Metadata Based Testbench Generation Automation

Daeseo Cha[1], Soonoh Kwon[2], Ahhyung Shin[3], Youngnam Youn[4], Youngsik Kim[5], Seonil Brian Choi[6]
Samsung Electronics Co., Ltd., Seoul, Korea
([1]dscha; [2]soonoh1.kwon; [3]ah0403.shin; [4]yyn612; [5]ys31.kim; [6]seonilb.choi@samsung.com)

Abstract: This paper introduces the concept of automated testbench generation techniques using metadata of design spec. It focuses on full chip level structural testbench for register and interconnect verification. We demonstrate what contents need to be captured in metadata and how to automate UVM (Universal Verification Methodology) testbench generation with metadata.

I. Introduction

Generally, the functional verification environment has been developed by referencing functional specification and RTL design implementation specification together. As the size and complexity of SOC design continues to increase and the development time spent for SOC (System on a Chip) design is required to be shortened, it is very important to follow the variation of functional spec and implement them into RTL design properly. Moreover, the functional spec of design is changed frequently during development cycle. If these changes are not managed systematically, it makes the discrepancy between functional spec and RTL implementation spec.

Register and interconnect verification are the basic features of SOC verification, and they need to be done in early stage of verification with the highest priority. Register verification checks whether all IPs are controlled correctly through register access transaction. Interconnect verification checks whether all masters and slaves are accessed correctly on backbone bus through random data access transaction. Usually, register verification environment consists of register model and address map (uvm_reg) for each IP, a VIP agent replacing CPU. Interconnect verification environment consisted of VIP agents replacing all masters and slaves. They are highly dependent on the design implementation spec coming from design team. The problem is that verification engineers collect this information by referring to design specification documents and it is likely to insert human-errors when making verification environment, hence the inconsistency between design and documents happens. If there is this kinds of error, it takes long time for debugging because it is running on full-chip. In order to remove this problem, this paper proposes metadata based testbench generation automation approach which is aligned with metadata based SOC design integration automation.

II. Proposed Approach

IP-XACT is the IEEE standard metadata format (IEEE1685) which includes an information for full chip RTL integration(https://www.accellera.org/downloads/standards/ip-xact). Mainly it consists of two parts, one is IP package that includes port list, Interface, and SFR spec, and the other is design architecture that includes IP instance, connection between IPs. However, it is not enough for automating testbench generation (such as security, parameters required for VIP configuration, etc.) so we improved this by extending the contents of metadata. Newly defined metadata contains data for design automation of system IPs such as backbone interconnect, clock and power management unit and so on. In addition, we added more information for testbench automation such as controlling sequences, into the extended metadata. With this extended metadata, we are able to automate RTL design and testbench generation.

Figure 1 shows the proposed full chip register and interconnect testbench generation flow using metadata. vBuilder(Verification Builder) parses metadata to populate required information for testbench generation, such as IP address map, registers, location where masters and slaves are located, interface protocols and signal for masters/slaves, from metadata (IP-XACT), and then it renders target project testbench with existing register testbench template (SFR TB Template) and interconnect testbench templates(BUS TB Template), which are reusable for all projects. Generated testbench by vBuilder consists of 1) instantiating VIP (Verification IP), 2) connecting VIP ports to RTL signals, 3) generating complete test scenarios.

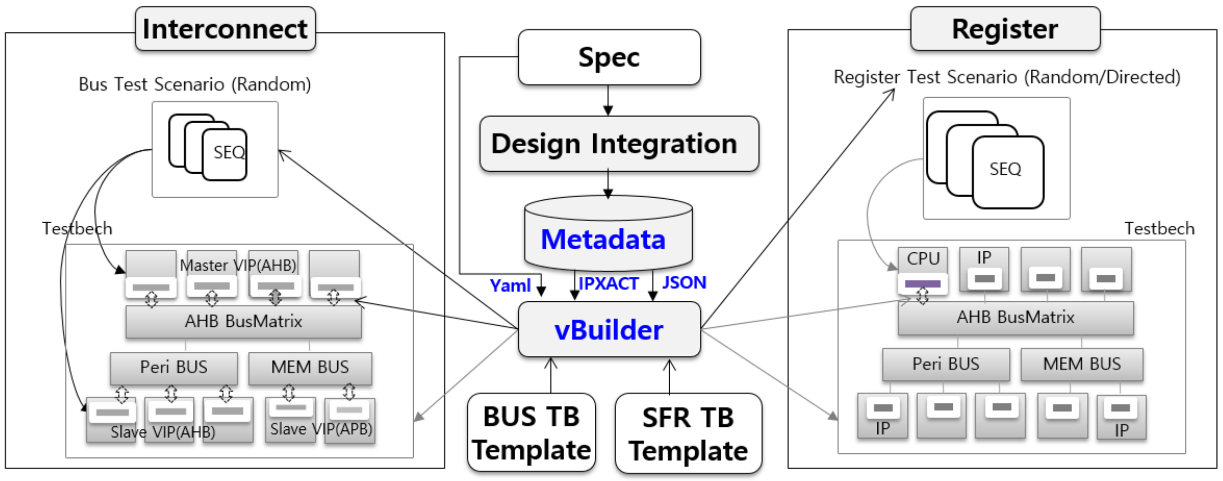


Figure 1. Metadata based Testbench Generation Flow.

