Real-Time Synchronization of C model with UVM Testbench

Kirtan Mehta
kirtan.mehta@onsemi.com
onsemi, CORVALLIS, OR, USA

***Abstract*- In ASIC verification, C-based models are essential for managing complex computations and arithmetic processing, serving as golden reference models. This paper introduces a methodology that leverages DPI-C (Direct Programming Interface for C) to automate the synchronization between C models and UVM (Universal Verification Methodology) testbenches. By integrating DPI-C functions into UVM Register Abstraction Layer (RAL) adapters, we achieve real-time updates and dynamic monitoring of bus traffic, significantly reducing the manual effort typically required by UVM predictors and scoreboards.**

I. INTRODUCTION

In ASIC (Application-Specific Integrated Circuit) verification, the complexity of modern designs necessitates robust and efficient methodologies to ensure accuracy and reliability. C-based models have emerged as indispensable tools in this process, particularly for their ability to handle intricate computations and serve as golden reference models. These models streamline testbench development and facilitate post-silicon validation, making them crucial for verifying the functionality of ASIC designs.

However, integrating these C-based models with UVM (Universal Verification Methodology) testbenches presents significant challenges, particularly in maintaining synchronization between the models and the RTL (Register Transfer Level) designs. Traditional methods, which rely heavily on UVM predictors and scoreboards, often require substantial manual intervention, leading to inefficiencies and potential errors.

This paper introduces a methodology that leverages DPI-C (Direct Programming Interface for C) to automate the synchronization process. By embedding DPI-C functions into UVM Register Abstraction Layer (RAL) adapters, we enable real-time monitoring and dynamic updates of bus traffic. This approach not only reduces the need for manual synchronization but also enhances the accuracy and efficiency of the verification process.

This paper demonstrates the implementation of this methodology through a test case that includes multiple RTL registers and a memory module. The integration of a UVM RAL model with a corresponding C model ensures that registers and memories are always in sync in real time. Additionally, the paper also addresses the challenges of handling burst operations via frontdoor memory transactions, demonstrating the scalability of our approach for both block-level and SoC (System on Chip) verification environments. By automating synchronization through DPI-C, our methodology significantly reduces verification time and minimizes the risk of manual errors. This paper aims to enhance the efficiency and reliability of ASIC verification by automating the synchronization between C-based models and UVM testbenches.

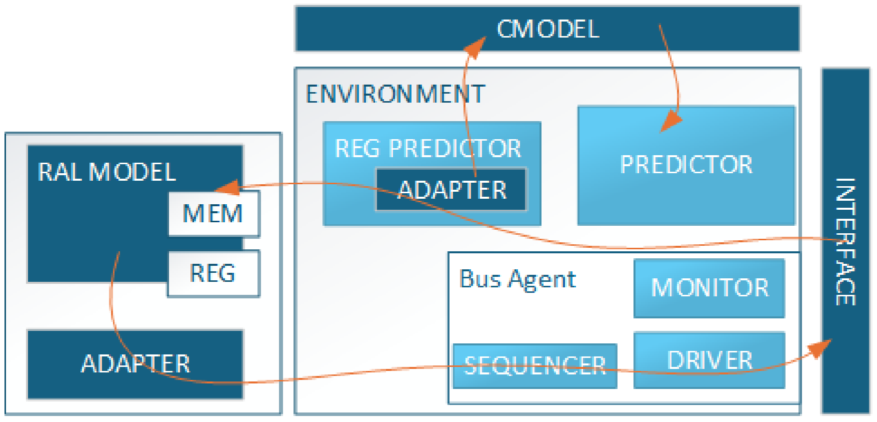

Figure 1: Flow diagram

II. IMPLEMENTATION

A. BUS INTERFACE

In this paper, we will use a simple bus interface to facilitate data transfer between the Testbench and RTL. The interface consists of three main types of signals: data bus, the address bus, and control signals. The 16-bit data bus carries the actual data being transferred, whether it is being read from or written to a memory location. The address bus specifies the target memory or I/O location, directing where the data should go or come from. Control signals, including the clock (clk) for synchronization, an active-high reset (reset) for initialization, and the Read/Not Write (RNW) signal to indicate the type of operation and manage the bus operations.

