Reuse UVM RTL verification tests for gate level simulation

David Vincenzoni - October 01, 2015

When we verify a System on Chip (SoC) that embeds microprocessors with several digital peripherals, and possibly analog blocks as well, we want to check all the implemented features and possible corner cases to minimize verification time. A mix of several techniques and methodologies are used to improve the functional verification and extract a measure of the grade of coverage: Formal verification and random constrained tests based on Universal Verification Methodology (UVM) increase the probability to discover bugs. Sometimes we create a perfect and effective test for RTL verification only to find out it can’t be reused during the gate level simulation because the UVM Monitors are hooked on internal SoC signals that can disappear or change after the implementation phase.

This article will describe how easy it is to create efficient self-checking tests that are straightforward, and reusable during gate level simulations. It is surprising that, by changing the data flow, we can have benefits for the test bench, reducing the complexity of scoreboards, which also means less time for test developing.

The flow is based on the instantiation of UVM Verification Components used for checking interfaces such as SPI, I²C, & UART, but it can also be extended for more complex interfaces.

SoC Verification Flow

The most effective SoC verification is based on the instantiation of several UVM Verification Components (UVM VC) placed inside the internal buses, on specific internal modules, and on the primary SoC interfaces. These UVM VC are used as bus protocol checkers (e.g., AMBA checker); serial protocol checkers, and active masters (e.g., FC, SPI, UART, JTAG, SATA, PCIe).

The tests developed based on UVM should be self-checking; every action, stimulus and transaction, must be verified by a checker, which in case of a mismatch raises a “flag” that stops the simulation and issues an error message that is shown on simulator console and written into a log file.

The verification of communication interfaces (e.g., SPI) needs the use of UVM VC formed by a Collector that gets its transaction on bus, a Monitor that checks the compliancy of protocol, and a Bus Functional Model (BFM) that generates the transactions. The data exchanged between the SoC and external UVM VC are verified through a module named scoreboard.

This scoreboard has at least two ports where the added object is matched with a second one – the reference. In case of a mismatch, an error is issued. This kind of checker must be reused during the
gate level simulation in order to stimulate the critical paths. Figure 1 shows a simple block diagram of a verification test bench that uses several checkers for an effective verification methodology.

Figure 1: Typical UVM Verification Test Bench

The yellow blocks are UVM VC. The Bus Monitor is a passive component that has only the monitor and checker.

A common data flow, for testing a serial peripheral, is to send data from the UVM VC towards the microprocessor and check that the data have arrived at the destination (microprocessor). In the second step, we send data from the microprocessor to the UVM VC checking that the correct data have arrived to the destination (UVM VC).

Figure 2 shows an example of a data scoreboard for a full duplex synchronous communication (e.g., SPI). The monitors (passive) on the UVM VC and on the peripheral bus are used for sending the data to the scoreboard: The data sent by UVM VC are added to the scoreboard TX path, and when they reach the peripheral, are sent to the same scoreboard for matching by means of the Bus Monitor. The data from peripherals are added to the scoreboard RX path and matched with the data received by the UVM VC passive monitor.
The main drawback of this approach is the portability at gate level simulation. During the implementation phase, internal signals that are available in RTL can disappear in the optimized Netlist, and the binding of UVM VC modules becomes difficult and sometimes impossible (e.g. unused ports that are removed during the synthesis).

Let’s take a look at something new...

**A New SoC Verification Flow**

The basic concept of the new flow is that the data checked in the *scoreboards* should not be coming from monitors bound on internal SoC nodes (see Figure 2). Consequently, the test bench configuration is modified in order to use only the UVM VC (active and passive) for data checking bound at the top level. The internal monitors (the ones bound on the internal node of SoC) are used only in RTL simulation, checking the compliancy of the buses and tracing the coverage.

