Bringing Constrained Random into SoC SW-driven Verification

Alberto Allara, alberto.allara@st.com
Fabio Brognara, fabio.brognara@st.com
STMicroelectronics - Italy
Overview

• SW-driven verification: benefits and drawbacks

• Verification Abstraction Layer (VAL)

• Randomizations in SW-driven SoC tests

• Example of usage model

• Conclusions
SW-driven verification technique

• Represents one of the most used techniques for System-On-Chip verification

• Advantages:
  – Easiness of use
  – Reusability across levels
    • Block level in simulation
    • SoC level in simulation
    • SoC level in co-emulation
    • In the actual Silicon

• Drawbacks:
  – Traditionally based on Direct tests
Randomization in SoC verification

• The EDA market is investing on *Model-based test generation tools*
  – The user models a system in terms of graphs (or other models) and such tools randomize and pre-generate C/C++ tests

• We propose a lightweight technique based on SystemVerilog UVM methodology
  – enabled by a Verification Abstraction Layer (VAL)
Verification Abstraction Layer (VAL)

- VAL is an interface that exposes the components of a SoC verification environment to an embedded CPU as Status and Control registers.
- VAL enables verification engineers to:
  - Configure the verification components directly from software.
  - Generate expected results to external scoreboard for self-checking capabilities.
  - Enable randomization in SW-driven verification.
VAL Structure

VAL

VAL-FE

Bridge

VAL-BE1

VAL-BE2

VAL-BEN

6 of (15)
How does the VAL work

Each VAL-BE is associated to a virtual Register file mapped on the Embedded memory.
How does the VAL work

Each memory transaction generated on a memory connected to the VAL is captured and propagated to the VAL-BE.

The data are moved from memory to VAL-BE and vice versa through backdoor memory accesses.

Each VAL-BE is sensitive to a different VAL_TRIG memory location.
The VAL Back-end component

- Each VAL-BE is the specialization of a UVM object class called `val_component`
- Includes the following knobs:
  - Memory address of the VAL_TRIG the component is sensitive to
  - Base memory address where the VAL-BE register map is located
- Supports the following APIs
  - `read_mem_word(bit[31:0] addr, output bit[31:0] data));
  - `read_mem_dword(bit[31:0] addr, output bit[63:0] data));
  - `write_mem_word(bit[31:0] addr, input bit[31:0] data));
  - `write_mem_dword(bit[31:0] addr, input bit[63:0] data));
- Defines the `val_op()` callback
- Defines the connection between memories and backdoor API functions
VAL connectivity

VAL-FE → Bridge → VAL-BE1 → Scoreboard → Seq → D

VAL-BE2 → cfg → M

VAL-BE3 → Analysis FIFO → Seq → D

10 of (15)
Randomization with VAL

- UVM object defined within a VAL-BE
- The CPU through VAL triggers the randomization of this object
- The content of the randomized object is exported to the system:
  - serialized back to memory in order to be accessible by SW
  - sent outside VAL-BE through analysis port
VAL randomization: an example

- Randomization of a LCD configuration

```plaintext
class val_lcd extends val_component;
    lcd_cfgPacket cfgp;

    function void val_op();
        case(VAL_TRIG)
            RAND_FRAME: begin
                a = (mem_addr - base_mem_addr )+4;
                read_mem_word(a,w); lcd_resolution=w;
                assert(cfgp.randomize() with { 
                    res == lcd_resolution} )
            end
            else `uvm_fatal("VAL_LCD","Unable to randomize");
        a += 4;
        for(int i=0; i < cfgp.rows; i++)
            for (int j=0; j<cfgp.columns; j++) begin
                write_mem_word(a,cfgp.data[i][j]); a+=4;
            end
```

Packet Randomization

Data serialization and copy into memory
Constraining the Randomization

- Two ways to control randomization: static and dynamic
- Static
  - Based on UVM factory mechanism
  - Replaces the object in the VAL-BE with a specialization of the same object with different constraints
- Dynamic
  - Relies on values passed from SW to control the randomization constraints at run-time
Example of VAL usage model

1) CPU triggers the VAL

2) VAL randomizes a LCD frame and copy into mem

3) The same data is sent to the scoreboard

4) The LCD controller transmits the frame

5) The frame collected is sent to the SB for the comparison
Conclusions and future works

• Proposal of a methodology to bring *Coverage Driven Constrained Random* capabilities within SW-driven Soc tests
  - The key element of the approach is VAL
• Through VAL the embedded SW can trigger the randomization of data for:
  - VIP configuration
  - IP configuration
  - Payload data generation
  - …
• Next step: development of a synthesizable VAL-FE to bring randomization into Co-Emulation environments
BACKUP SLI DES
VAL-BE design: the tube example

- The VAL_TUBE implements the UVM reporting mechanisms as well as the end-of-simulation mechanism

```c
typedef enum _uvm_severity {
    UVM_INFO =0,
    UVM_WARNING,
    UVM_ERROR,
    UVM_QUIT
} uvm_severity;

typedef struct _val_tube_registers {
    uint32_t VAL_TRIG;
    uint32_t STR_BUFF[50];
} val_tube_registers;

void uvm_error(const char* str) {
    uvm_report(str,(uint32_t)UVM_ERROR);
}
void uvm_warning(const char* str) {
    uvm_report(str,(uint32_t)UVM_WARNING);
}
void uvm_info(const char* str) {
    uvm_report(str,(uint32_t)UVM_INFO);
}
void uvm_report(const char* str,uint32_t sev) {
    volatile val_tube_registers* p=(volatile val_tube_registers*)VAL_TUBE_START;
    strcpy((char*)p->STR_BUFF,str);
    *(p->VAL_TRIG) = sev;
}
void global_stop_request(void) {
    volatile val_tube_registers* p=(volatile val_tube_registers*)VAL_TUBE_START;
    *(p->VAL_TRIG) = (uint32_t)UVM_QUIT;
}
```
VAL-BE design: the tube example

- HVL side implementation

```cpp
class val_tube extends val_component;
    string s;

    virtual function void val_op();
        if (VAL_TRIG == 3) begin
            `uvm_info("VAL_TUBE","stop activated from SW",UVM_NONE);
            uvm_test_done.drop_objection(this,"Stop triggered by SW");
        end
        else begin
            s = read_string_from_mem((mem_addr - base_mem_addr ) + 4);
            case(VAL_TRIG)
                0: `uvm_info("VAL_TUBE",s,UVM_NONE);
                1: `uvm_warning("VAL_TUBE",s);
                2: `uvm_error("VAL_TUBE",s);
            endcase
        end
    endfunction : val_op

    virtual function string read_string_from_mem(input bit[31:0] a);
        ...
    endfunction : read_string_from_mem
endclass : val_tube
```