A. Metadata

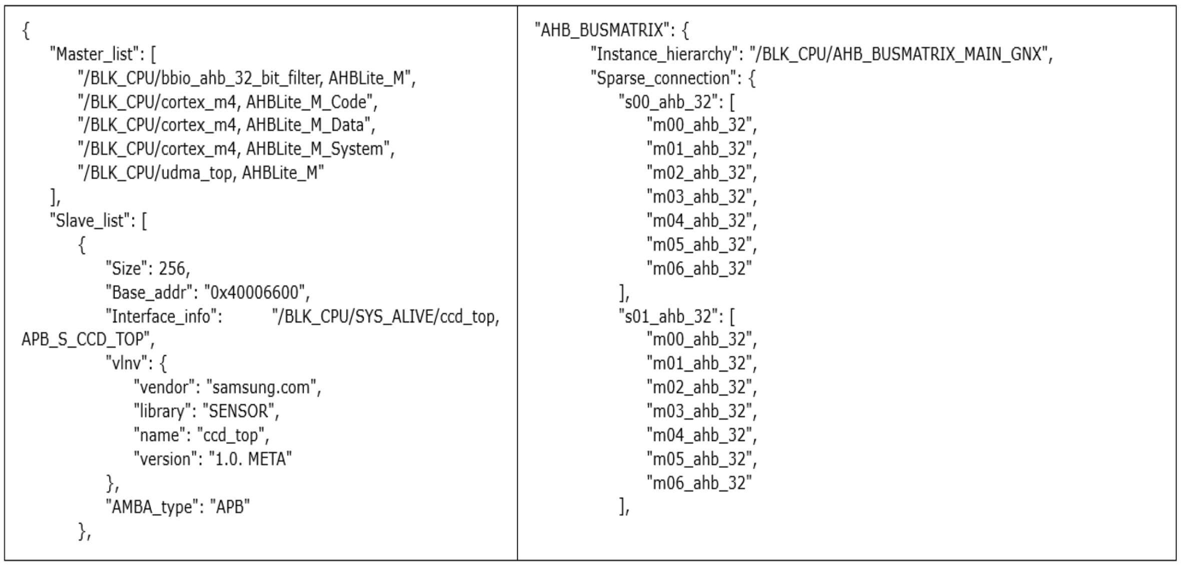
IP-XACT provides an IP connectivity standard for full chip RTL integration. Fig 2 shows the content which is described by JSON (JavaScript Object Notation), extracted from IP-XACT. It contains useful information for generating verification environment such as hierarchical instance name, interface name for masters and slaves, interface parameters and so on. They are used to integrate VIP which generates random bus transaction and checker to monitor DUT response through the interface in verification environment.

```
"/BLK_CPU/AHB2APB_CPU_0": {
            "interfaces": {
                "AHB00_S": {
                    "mode": "slave",
                    "connection":
"/BLK_CPU/AHB_BRIDGE_CPU,AHB02_MS",
                    "portmap": {
                        "HADDR": "HADDR",
                        "HCLK": "HCLK",
                        "HRDATA": "HRDATA",
                        "HREADY": "HREADY",
                        "HREADYOUT": "HREADYOUT",
                        "HRESETn": "HRESETn",
                        "HRESP": "HRESP",
                        "HSELx": "HSEL",
                        "HSIZE": "HSIZE",
                        "HTRANS": "HTRANS",
                        "HWDATA": "HWDATA",
                        "HWRITE": "HWRITE"
                    }
                },
```

```
"PRDATAS0": {
                    "width": "32",
                    "direction": "in",
                    "slices": {
                        "31,16": {
                            "connection": [
                                "/BLK_CPU/AHB2APB_CPU_0,PRDATAS0,15,0",
                                "/BLK_CPU/AHB2APB_CPU_0,PRDATAS0,31,16",
                                "/BLK_CPU/AHB2APB_CPU_0,PRDATAS1,15,0",
                                "/BLK_CPU/AHB2APB_CPU_0,PRDATAS1,31,16",
                                "/BLK_CPU/TAP,p_rdata,15,0"
                            ]
                        },
                        "15,0": {
                            "connec
                            "connection": [
                                "/BLK_CPU/AHB2APB_CPU_0,PRDATAS0,15,0",
                                "/BLK_CPU/AHB2APB_CPU_0,PRDATAS0,31,16",
                                "/BLK_CPU/AHB2APB_CPU_0,PRDATAS1,15,0",
                                "/BLK_CPU/AHB2APB_CPU_0,PRDATAS1,31,16",
                                "/BLK_CPU/TAP,p_rdata,15,0"
                            ]
```

Figure 2. Metadata for Interfaces.

Fig 3 shows the contents for **advanced** information which describes where masters and slaves are located, how connections are made between masters and slaves, what address spaces are used for slaves, what interface protocol is used for a slave and so on. It is necessary information to generate full chip interconnection testbench.