At this point, it is necessary to change the data flow for the packet exchanged between UVM VC and SoC peripherals. The main requests are:

- Data randomization
- Portability at Gate Level Simulation
- Ease of use

The random constrained data packets generated by the UVM VC and received by the peripheral are used by the SoC for generating the outbound packets. In order to increase the randomization, the
microprocessor in the SoC computes the CRC on received data and uses the results as data to be transmitted.

The *scoreboard* will get the data from UVM VC, and before adding them in the data list, it will compute the CRC. The data coming back from the peripheral are added directly to the *scoreboard* for the matching process. Figure 3 shows an example of this new flow.

![Figure 3: New approach of data checking](image)

The computation of CRC is really straightforward; for the *scoreboard* the built-in function (pseudo-method) can be used, which is available, for example, in *e* language:

- `list_bytes.crc_32(from byte, number of bytes)`
- `list_bytes.crc_8(from byte, number of bytes)`

It reads the list byte-by-byte and returns the integer value of the CRC function of a list of bits or bytes. The CRC can be computed defining the starting byte (through the first parameter *from byte*) and on selected number of bytes (through the second parameter *number of bytes*).

The generator polynomial for the 32-bit CRC is:

\[ x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1 \]

The generator polynomial for the 8-bit CRC is:

\[ x^8 + x^2 + x + 1 \]
The analogous function is used in the C code executed by the microprocessor in the SoC:

```c
int ComputeCrc8(int NewByte, int CrcReg)
{
    int i;
    int tmpBit, syndrome, newSyndrome;

    syndrome = CrcReg;
    for (i = 0; i < 8; i++)
    {
        tmpBit = ((syndrome >> 7) & 0x1) ^ ((NewByte >> (7-i)) & 0x1);
        switch (tmpBit)
        {
            case 1: newSyndrome = (((syndrome & 0x0F) ^ POLY_GEN8) << 1) + 0x1;
                   break;
            default: newSyndrome = syndrome << 1;
                    break;
        }
    }

    syndrome = newSyndrome;
}

Where the POLY_GEN8 is 0x03. The function can be extended for CRC32 using the POLY_GEN32 = 0x2608EDB.

The use of this methodology greatly simplifies data management on the scoreboard; since the data checking is performed on a serial interface, it is independent of the data size of the microprocessor bus, which can be 8, 16, 32 bits or more. Below is an example of a scoreboard.

```c
// added frame - match is found in trans match port
spi_frame_add predict(item : st_spi_transaction) is only {
    var data_spi_crc: st_spi_transaction;
    var data_list : list of bit;
    var data_crc : byte;

    data_spi_crc = item.copy();

    if (item.data_tx != 0) {
       // Compute the CRC before to add in the scoreboard
       data_list = pack(NULL, item.data_tx);
       data_crc = data_list.crc_8(0, data_size(data_len));
       item.data_crc = pack(NULL, data_crc, ~data_crc);

       add_to_scobd(item);
       set_match_port(item, spi_frame_match);
    }
}

spi_frame_match_reconstruct(item : st_spi_transaction) is only {
    item.data_crc = item.data_tx;

