatically. In addition, IP-XACT based design automation from IP packaging to SOC assemble provides robust design metadata and also enabled stable testbench automation.

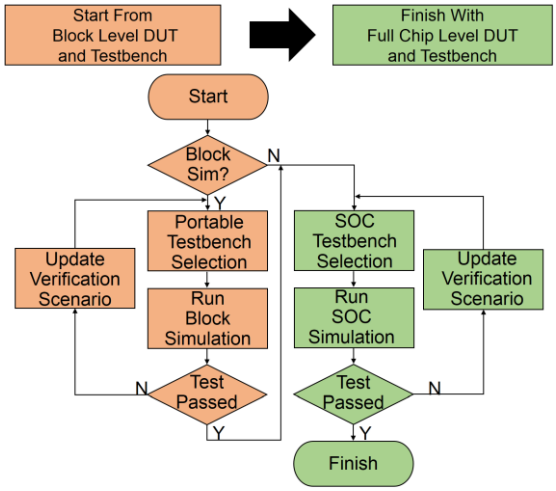


Figure 9. Improved verification flow chart

The IP-XACT information on the SOC design includes 1) asynchronous bridges and their signals (bus interfaces) for each design block, 2) registers in each design block, and 3) clock and reset information. Using this IP-XACT information, we can automatically create verification components such as VIPs, uvm_reg_blocks, and uvm_reg_maps, and compose portable block-level testbenches. This IP-XACT based testbench automation eliminates maintenance issues in a single merged testbench.

V. Experimental Result

The proposed portable block testbench architecture and automated testbench generation are applied to the design verification flow for our premium mobile SOC. Table 1 summarizes the comparison result for the simulation run time of three representative design blocks. We choose Cadence Xcelium 18.09 as a simulator, and server OS is Red Hat Enterprise Linux Server 7.3.

TABLE I
Simulation Run Time Comparison

Block Type	Number of Verification Scenarios	Simulation Run Time With SOC Testbench (min)	Simulation Run Time With Portable Block Testbench (min)	Simulation Run Time Reduction	Number Of Blocks
Security	39	5,239	129	97.5%	1
Storage	48	20,706	2,936	85.8%	1
Display	22	10,991	3,748	65.9%	4

In the security block, storage block, display blocks, 39, 48 and 22 verification scenarios, respectively, were developed using portable block testbenches. Comparing the simulation run times for all verification scenarios, we can see a 97.5% reduction of simulation time from 5,239 minutes to 129 minutes in the security block. The storage block shows an 85.8% reduction from 20,706 minutes to 2,936 minutes, and the display blocks show a 65.9% reduction from 10,991 to 3,748 minutes.

One thing to note is that display blocks with portable block testbench show a little bit lower simulation run time reduction compare to the others. Because the portable block testbench for security or storage block consists of a single design block and, in contrast, the display block consists of four design blocks.

Developed verification scenarios are 100% vertically reusable from the portable block testbenches to the SOC testbench. In generating testbenches, manually developing testbench might have taken at least three days for each block while our automatic testbench generation takes 30 minutes for the entire block.

VI. Conclusion

In this research, the time to develop SOC level verification scenarios is dramatically reduced by using portable block testbenches and by automating a testbench generation based on IP-XACT. It is also shown that the major testbench structure changes improve the reusability of verification scenarios developed at block level to SOC level. The experimental results show that the simulation run time can be reduced by 65%~95%. This proposed approach has been widely used in our SOC level verification.

Acknowledgment

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-00421, AI Graduate School Support Program) and the MOTIE(Ministry of Trade, Industry & Energy) (10080594) and in part by KSRC(Korea Semiconductor Research Consortium) support program for the development of the future semiconductor device.

References

[1] H. Foster, “Trends in function verification trends: A 2014 study”, *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015.

[2] H. Foster, “Trends in function verification trends: A 2016 Industry study”, *Proceedings of the Design and Verification Conference and Exhibition US (DVCon)*, Feb. 2017

[3] H. Foster, “Prologue: The 2018 Wilson Research Group Functional Verification Study”, Retrieved from http://reurl.kr/3FB52051BG

[4] IEEE. *"1685-2009 IEEE standard for IP-XACT, Standard Structure, Packaging, Integrating, and Reusing IP within Tool Flows."*.

[5] H. Zhaohui, et al. "Practical and efficient SOC verification flow by reusing IP testcase and testbench." *2012 International SoC Design Conference (ISOCC)*. IEEE, 2012

[6] V.S. Rashmi, S. Giridhar, and B. Sirisha "A methodology to reuse random IP stimuli in an SoC functional verification environment." *2015 19th International Symposium on VLSI Design and Test*. IEEE, 2015.

[7] Y. Cho, Y. Kim, and S. B. Choi, “IP-XACT based SoC interconnect verification automation”, *Proceedings of the Design and Verification Conference and Exhibition US (DVCon)*, Feb. 2018.

[8] W. Lee, Y. Kim, S. B. Choi, “IP-XACT based on design automation for micro architecture specification”, *Design Automation Conference(DAC) Designer/IP Track*, Jun. 2017.

[9] A. Kwon, W. Lee, Y. Kim, Y. Kim, and S. B. Choi, “Topology Specification Management based on IP-XACT”, *Design Automation Conference(DAC) Designer/IP Track*, Jun. 2018.