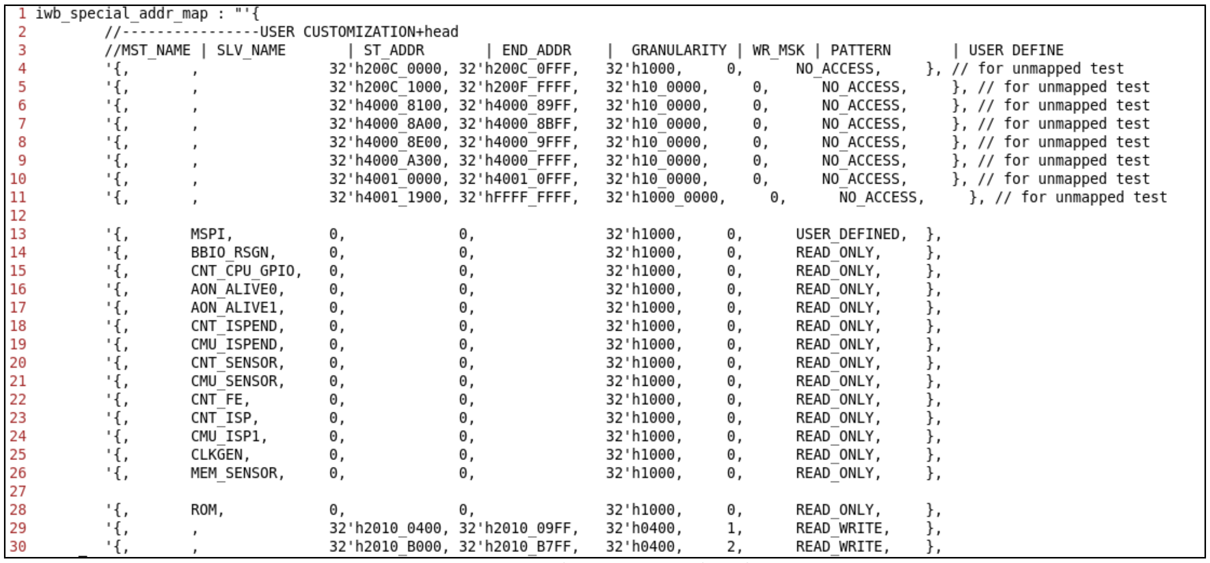


```
{
    "Master_list": [
        "/BLK_CPU/bbio_ahb_32_bit_filter, AHBLite_M",
        "/BLK_CPU/cortex_m4, AHBLite_M_Code",
        "/BLK_CPU/cortex_m4, AHBLite_M_Data",
        "/BLK_CPU/cortex_m4, AHBLite_M_System",
        "/BLK_CPU/udma_top, AHBLite_M"
    ],
    "Slave_list": [
        {
            "Size": 256,
            "Base_addr": "0x40006600",
            "Interface_info":        "/BLK_CPU/SYS_ALIVE/ccd_top,
APB_S_CCD_TOP",
            "vlnv": {
                "vendor": "samsung.com",
                "library": "SENSOR",
                "name": "ccd_top",
                "version": "1.0. META"
            },
            "AMBA_type": "APB"
        },
```

```
"AHB_BUSMATRIX": {
        "Instance_hierarchy": "/BLK_CPU/AHB_BUSMATRIX_MAIN_GNX",
        "Sparse_connection": {
            "s00_ahb_32": [
                "m00_ahb_32",
                "m01_ahb_32",
                "m02_ahb_32",
                "m03_ahb_32",
                "m04_ahb_32",
                "m05_ahb_32",
                "m06_ahb_32"
            ],
            "s01_ahb_32": [
                "m00_ahb_32",
                "m01_ahb_32",
                "m02_ahb_32",
                "m03_ahb_32",
                "m04_ahb_32",
                "m05_ahb_32",
                "m06_ahb_32"
            ],
```

Figure 3. Metadata for Interconnect Topology.

Fig 4 shows the contents for **project specific information** which needs to be considered separately. It is a code snippet of address map for a project, which is used by interconnect verification environment that generates random transactions to backbone bus. Line 2~11 describes unmapped logic and line 13~28 describes read-only region. This information is used for the policy of stimulus and checker. Since it is extended metadata for automating verification environment, it is hard to describe them with existing IP-XACT only. So we chose **yaml** to model this contents because it is simple to describe, has existing interface with Jinja for rendering.



```
 1 iwb_special_addr_map : "'{
 2        //----------------USER CUSTOMIZATION+head
 3        //MST_NAME | SLV_NAME       | ST_ADDR       | END_ADDR    |  GRANULARITY | WR_MSK | PATTERN       | USER DEFINE
 4        '{,       ,               32'h200C_0000, 32'h200C_0FFF,   32'h1000,     0,      NO_ACCESS,     }, // for unmapped test
 5        '{,       ,               32'h200C_1000, 32'h200F_FFFF,   32'h10_0000,     0,      NO_ACCESS,     }, // for unmapped test
 6        '{,       ,               32'h4000_8100, 32'h4000_89FF,   32'h10_0000,     0,      NO_ACCESS,     }, // for unmapped test
 7        '{,       ,               32'h4000_8A00, 32'h4000_8BFF,   32'h10_0000,     0,      NO_ACCESS,     }, // for unmapped test
 8        '{,       ,               32'h4000_8E00, 32'h4000_9FFF,   32'h10_0000,     0,      NO_ACCESS,     }, // for unmapped test
 9        '{,       ,               32'h4000_A300, 32'h4000_FFFF,   32'h10_0000,     0,      NO_ACCESS,     }, // for unmapped test
10        '{,       ,               32'h4001_0000, 32'h4001_0FFF,   32'h10_0000,     0,      NO_ACCESS,     }, // for unmapped test
11        '{,       ,               32'h4001_1900, 32'hFFFF_FFFF,   32'h1000_0000,     0,      NO_ACCESS,     }, // for unmapped test
12
13        '{,       MSPI,           0,             0,               32'h1000,     0,      USER_DEFINED,  },
14        '{,       BBIO_RSGN,      0,             0,               32'h1000,     0,      READ_ONLY,     },
15        '{,       CNT_CPU_GPIO,   0,             0,               32'h1000,     0,      READ_ONLY,     },
16        '{,       AON_ALIVE0,     0,             0,               32'h1000,     0,      READ_ONLY,     },
17        '{,       AON_ALIVE1,     0,             0,               32'h1000,     0,      READ_ONLY,     },
18        '{,       CNT_ISPEND,     0,             0,               32'h1000,     0,      READ_ONLY,     },
19        '{,       CMU_ISPEND,     0,             0,               32'h1000,     0,      READ_ONLY,     },
20        '{,       CNT_SENSOR,     0,             0,               32'h1000,     0,      READ_ONLY,     },
21        '{,       CMU_SENSOR,     0,             0,               32'h1000,     0,      READ_ONLY,     },
22        '{,       CNT_FE,         0,             0,               32'h1000,     0,      READ_ONLY,     },
23        '{,       CNT_ISP,        0,             0,               32'h1000,     0,      READ_ONLY,     },
24        '{,       CMU_ISP1,       0,             0,               32'h1000,     0,      READ_ONLY,     },
25        '{,       CLKGEN,         0,             0,               32'h1000,     0,      READ_ONLY,     },
26        '{,       MEM_SENSOR,     0,             0,               32'h1000,     0,      READ_ONLY,     },
27
28        '{,       ROM,            0,             0,               32'h1000,     0,      READ_ONLY,     },
29        '{,       ,               32'h2010_0400, 32'h2010_09FF,   32'h0400,     1,      READ_WRITE,    },
30        '{,       ,               32'h2010_B000, 32'h2010_B7FF,   32'h0400,     2,      READ_WRITE,    },
```

