Abstract

The key to System-on-Chip [SoC] design success is verification. Most state-of-the-art verification methodologies use a transaction level Reference Models that is usually untimed. Problems appear when asynchronous transactions have to be executed according to the order they occurred. This paper proposes a solution for timed transactions to treat real time asynchronous applications using a functional verification methodology called VeriSC [1]. In order to demonstrate timed transactions in VeriSC, the DigiSeal project which consists of the digital circuit for monitoring the unauthorized opening of residential energy meters is used a case study.

1. Introduction

Nowadays, the cornerstone in the project of an IP core is functional verification. It consumes about 70% of the overall project resources [2]. Functional verification is a process used to demonstrate by simulation that the intent of a design is preserved in its implementation. It is currently often used in the verification of integrated circuit designs, along with other methods such as formal verification, semi-formal verification, hardware emulation, prototyping, etc. Functional verification uses a testbench to create an environment for simulating the DUV (Design Under Verification). The testbench simulates all functionalities of the DUV, comparing it to the specification. Thus functional verification consists of three processes: testbench implementation, simulation and debugging.

Traditional methodologies propose the implementation of the DUV before the implementation of the testbench [3,4,5]. VeriSC functional verification methodology permits the creation of the simulation environment (testbench) before or in parallel with the Design Under Verification (DUV). The testbench implementation of this methodology is transaction-level, coverage-driven, random-constrained and self-checking. The testbench uses FIFOs at transaction level, but communicates with the DUV at signal level. In order to make this communication possible, the testbench uses TDrivers and a TMonitors to translate data from the transaction level to signal level and back, respectively. The testbench is also composed of a Source in order to input data to the simulation, a Reference Model and a Checker in order to automatically compare the results coming from the Reference Model and the DUV.

VeriSC allows for testbench construction and debugging before RTL code of the DUV is available. Traditional methodologies build the testbench around the DUV. They may be adapted to provide a testbench before RTL design, however only at the expense of writing extra code for the DUV that takes the place of the future RTL implementation. VeriSC does not need any extra code writing because TDrivers, TMonitors and the Reference Model are reused to take the place of the DUV before RTL code is available.
An advantage of this approach is that one can begin the verification process before
the RTL design and better consistency of sub-block interfaces is achieved during
hierarchical refinement by the TDriver/TMonitor approach and code reuse.

Typically Reference Models are untimed. This is a problem when asynchronous
transactions have to be executed according to the order they occurred. This paper proposes
the extension of VeriSC with timed transactions to treat real time asynchronous applications.

In order to demonstrate timed transactions in VeriSC, this paper presents as a case
study the project of a digital device used for monitoring the unauthorized opening of
residential energy meters (DigiSeal).

The remainder of this paper is organized as follows. Section 2 describes the VeriSC
methodology. Section 3 presents the DigiSeal project. Section 4 shows the results and
Section 5 concludes the work.

2. VeriSC Methodology

Traditional verification methodologies have some downsides, such as: considerable
project time is spent in verification; the testbench code often can not be reused; big amounts
of work needed to adapt a testbench to each element of the project hierarchy, as
hierarchical decomposition often does not consider functional verification; the verification
team needs to wait until RTL implementation has finished; and the testbenches are
debugged together with the RTL implementation, your Reference Models are untimed, and a
simulation error may be caused by a testbench bug as well as by an error in the DUV.

Most of these problems come from the fact that traditional methodologies encourage
implementing first the DUV, and then the testbench [2, 4, 6].

Using these verification approaches, the testbenches usually get more complex when
compared to when testbenches are created beforehand.

VeriSC proposes a better integration between RTL implementation and verification.
Through this methodology, testbenches are implemented before or along with RTL
implementation. Some advantages of this approach are: the functional verification can be
done incrementally throughout each iteration of hierarchical refinement, and errors can be
corrected before RTL implementation; the verification process is usually more efficient when
the hierarchical subdivision is done from a verification perspective.

As most project effort is concentrated in the verification process, the overall design
effort decreases significantly; the Interface drivers and monitors may be reused in a way to
assure interface consistency throughout the hierarchical decomposition process; the
testbench and DUV design flaws can be detected and debugged earlier so overall
debugging time is reduced; A problem exists when asynchronous transactions have to be
executed according to the order they occurred. VeriSC allows timed transactions to treat real
time asynchronous applications.