The bus interface employs the RNW signal to determine the operation mode: when RNW is low, it triggers a write operation, allowing data to be driven into the design. Both the address and data are sent simultaneously on the same positive clock edge, ensuring synchronized communication. For read operations, the data from the RTL is read on the subsequent clock cycle, providing a predictable and orderly data retrieval process.

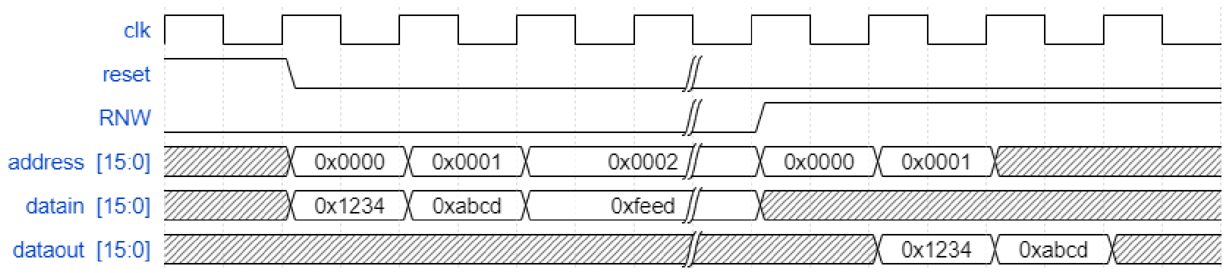

Figure 2: Timing Diagram

B. REGISTERS AND MEMORY.

The table provides a detailed configuration of three registers and a memory block, including field sizes, access types, reset values, field offsets, and address offset that is used in the implementation. This configuration is consistently applied across the C model, Register Abstraction Layer (RAL), and Register Transfer Level (RTL) implementations.

TABLE I.
REGISTER AND MEMORY CONFIGURATION TABLE

Registers/Memory	Field	Size (bits)	Access Type	Reset Value	Field offset	Address Offset
reg1	field2	8	RW	0x10	8	0x0
	field1	8	RW	0x20	0	0x0
reg2	field2	8	WO	0x30	8	0x2
	field1	8	WO	0x40	0	0x2
reg3	field2	8	RO	0x50	8	0x4
	field1	8	RO	0x60	0	0x4
memory		16x1024	RW	0x00	-	0x6

C. C MODEL

The Cmodel header in Figure 3, mirrors the register and memory structure of the UVM Register Abstraction Layer (RAL) model. The reg_field_t structure is introduced to represent individual register fields, detailing attributes like value, reset value, size, offset, and access type (e.g., read/write, read-only, write-only). This setup allows for precise control and documentation of each field within a register.

Building on this, the reg_t structure aggregates multiple fields into a single register, specifying the register's size, address, and map offset. The reg_block_s structure further organizes these registers (reg1, reg2, reg3) into a cohesive block alongside a memory array of 1024 location of type hwint, enabling efficient management of related hardware components.

This modularity and clarity simplify code maintenance, reduce errors, and streamline debugging. Consistency across the codebase ensures accurate hardware representation in verification models, aligning the C model with the UVM RAL model.

```
typedef uint16_t hwint;                  // 16-bit unsigned int

typedef struct {
hwint      value;                        // Field value
hwint      reset_value;                  // Reset value
uint8_t    size;                         // Size in bits
uint8_t    field_offset;                 // Field offset
const char* access;                      // "RW", "RO", "WO"
} reg_field_t;

typedef struct {
   reg_field_t fields[2];                // Fields array
   uint8_t    size;                      // Size in bits
   hwint      address;                   // Address
   uint8_t    map_offset;                // Map offset
} reg_t;

typedef struct {
   reg_t reg1;                           // Register 1
   reg_t reg2;                           // Register 2
   reg_t reg3;                           // Register 3
   hwint memory[1024];                   // Memory Array
} reg_block_s;

extern reg_block_s reg_block            // Register block

#define reg1_ADDRESS          0          // reg1 addr
#define reg2_ADDRESS          2          // reg2 addr
#define reg3_ADDRESS          4          // reg3 addr
#define memory_ADDRESS        6          // Memory addr
#define memory_LOCATIONS 1024            // Memory locations
```

Figure 3: C MODEL HEADER STRUCTURE

D. CMODEL HEADER AND REGMODEL COMPARISON

The UVM code snippet in Figure 4, two fields, field2 and field1, are defined for a register. These fields are instances of the uvm_reg_field class, which is used to model individual fields within a register. The build function is responsible for creating and configuring these fields. Each field is created using the create method and then configured with specific properties such as width, starting bit position, access type (read-only in this figure), and reset behavior.