Figure 4. Metadata for project specific information

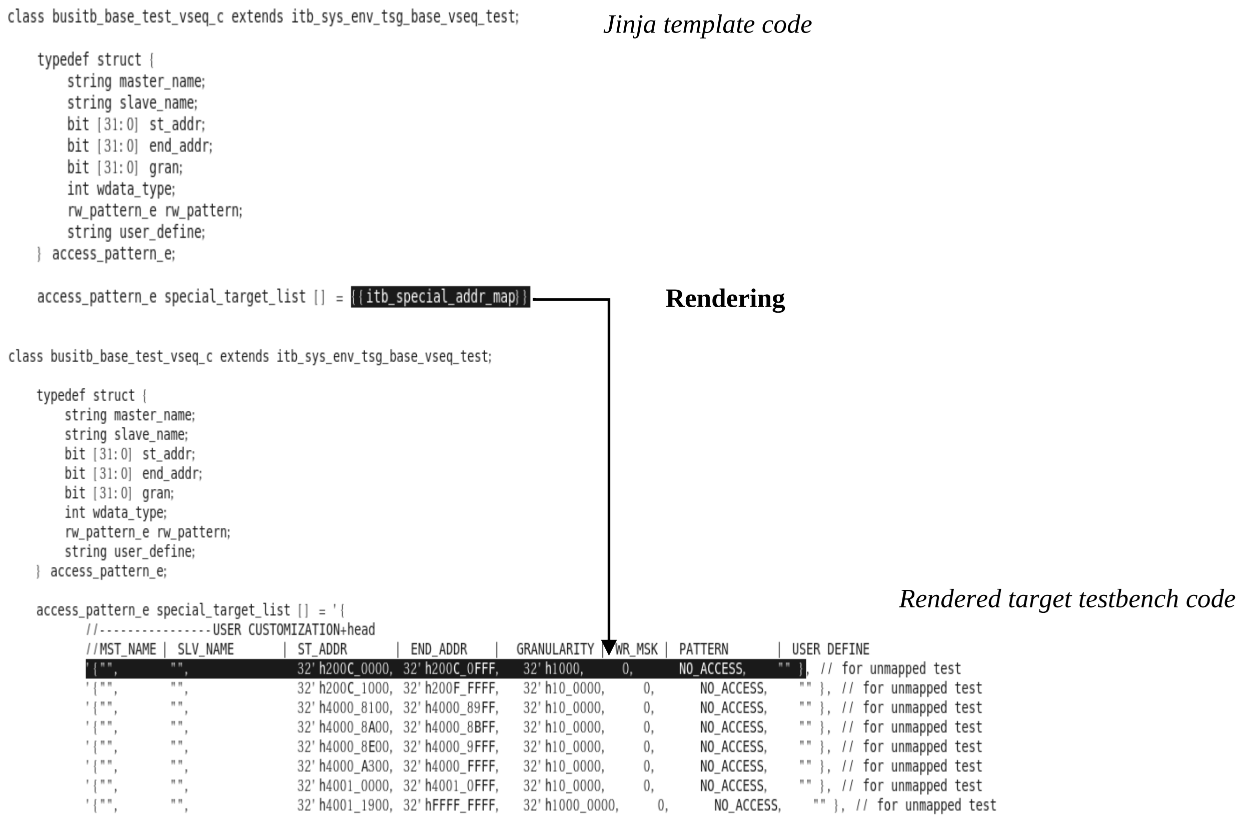
B. Testbench Generation

The previous approach to create full-chip interconnect testbench was a manual process:

- Step 1: Extract required information for testbench from design specification documents
- Step 2: Generate testbench initial version
- Step 3: Modify testbench code according to RTL implementation spec
- Step 4: Clean-up testbench (build & compile)
- Step 5: Start verification

Current approach is able to automate from step 1 to step 4 using metadata. vBuilder (In-house verification builder to generate various target testbench such as generic IP, full-chip testbench or application specific testbench generation for register, interconnect) parses metadata of either json or IP-XACT, and generates UVM testbench

vBuilder uses **Jinja** to handle project specific metadata when generating testbench like Fig 5. It renders target testbench code by combining jinja template code with yaml data file.



```
class busitb_base_test_vseq_c extends itb_sys_env_tsg_base_vseq_test;

    typedef struct {
        string master_name;
        string slave_name;
        bit [31:0]  st_addr;
        bit [31:0]  end_addr;
        bit [31:0]  gran;
        int wdata_type;
        rw_pattern_e rw_pattern;
        string user_define;
    }  access_pattern_e;

    access_pattern_e special_target_list [] = {{itb_special_addr_map}}
```

```
class busitb_base_test_vseq_c extends itb_sys_env_tsg_base_vseq_test;

    typedef struct {
        string master_name;
        string slave_name;
        bit [31:0]  st_addr;
        bit [31:0]  end_addr;
        bit [31:0]  gran;
        int wdata_type;
        rw_pattern_e rw_pattern;
        string user_define;
    }  access_pattern_e;

    access_pattern_e special_target_list [] = '{
        //----------------USER CUSTOMIZATION+head
        //MST_NAME | SLV_NAME       | ST_ADDR        | END_ADDR     |  GRANULARITY |WR_MSK | PATTERN       | USER DEFINE
        '{"",        "",              32'h200C_0000, 32'h200C_0FFF,   32'h1000,       0,       NO_ACCESS,     "" }, // for unmapped test
        '{"",        "",              32'h200C_1000, 32'h200F_FFFF,   32'h10_0000,      0,       NO_ACCESS,     "" }, // for unmapped test
        '{"",        "",              32'h4000_8100, 32'h4000_89FF,   32'h10_0000,      0,       NO_ACCESS,     "" }, // for unmapped test
        '{"",        "",              32'h4000_8A00, 32'h4000_8BFF,   32'h10_0000,      0,       NO_ACCESS,     "" }, // for unmapped test
        '{"",        "",              32'h4000_8E00, 32'h4000_9FFF,   32'h10_0000,      0,       NO_ACCESS,     "" }, // for unmapped test
        '{"",        "",              32'h4000_A300, 32'h4000_FFFF,   32'h10_0000,      0,       NO_ACCESS,     "" }, // for unmapped test
        '{"",        "",              32'h4001_0000, 32'h4001_0FFF,   32'h10_0000,      0,       NO_ACCESS,     "" }, // for unmapped test
        '{"",        "",              32'h4001_1900, 32'hFFFF_FFFF,   32'h1000_0000,      0,       NO_ACCESS,     "" }, // for unmapped test
```