To the best of our knowledge, there is no other methodology capable to provide all
these advantages.

In order to use the proposed methodology it is necessary to have a Reference Model
(RM), which is a golden model that implements all specified functionalities of the DUV.

The general testbench scheme, used in VeriSC is shown in Figure 1.

![General testbench scheme](image-url)
The testbench is composed of the following elements: Source, TDriver(s), TMonitor(s), Reference Model (RM) and Checker. The testbench functionality is to input data into the DUV and the RM, capturing the outputs and comparing automatically if both of them are equivalent. The synchronization mechanism of the testbench is implemented by means of First-In-First-Out queues (FIFOs) through which transactions are sent. The testbench structure shown is intended to be used for synchronous designs with a single clock signal, and asynchronous designs.

The functionality of each block of the testbench is briefly described in the following [6].

**Source**: It is responsible for providing transaction level (TL) data to the DUV and to the RM. It is connected to the RM and to the TDriver(s) by means of FIFO(s). There is one FIFO for each input interface of the DUV. The same number of FIFOs is going to the RM and to the TDrivers.

**TDriver**: It receives TL data from the Source, transforms it in specified protocol signals and passes these signals along with the required data to the DUV. A TDriver functions as a bridge between TL and signal level. There is always one TDriver for each input interface to the DUV.

**TMonitor**: It is a bridge between signal data and transactions. Thus, it is responsible for receiving the protocol and data signals from the DUV and transforming them into TL data. The TMonitor puts the TL data into a FIFO and passes them to the Checker.

**Checker**: It is responsible for comparing TL data coming from the RM with TL data coming from TMonitor(s) to see if they are equivalent. The checker will automatically compare the outputs from RM and DUV and prints error messages if they are not equivalent.

**Reference Model (RM)**: It receives TL data from the Source through FIFO(s) and sends TL data to the Checker through FIFO(s). If not provided by a third part, the verification engineer must implement the functionality of the RM.

In order to implement a testbench, the implementation is done in small incremental steps, each step resulting in a working simulation that can be easily debugged. Some of these steps consist in reusing elements from the testbench in order to replace the functionality of the DUV.

### 3. DigiSeal Case Study

The DigiSeal project consists of the digital circuit for the detection of tampering with energy meters for electric power distribution installed at customer homes. This device should minimize electric power thefts and reduce maintenance and repair costs. It should also reduce the number of energy failures due to an overload in the transmission network. It has a sensor to detect the violation of the meter box and performs cryptography and forward error correction to be able to communicate with a palmtop computer through an external RF transceiver, as shown in Figure 2.

The whole system works as follows: the DigiSeal installed in the box will be monitoring the violation detector and store a certain number of violation events with time and date. The RF data transmission will only take place if it receives a status request from a PDA within the area. The PDA used by the controller sends a status request to the DigiSeal. The DigiSeal sends its identification and status information ("OK", "VIOLATED" or "IN MAINTENANCE") back to the PDA. Then, the DigiSeal can send the list of violation events to the PDA on request.

In order to achieve the overall system project’s objective, the DigiSeal needs to be able to capture the position of an electric switch connected to the lid of the power meter box which acts as violation detector, perform cryptography and forward error correction (FEC) [7, 8] for RF communication and interfacing to a RF transceiver.
The project flow used for the DigiSeal was composed of the following steps (see Figure 3):

**Step 1 – Specification:** Functional requirements listing. In this step the functionalities of the circuit was captured; The algorithms to be used to obtain the desired functionality was specified in this phase; Detailed circuit specification. Here occurred the decomposition of system circuitry in its main functional blocks, with specific block parameter details. VeriSC methodology was adopted for hierarchical decomposition using TLM (Transaction Level Models) for the whole circuit and each of its blocks;

**Step 2 – Testbenches:** VeriSC methodology was used to write all the testbenches needed;

**Step 3 – RTL and Debugging:** RTL and Debugging: RTL implementation of each functional block. In this step the synthesizable description of each block was written and
debugged using the testbenches provided by step 2; Block integration and functional verification of the whole circuit using the testbench provided in step 2;

**Step 4 – Synthesis**: Per block synthesis, through an automatic synthesis tool (Altera Quartus 5.0 [9]);

**Step 5 – Simulation after Synthesis**: Functional verification of the netlist resulting from synthesis of each block, including logic gate delays. Uses testbenches from step 2; Logical synthesis and layout synthesis of the whole circuit; Functional verification of the whole circuit netlist including interconnection delays;

**Step 6 – Prototyping**: Circuit implementation using Altera FPGA Cyclone II EP2C35 [10];

**Step 7 – Layout**: Logical and layout synthesis for silicon. In this case, this step aimed only at estimating silicon area, without accounting for test structures.