In the C code snippet in Figure 5, a similar implementation is done for a register block, which can be compared to the Register Abstraction Layer (RAL) model in UVM. The build_C function defines and initializes a register (reg3) with its fields. Each field within the register is initialized with properties such as value, reset_value, size, field_offset, and access. The register is configured with its size, address, and map offset. Additionally, the memory array within the register block is initialized to zero. This setup ensures that the register and its fields are correctly initialized and accessible in the hardware design.

```
rand uvm_reg_field field2;              /*field 2 */
rand uvm_reg_field field1;              /*field 1 */

// Function : build
   virtual function void build();
      this.field2 = uvm_reg_field::type_id::create("field2");
      this.field2.configure(.parent(this), .size(8), .lsb_pos(8), .access("RO"), .volatile(0), .reset(8'd60),
            .has_reset(1), .is_rand(1), .individually_accessible(0));
      this.field1 = uvm_reg_field::type_id::create("field1");
      this.field1.configure(.parent(this), .size(8), .lsb_pos(0), .access("RO"), .volatile(0), .reset(8'd50),
            .has_reset(1), .is_rand(1), .individually_accessible(0));
   endfunction
```

Figure 4: UVM REGMODEL BUILD

```
void build_C (hwint offset) {
   reg_block.reg3 = (reg_t){
      .fields = {
         {.value = 0, .reset_value = 0x50, .size = 8, .field_offset = 0, .access = "RO"},
         {.value = 0, .reset_value = 0x60, .size = 8, .field_offset = 8, .access = "RO"}
      },
      .size = 16,
      .address = reg3_ADDRESS + offset,
      .map_offset = 4
   };
   for (int i = 0; i < memory_LOCATIONS; i++) {
      reg_block.memory[i] = 0;
   }
}
```

Figure 5: C MODEL HEADER BUILD

Both the UVM and C code define and configures fields within a register, ensuring that each field has specific properties and behaviors. The UVM code is used for verification purposes, allowing fields to be randomized and tested, while the C code provides a hardware abstraction layer, defining how the register and its fields are initialized and accessed in a hardware design. These implementations work together to ensure that the register fields are correctly modeled in both the verification environment (UVM) and C based models.

E. UVM ENVIRONMENT

The code snippet in Figure 6, shows how the UVM environment builds the C model alongside the Register Abstraction Layer (RAL) using the DPI-C function. The build_C function is declared to take an offset as an input. Within the build_phase function, this DPI-C function is called with the specified offset to construct the C model. This call ensures that the C model is built and integrated with the RAL during the build phase of the UVM environment, facilitating synchronized operation between the testbench and the C model.

```
   import "DPI-C" function void build_C(input int offset);
   virtual function void build_phase (uvm_phase phase);
         …
         e_regworks_regs.build();                        // Building RAL model for UVM Testbench
         e_regworks_regs.regworks_map.set_base_addr (reg_offset);  //Set the offset for Register Model
         build_C( offset );                        // Building Cmodel along with RAL in environment.
         …
   endfunction
```

Figure 6: UVM ENVIRONMENT

F. RAL ADAPTERS

The provided code snippet in Figure 7, demonstrates how the UVM Register Adapter writes to the C model using the DPI-C function. The cmodel_write function is declared to take an address and data as inputs. Within the bus2reg function, the type of operation (read or write) is determined by the RNW signal. For write operations (RNW == 0), the data from trans_h.datain is written to the C model using cmodel_write(trans_h.addr, trans_h.datain). The address for the operation is set from trans_h.addr, ensuring that the correct location in the C model is updated. This integration allows for direct interaction between the UVM testbench and the C model, ensuring that any write operations on the bus are immediately reflected in the C model.

```
virtual function void bus2reg (uvm_sequence_item bus_item,
                               ref uvm_reg_bus_op rw);
…
  rw.kind = (trans_h.RNW == 1) ? UVM_READ : UVM_WRITE;
  rw.addr = trans_h.addr;
  if (trans_h.RNW == 0) begin
    rw.data = trans_h.datain;
    cmodel_write(trans_h.addr, trans_h.datain);          //   DPI write function to C Model
  end else begin
    rw.data = trans_h.dataout;
  end
  …
endfunction: bus2reg
```

Figure 7: RAL BUS2REG ADAPTER

G. C MODEL WRITE FUNCTION

The cmodel_write function is responsible for writing data to specific hardware addresses in a simulation environment. When the address matches a particular register (reg_block.reg3.address), and the register is writable, the function updates the register fields with the provided data. The lower 8 bits of the data are stored in the first field, while the next 8 bits are stored in the second field. This ensures accurate simulation of register operations.