Figure 5. Rendering testbench code using Jinja

C. Configurable Testbench Generation

By default testbench configuration, all masters and slaves are replaced with active VIP (Verification IP) components in interconnect testbench. Active VIP master generates bus transaction to backbone bus and active VIP slave responds to backbone bus. This configuration can be used from early verification stage. However, once real IP RTLs are available later, they need to be included in interconnect testbench for performance verification or stress tests. To do that, our new approach can support this configuration update easily by changing parameters from SYS.CSV file. Fig 6. shows how to configure SYS.CSV. If “attach active agent” will be set to “N”, RTL IP will be used from interconnect testbench.

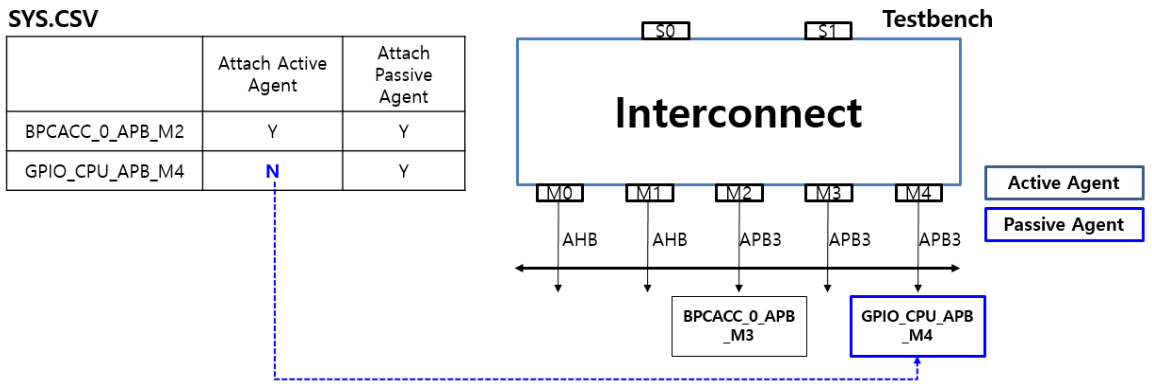


Figure 6. Configurable Interconnect Testbench Generation

D. *Sign-off for Register Sanity Test using Metadata*

IP-XACT and RTL are mandatory deliverables on IP hand-off and it needs to be guaranteed that IP-XACT and RTL are functionally equivalent. To reinforce this, this check requirement is added to IP sign-off system and this check can be done automatically by leveraging register testbench automation using metadata above.

IP-XACT provides the way of describing register with various attributes and they can be modeled using uvm_reg library in UVM testbench. In order to increase the quality of register checking in IP sign-off system, the functional behaviors in UVM testbench for IP-XACT tags for **testable, constraint and coverage** related tags in Table 1, was customized.

Metadata (IP-XACT)	Testbench (UVM_REG)
<spirit:field>	class hsi_unipro16_pa_std_region_pa_txhsg2synclength_pa_txhsg2synclen_msb_c
<spirit:name>PA_TxHsG1SyncLength_MSB</spirit:name>	extends uvm_reg_field;
<spirit:**writeValueConstraint**>	**constraint valid { value inside { ['h0:'h1] }; }**
<spirit:minimum>0</spirit:minimum>	
<spirit:maximum>1</spirit:maximum>	endclass :
</spirit:writeValueConstraint>	hsi_unipro16_pa_std_region_pa_txhsg2synclength_pa_txhsg2synclen_msb_c
</spirit:field>	
<spirit:register>	class hsi_unipro16_component_region_comp_option_suite_c extends uvm_reg;
<spirit:name>COMP_OPTION_SUITE</spirit:name>	rand uvm_reg_field reserved_ff0;
<spirit:addressOffset>0x20</spirit:addressOffset>	rand uvm_reg_field rx_symbol_clk1_reset_type;
<spirit:size spirit:resolve="immediate">32</spirit:size>	rand uvm_reg_field rx_symbol_clk0_reset_type;
<spirit:field>	
<spirit:name>Reserved_FF0</spirit:name>	covergroup cg_vals;
<spirit:bitOffset>2</spirit:bitOffset>	option.per_instance = 1;
<spirit:bitWidth	reserved_ff0: coverpoint reserved_ff0.value[29:0]
spirit:resolve="immediate">30</spirit:bitWidth>	{
</spirit:field>	**`AUTO_COV_MIN_MAX(30)**
<spirit:field>	}
<spirit:name>rx_symbol_clk1_reset_type</spirit:name>	rx_symbol_clk1_reset_type: coverpoint rx_symbol_clk1_reset_type.value[0:0];
<spirit:bitOffset>1</spirit:bitOffset>	rx_symbol_clk0_reset_type: coverpoint rx_symbol_clk0_reset_type.value[0:0];
<spirit:bitWidth	endgroup
spirit:resolve="immediate">1</spirit:bitWidth>	
</spirit:field>	define AUTO_COV_MIN_MAX(VAL**)** **₩**
<spirit:field>	bins min = {0}; ₩
<spirit:name>rx_symbol_clk0_reset_type</spirit:name>	bins mid = {[1:({VAL{1'b1}})-1]}; ₩
<spirit:bitOffset>0</spirit:bitOffset>	bins max = {{VAL{1'b1}}};
<spirit:bitWidth	

spirit:resolve="immediate">1</spirit:bitWidth> </spirit:field> </spirit:register>	