The figure 4 shows the transaction level schematic of the DigiSeal.

![DigiSeal transaction level model](image)

**Figure 4. DigiSeal transaction level model**

It works in the following way: the data come encrypted and with transmission errors through transaction RFrequest. They enter the FEC block that uses the Golay [11] algorithm to correct the data. Then, Through the CRIPTRquest transaction the data enter the Cryptor, that decrypts the encrypted data using the DES protocol [7, 8, 12]. At this time, the data will go through the Request transaction to enter the StateMachine block where the DigiSeal status and a history of violation events are stored. The sensor switch is connected to this block. Next, the data exit through the transaction Reply. It enters the Cryptor block to be encrypted. Finally, the data go out through the CRIPTreplay transaction to the EEC block, which implements the Golay error code generation algorithm. Then the data is leaves through an RFreply transaction.

The blocks of the DigiSeal, Figure 4, were coded in Verilog and VHDL and SystemC was used for the Reference Models and testbenches [11, 12]. Each block of the DigiSeal corresponds to a block of the DUV that should be verified independently. The corresponding RMs must have the same hierarchical granularity and functionalities.

### 3.1. Hierarchical decomposition of the Reference Model

The RM of the whole DigiSeal is decomposed hierarchically. The decomposition of the RM must be equivalent to the intended hierarchical decomposition of the DUV. Each block resulting from the decomposition of the RM is treated as a RM of its own and subject-ed to tests using Pre_Source and Pre_Sink.

This generation of testbenches of the hierarchical decomposition of the RM is reported in Figure 5.
This sub-step uses the overall RM, Source and Checker to verify if the RMs linked together are functionally equivalents to the original RM. In order to perform it, the Source will input data in both, the overall RM and the composition of the RMs of FEC, Cryptor, EEC, and StateMachine, and the Checker will compare the results. After this step, each block will have its own RM block verified and a testbench is created for each of the individual blocks.

3.2. Testbenches for each block of the DUV

This step generated the testbenches for each block resulting from the hierarchical decomposition of DUV and RM. The individual testbenches are created in a similar way as the testbench for the complete DUV (subsection 3.1).

To create the testbench for each block FEC_RM, Cryptor_RM, EEC_RM, and STATE_MACHINE_RM, the Source and the Checker of each block were tested and then the TDriver(s) and the TMonitor(s) of each block were also tested. Many parts could be reused: In the testbench for Cryptor, we reused TMonitorRIPTreq and TMonitorReply from the testbench for FEC and State Machine respectively, because the output interface of FEC and State Machine were the same as the input interface of Cryptor. The testbench for Cryptor reused TDriverRequest and TDriverRIPTreply from the testbench for State Machine block and EEC block respectively, because the output interface of Cryptor was the same as the input interface of State Machine block and EEC block. All elements could be reused in some of the necessary configurations. This reuse is very important because it greatly reduces the risk of interface inconsistencies when linking the blocks, in the next step.

This substep consists of replacing the group (TDriverRep, STATE_MACHINE_RM, TMonitorVio and TMonitorReq) by the corresponding complete DUV assembled from the RTL block, as indicated in the example by the State Machine block in Figure 6. This includes the RTL design task that appears inserted into the VeriSC verification methodology.
This is made for verifying all blocks of the design, they were linked together and verified. Then, the last step is to replace the whole DUV testbench (subsection 3.1, Figure 5) by the complete DUV assembled from the RTL blocks to perform functional verification of the complete DUV.

In this way, each block of the DUV was verified, they were linked together and a regression test was done using the overall testbench to see if the communication between blocks did not introduce errors.