For memory operations, the function checks if the address falls within a defined memory range. If it does, the function calculates the appropriate memory index and writes the data to the corresponding memory location in the reg_block.memory array. This helps simulate interactions between the processor and peripheral devices, ensuring data is correctly routed and stored.

```
void cmodel_write(hwint addr, hwint data) {
  if (addr == reg_block.reg3.address) {
    if (strcmp(reg_block.reg3.fields[0].access, "RO") != 0) {
      reg_block.reg3.fields[0].value = data & 0xFF;
      reg_block.reg3.fields[1].value = (data >> 8) & 0xFF;
    }
...
  if (addr >= memory_ADDRESS && addr < memory_ADDRESS + memory_LOCATIONS * 2) {
    reg_block.memory[(addr - memory_ADDRESS) / 2] = data;
    printf("Written to memory[%d]: %x\n", (addr - memory_ADDRESS) / 2, data);
  }
…
}
```

Figure 8: C MODEL WRITE FUNCTION

III. ADVANTAGES

Implementing the proposed methodology for real-time synchronization of C models with UVM testbenches using DPI-C offers several significant advantages. Firstly, automating synchronization by embedding DPI-C functions into UVM Register Abstraction Layer (RAL) adapters reduces the manual effort typically required by UVM predictors and scoreboards. This streamlines the verification process and minimizes potential human error. Additionally, the methodology enables continuous updates and dynamic monitoring of bus traffic, ensuring that the C model and the UVM testbench are always in sync. This real-time synchronization enhances the accuracy and efficiency of the verification process, leading to more reliable and consistent results.

The methodology's scalability allows it to handle complex operations such as burst operations via frontdoor memory transactions, making it suitable for both block-level and SoC (System on Chip) verification environments. Consistency across models is also enhanced, as the C model mirrors the register and memory structure of the UVM RAL model, ensuring uniformity across the codebase. Overall, this methodology significantly enhances the efficiency, accuracy, and reliability of ASIC verification, contributing to more robust and timely project outcomes.

IV. LIMITATION

The successful integration of DPI-C functions into UVM RAL adapters heavily relies on having a common definition of registers and memories in both the cmodel header and RAL model. Ensuring this commonality is crucial for smooth integration and operation. Despite its importance, this process can be quite complex due to the intricate interface protocols involved and the necessity for expertise in both C programming and UVM. Maintaining synchronization as the design evolves can also be demanding, requiring continuous effort.

Performance overhead is another significant concern. Real-time updates and dynamic monitoring of bus traffic can slow down the simulation and require substantial computational resources, ultimately impacting overall system performance. Scalability issues arise with larger designs, where efficiently managing extensive register and memory configurations can become challenging. Managing burst operations via frontdoor memory transactions can grow increasingly complex as the design size expands.

REFERENCES

[1] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), vol., no., pp. 1-1315, Feb. 22, 2018, doi: 10.1109/IEEESTD.2018.8299595.

[2] J. Aynsley, "SystemVerilog Meets C++: Re-use of Existing C/C++ Models Just Got Easier," Doulos, 2010. [Online]

[3] Joshua Hardy, "Modeling a Hierarchical Register Scheme with UVM," in *DVCon Proceedings*, [2017]. Available: https://dvconproceedings.org/wp-content/uploads/modeling-a-hierarchical-register-scheme-with-uvm.pdf

[4] M. Litterick and M. Harnisch, "Advanced UVM Register Modeling," in *DVCon Proceedings*, Available: https://dvconproceedings.org/wp-content/uploads/advanced-uvm-register-modeling.pdf

[5] B. Oden, "UVM Framework (UVMF)," Siemens Verification Academy, 2023. [Online]. Available: https://verificationacademy.com/topics/uvm-universal-verification-methodology/uvmf