<spirit:register>
    <spirit:name>PA_AvailTxDataLanes</spirit:name>
    <spirit:field>
        <spirit:name>PA_AvailTxDataLanes</spirit:name>
        <spirit:testable>false</spirit:testable>
    </spirit:field>
	class hsi_unipro16_pa_std_region_pa_availtxdatalanes_c extends uvm_reg;
	        uvm_resource_db#(bit)::set({"REG::",get_full_name()}, "NO_REG_HW_RESET_TESTS", 1, this);
	        uvm_resource_db#(bit)::set({"REG::",get_full_name()}, "NO_REG_ACCESS_TESTS", 1, this);

<spirit:register>
    <spirit:name>COMP_RESET</spirit:name>
    <spirit:field>
        <spirit:name>comp_reset</spirit:name>
        <spirit:testable spirit:**testConstraint="readOnly">**true</spirit:testable>
    </spirit:field>
	class hsi_unipro16_component_region_comp_reset_c extends uvm_reg;
	        uvm_resource_db#(bit)::set({"REG::",get_full_name()}, "NO_REG_ACCESS_TESTS", 1, this);

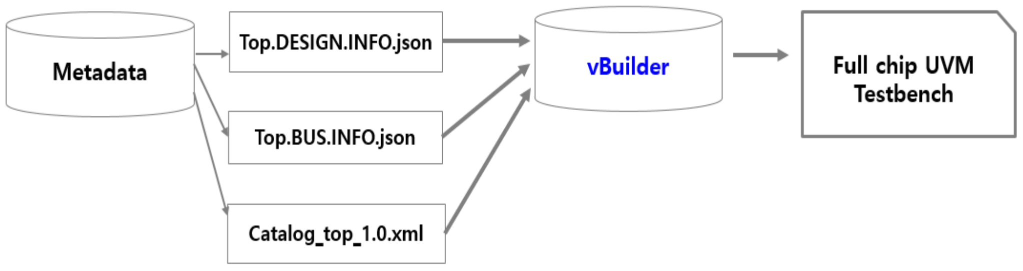
Table 1. Metadata for Register Sanity

III. Case Study

A. *Full chip Register Verification*

✓ DUT Info
- Image Sensor (50Mp)

✓ Inputs
- Top.DESIGN.INFO.json: AMBA bus interface details for all masters and slaves
- Top.BUS.INFO.json: Back-bone bus details
- Catalog_top_1.0.xml: All IP-XACT information for IP/TOP (Registers, Fileset, Clock, Reset)

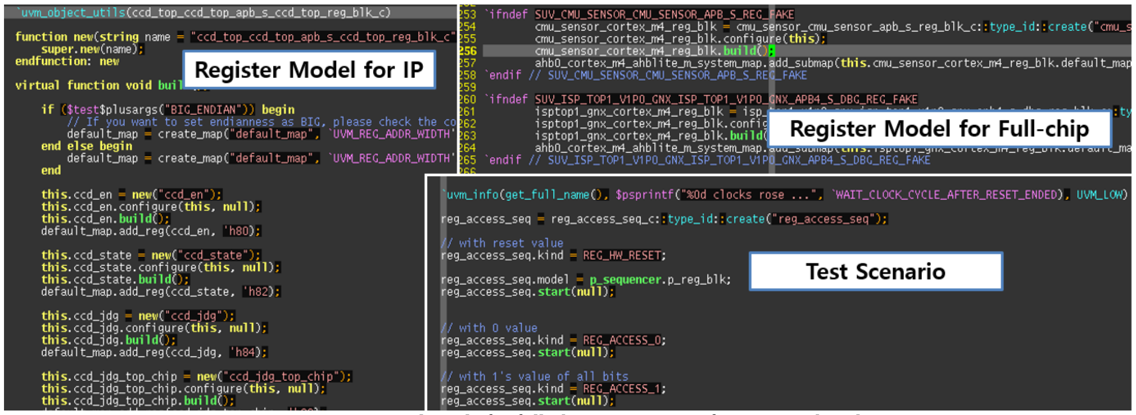


```
% vbuilder -ipjson top.DESIGN.INFO.json -sbjson Top.BUS.INFO.json  ₩
           -xmlcat Catalog_top_1.0.xml -filelist vcode.f ₩
           -corename cortex_m4_AHBLite_M_System
```

Figure 7. Full-chip Register Testbench Generation Flow

✓ Output: UVM Testbench
- Add AHB master VIP to Cortex M4 System Bus
- Generate uvm_reg model for all IPs
- Generate test scenario on top of boot sequence which user provides
- Build and run script