    if (item.data_tx != 0) {
       match_in_scobd(item);
    }
}
```

Full Duplex Synchronous Interface
In case of a full duplex synchronous interface such as an SPI, we can have two possible modes: the peripheral is slave, or the peripheral is master.

When the peripheral is slave, the transactions are initiated by the external UVM VC. The full duplex modes imply the necessity of sending and receiving data at the same time. Since we want to avoid data generation on the SoC side (no cabled data on C code), there is a systematic delay on valid data. This delay shall be implemented on the scoreboard in order to properly compare the data exchanged.

![Figure 4: Full Duplex data sequence for Slave Mode](image)

Figure 4 shows an example of a data sequence exchanged between the master (UVM VC) and slave (peripheral). The first symbols transmitted by the peripheral are ‘do not care’ for the scoreboard and are thus discharged. The microprocessor takes some time to get the data from the buffer and compute the CRC. After a while, the data are ready and the exchanging continues with the random generation from the UVM VC and pseudo random data from the peripheral side (CRC computed over $D_x$).

We can have an automatic symbol synchronization, by using a conventional ‘do not care’ data (e.g., zero). Of course, this data shall not be generated by the UVM VC. The peripheral on the SoC will send zero until the CRC data are ready, and the UVM VC terminates with enough zero symbols for completing the matching in the scoreboards.

When the peripheral is master, the transactions are initiated by itself through the microprocessor.

![Figure 5: Full Duplex data sequence for Master Mode](image)

Figure 5 shows an example of a data sequence exchanged between the master (peripheral) and slave (UVM VC). The first symbols transmitted by the peripheral are ‘do not care’ for the scoreboard. The UVM VC replies with random symbols $D_x$ that the microprocessor uses to compute the pseudo random data transmitted by the peripheral. As above, the yellow highlighted symbols are the one compared in the scoreboard.

The data flow is quite similar to the case of slave mode. Figure 5 shows an example of a data sequence exchanged between the master (peripheral) and slave (UVM VC). The first symbols transmitted by the peripheral are ‘do not care’ for the scoreboard. The UVM VC replies with random symbols $D_x$ that the microprocessor uses to compute the pseudo random data transmitted by the peripheral. As above, the yellow highlighted symbols are the one compared in the scoreboard.

**Half-Duplex Interface**

In case of half-duplex interface such as I²C, we can have two possible modes: the peripheral is slave, or the peripheral is master.
When the peripheral is slave, the transactions are initiated by the external UVM VC. The half-duplex modes are based on a communication protocol where the master sends commands requesting data as a reply or sends data to the dedicated slave.

As for full duplex mode, it is necessary to define a good flow that avoids the data in C code (not random) and keeps the data checking simple:

- Start the transaction by sending the write command. The UVM VC will start sending data packets. These data are added in the scoreboard after the computation of CRC.
- The received data are used by the DUT as ‘reply of read’ command. The microprocessor computes the CRC on received data and prepares the data packets for reply.
- The UVM VC sends the read command. The peripheral start sending the previously prepared data. These data are added in the scoreboard for matching.

Figure 6 shows a simple diagram of this data handshake.

Master Mode

When the peripheral is master, the transactions are initiated by themselves through the microprocessor.

In this case, in order to exploit the random constrained feature of UVM VC, the protocol shall be

- Start the transaction by sending the read command. The UVM VC will start sending data packets as reply. These data are added in the scoreboard after the computation of CRC.
- The received data are used by the DUT as ‘data for write’ command. The microprocessor computes the CRC on received data and prepares the packets for the next step.
- The peripheral sends the write command, and then it starts sending the previously prepared data. These data are added in the scoreboard for matching.

Figure 7 shows a simple diagram of this data handshake.
Complex Protocol

The same methodology can also be used for complex protocols such as USB or Ethernet. The concept is the same: for full duplex communication, the initial symbols are ‘do not care’ (null symbols), and then the DUT uses the received samples to compute the CRC and send the data back.

For half duplex, the data exchange is initiated by the UVM VC, and then the DUT uses the received packet to build the transmission packets.

Conclusion

In this article, we defined a different approach on data flow when we are verifying SoC, checking the communication interfaces. The main target is, while in the gate level simulation, to directly reuse the tests developed in RTL. We saw the new flow using the CRC for generating the data coming from DUT. We can generalize the concept by saying that the data sent back by the DUT are a function of random constrained data generated by the UVM test bench. The more complex the function is that is used in the DUT, the safer the test is for bug hunting.

This approach has three important advantages: it exploits the random constrained data peculiar to the UVM, it is portable for gate level simulation without changing any UVM or C code, and last but not least, it is easy to use. Moreover, the homogeneity of the objects compared in the scoreboard keeps it really simple and straightforward.