✓ Generated testbench code:



```
`uvm_object_utils(ccd_top_ccd_top_apb_s_ccd_top_reg_blk_c)

function new(string name = "ccd_top_ccd_top_apb_s_ccd_top_reg_blk_c"
    super.new(name);
endfunction: new

virtual function void buil

    if ($test$plusargs("BIG_ENDIAN")) begin
        // If you want to set endianness as BIG, please check the co
        default_map = create_map("default_map", `UVM_REG_ADDR_WIDTH'
    end else begin
        default_map = create_map("default_map", `UVM_REG_ADDR_WIDTH'
    end

    this.ccd_en = new("ccd_en");
    this.ccd_en.configure(this, null);
    this.ccd_en.build();
    default_map.add_reg(ccd_en, 'h80);

    this.ccd_state = new("ccd_state");
    this.ccd_state.configure(this, null);
    this.ccd_state.build();
    default_map.add_reg(ccd_state, 'h82);

    this.ccd_jdg = new("ccd_jdg");
    this.ccd_jdg.configure(this, null);
    this.ccd_jdg.build();
    default_map.add_reg(ccd_jdg, 'h84);

    this.ccd_jdg_top_chip = new("ccd_jdg_top_chip");
    this.ccd_jdg_top_chip.configure(this, null);
    this.ccd_jdg_top_chip.build();
```

```
`ifndef SUV_CMU_SENSOR_CMU_SENSOR_APB_S_REG_FAKE
        cmu_sensor_cortex_m4_reg_blk = cmu_sensor_cmu_sensor_apb_s_reg_blk_c::type_id::create("cmu_s
        cmu_sensor_cortex_m4_reg_blk.configure(this);
        cmu_sensor_cortex_m4_reg_blk.build();
        ahb0_cortex_m4_ahblite_m_system_map.add_submap(this.cmu_sensor_cortex_m4_reg_blk.default_map
`endif // SUV_CMU_SENSOR_CMU_SENSOR_APB_S_REG_FAKE

`ifndef SUV_ISP_TOP1_V1P0_GNX_ISP_TOP1_V1P0_GNX_APB4_S_DBG_REG_FAKE
        isptop1_gnx_cortex_m4_reg_blk = isp_
        isptop1_gnx_cortex_m4_reg_blk.config
        isptop1_gnx_cortex_m4_reg_blk.build(
        ahb0_cortex_m4_ahblite_m_system_map.
`endif // SUV_ISP_TOP1_V1P0_GNX_ISP_TOP1_V1P0_GNX_APB4_S_DBG_REG_FAKE
```

```
`uvm_info(get_full_name(), $psprintf("%0d clocks rose ...", `WAIT_CLOCK_CYCLE_AFTER_RESET_ENDED), UVM_LOW)

reg_access_seq = reg_access_seq_c::type_id::create("reg_access_seq");

// with reset value
reg_access_seq.kind = REG_HW_RESET;

reg_access_seq.model = p_sequencer.p_reg_blk;
reg_access_seq.start(null);


// with 0 value
reg_access_seq.kind = REG_ACCESS_0;
reg_access_seq.start(null);

// with 1's value of all bits
reg_access_seq.kind = REG_ACCESS_1;
reg_access_seq.start(null);
```

Figure 9. Example code for full-chip Register Verification Testbench

✓ Result:

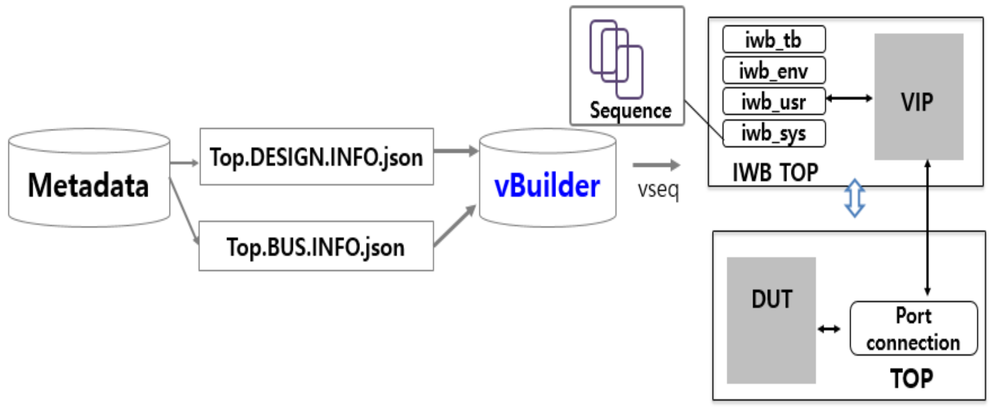
Register testbench generation is fully automated using metadata and it successfully runs as one-shot and found a few IP-XACT/RTL issues quickly. Table 2 shows total runtime for register verification in this design.

Step	Time	Iteration	Total Time
1. Testebench Generation Time	00:02:49	1	**00:38:26**
2. RTL compile	00:00:30	1	
3. Compile (multi-snapshot)	00:00:30	49 (IP)	
4. Run Tests	00:00:13	49 (IP)	

Table 2. Full-chip Register Testbench Runtime

B. *Full chip Interconnect Verification*

✓ DUT Info
- Image Sensor (50Mp)
- Bus Master: 5 with AHB interfaces
- Bus Slave: 12 AHB slaves, 105 APB slaves



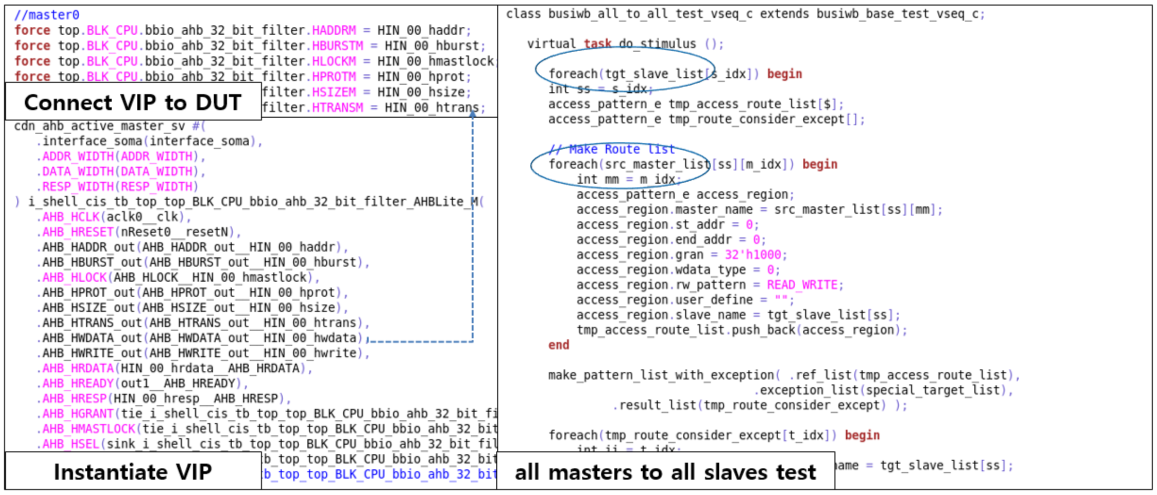
```
% vbuilder -ipjson top.DESIGN.INFO.json -sbjson vbuilder_input.json -bus
```

Figure 8. Full-chip Interconnect Testbench Generation Flow

✓ Output: UVM Testbench
- Add AHB master VIP to Cortex M4 System Bus
- Generate uvm_reg model for all IPs

- Generate test scenarios on top of boot sequence which user provides
  - base_test, one_to_all_test , all_to_one_test, all_to_all_test, unmapped_test
- Build and run script

✓ Generated testbench code:



```
//master0
force top.BLK_CPU.bbio_ahb_32_bit_filter.HADDRM = HIN_00_haddr;
force top.BLK_CPU.bbio_ahb_32_bit_filter.HBURSTM = HIN_00_hburst;
force top.BLK_CPU.bbio_ahb_32_bit_filter.HLOCKM = HIN_00_hmastlock;
force top.BLK_CPU.bbio_ahb_32_bit_filter.HPROTM = HIN_00_hprot;
filter.HSIZEM = HIN_00_hsize;
filter.HTRANSM = HIN_00_htrans;
cdn_ahb_active_master_sv #(
   .interface_soma(interface_soma),
   .ADDR_WIDTH(ADDR_WIDTH),
   .DATA_WIDTH(DATA_WIDTH),
   .RESP_WIDTH(RESP_WIDTH)
) i_shell_cis_tb_top_top_BLK_CPU_bbio_ahb_32_bit_filter_AHBLite_M(
   .AHB_HCLK(aclk0__clk),
   .AHB_HRESET(nReset0__resetN),
   .AHB_HADDR_out(AHB_HADDR_out__HIN_00_haddr),
   .AHB_HBURST_out(AHB_HBURST_out__HIN_00_hburst),
   .AHB_HLOCK(AHB_HLOCK__HIN_00_hmastlock),
   .AHB_HPROT_out(AHB_HPROT_out__HIN_00_hprot),
   .AHB_HSIZE_out(AHB_HSIZE_out__HIN_00_hsize),
   .AHB_HTRANS_out(AHB_HTRANS_out__HIN_00_htrans),
   .AHB_HWDATA_out(AHB_HWDATA_out__HIN_00_hwdata),
   .AHB_HWRITE_out(AHB_HWRITE_out__HIN_00_hwrite),
   .AHB_HRDATA(HIN_00_hrdata__AHB_HRDATA),
   .AHB_HREADY(out1__AHB_HREADY),
   .AHB_HRESP(HIN_00_hresp__AHB_HRESP),
   .AHB_HGRANT(tie_i_shell_cis_tb_top_top_BLK_CPU_bbio_ahb_32_bit_fi
   .AHB_HMASTLOCK(tie_i_shell_cis_tb_top_top_BLK_CPU_bbio_ahb_32_bit
   .AHB_HSEL(sink_i_shell_cis_tb_top_top_BLK_CPU_bbio_ahb_32_bit_fil
b_top_top_BLK_CPU_bbio_ahb_32_bit
b_top_top_BLK_CPU_bbio_ahb_32_bit
```

```
class busiwb_all_to_all_test_vseq_c extends busiwb_base_test_vseq_c;

   virtual task do_stimulus ();

      foreach(tgt_slave_list[s_idx]) begin
      int ss = s_idx;
      access_pattern_e tmp_access_route_list[$];
      access_pattern_e tmp_route_consider_except[];

      // Make Route list
      foreach(src_master_list[ss][m_idx]) begin
         int mm = m_idx;
         access_pattern_e access_region;
         access_region.master_name = src_master_list[ss][mm];
         access_region.st_addr = 0;
         access_region.end_addr = 0;
         access_region.gran = 32'h1000;
         access_region.wdata_type = 0;
         access_region.rw_pattern = READ_WRITE;
         access_region.user_define = "";
         access_region.slave_name = tgt_slave_list[ss];
         tmp_access_route_list.push_back(access_region);
      end

      make_pattern_list_with_exception( .ref_list(tmp_access_route_list),
                                        .exception_list(special_target_list),
              .result_list(tmp_route_consider_except) );

      foreach(tmp_route_consider_except[t_idx]) begin
         int ii = t_idx;
ame = tgt_slave_list[ss];
```

Figure 10. Example code for full-chip Interconnect Verification Testbench

✓ Result

Table 3 shows total runtime for sanity test scenario of interconnect testbench generated by vBuilder. This sanity tests consisted of one master to one slave access test, one master to all slave access test, all masters to one slave access test.

Step	Time	Iteration	Total Time
1. Testebench Generation Time	00:04:39	1	**02:07:57**
2. Build simulation snapshot	00:03:18	1	
4. Run Sanity Test	00:10:00	12	

Table 3. Full-chip Interconnect Testbench Runtime

IV. RESULT

The setup time for interconnect testbench of SOC design used to take two weeks with previous approach at the flagship mobile AP SOC project because it requires many iterations due to mismatches between RTL and testbench. However, the proposed metadata based testbench generation approach reduces this setup time in a day by removing manual process and human-errors. In addition, since interconnect testbench automation can generate target testbench having different configuration of VIP/RTL agents automatically, the scope of verification is easily extended to performance verification, power estimation. The accuracy for performance and power analysis is increased by including key RTL agents like DRAM.

As IP and full chip sign-off for register access test can be performed easily using vBuilder(metadata), the quality of RTL was improved from early stage of design cycle. Thus, verification engineers can focus on functional verification from the beginning. Prior to vBuilder, it requires many iterations due to the mismatch between RTL and specification documents.

V. CONCLUSION

In this paper, we present new methodology regarding how to automate register and interconnect testbench generation using design spec metadata. It contributes to reduce testbench setup time for SOC design from two weeks to one day as well as removing debug efforts due to the discrepancy between RTL design and testbench. Moreover, we can significantly improve the verification quality of register and interconnect verification with this automated approach and it helps us extend verification coverage, start complex function test quickly.

REFERENCES

[1] IEEE Standard for Universal Verification Methodology Language Reference Manual, " in IEEE Std 1800.2-2017, 26 May 2017
[2] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, " in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), 22 Feb. 2018
[3] IP-XACT, https://en.wikipedia.org/wiki/IP-XACT
[4] Accellera, https://www.accellera.org
[5] Introducing JSON, https://www.json.org