3.3. Timed transactions in the State Machine block

In DigiSeal the violation detector and stores a certain number of violation events with time and date. This is made in the State Machine Block, when the data arrives for the Request transaction, it stores the DigiSeal status and a history of violation events. The State Machine also reacts to the arrival of the Violated transaction that modifies a history of violations events. This last transaction is asynchronous in relation to Request transaction.

To solve this problem, transactions with time were inserted in the testbench of the State Machine (see Figure 6). This synchronized the transactions Request and Violated during its passing through the fifos of the Reference Model. In practice, that is implemented in the following way: The whole testbench was written in SystemC, except DUV (Verilog). In the module Source from the State Machine, randomized transactions were created for the interval of time of the simulation using constraints of the SystemC, for example:

```cpp
01. class Req_constraint_class: public scv_constraint_base { (...)
02.    simulation_time_distrib.push(pair<int,int>(15000, 40000), 100);
03.    req_sptr->simulation_time.set_mode(simulation_time_distrib);
04.    req_sptr->time.set_mode(time_distrib); (...));
05. class viol_constraint_class: public scv_constraint_base { (...)
06.    viol_sptr->simulation_time.set_mode(simulation_time_distrib); }};
07.
08. SC_MODULE(source) {
09.    Req_constraint_class Req_consraint; viol_constraint_class viol_consraint;
10.    sc_fifo_out <Violate *> violate_modref, violate_duv;
11.    unsigned int last_simulation_time = 0; Violate vio;
12.    while(1) { if(type=="Violate") { vio = viol_consraint.viol_sptr.read();
13.        vio.simulation_time += last_simulation_time
14.        last_simulation_time = vio.simulation_time;
15.        if(last_sensor==CLOSED) last_sensor=OPEN; else last_sensor=CLOSED;
16.        violate_modref.write(new Violate(vio)); violate_duv.write(new
17.            Violate(vio));
```
In the code above, the lines 13, 14, 20, 21 and 22 are creating a probability distribution to the simulation time and it guarantee that the time of simulation is increasing and that it is being stored. In the TDriver_vio and TDriver_req the lines 28, 29, 30, 37, 38 and 39 show the recovery of the time simulated in the Source and the capture of the current time of simulation. Then, they are compared, if they are asynchronous, the wait command will help to synchronize the time of simulation of the transaction.

In the TMonitor_vio and TMonitor_req the lines 44 and 50 show that the times of simulation are already synchronized in relation to the current time of simulation. Therefore, they are ready to send the data for the Reference Model.

In the State Machine Reference Model, the line 53 presents the functions that execute the functionalities of the request and of the violation respectively. The line 55 returns a reference to an event which will be notified at the time of a writing in the fifo. In this case, the event can be request or violation. In the lines 60 and 61, the synchronized capture of the event of the request with the violation event happens. When the request transaction exist, but the violation transaction doesn't exist in the Reference Model, the time of simulation of this transaction it will be the same of the request. When a request and violation transaction exist, then the time of simulation of the request will be the same time of simulation of the violation. This guarantees the synchronization of these transactions that originally were asynchronous. Only then, the exec_request() function will work. The processes of the lines 62 and 63 happens in a similar way.
while(req_ptr || viol_ptr || request.num_available()+violate.num_available() > 0){
  if(req_ptr==NULL && request.num_available() > 0) req_ptr = request.read();
  if(viol_ptr==NULL && violate.num_available() > 0) viol_ptr = violate.read();
  if(req_ptr && (viol_ptr==NULL || req_ptr->simulation_time <= viol_ptr->simulation_time)) exec_request();
  else if(viol_ptr && (req_ptr==NULL || viol_ptr->simulation_time <= req_ptr->simulation_time)) exec_violate();
} SC_CTOR(State_Machine_RM){ (...)}

4. Results

The results consist of the case study prototype, after functional verification of the individual blocks, the regression test of the top level DUV was done. It presented only one error due to a functionality that was not correctly covered at block level verification. Only one day was spent in debugging and solving the problem. After conclusion of the verification phase, logic synthesis was done in order to generate a netlist with delay back annotation for the target FPGA device Altera Cyclone II EP2C35 [10].

The same functional verification testbench was used for post-synthesis simulation. The FPGA implementation was fully functional at the first attempt, proving the efficiency of the VeriSC verification methodology adopted using timed transactions in some blocks. For area estimation, a silicon layout for AustriaMicrosystem's C35B3 technology [13] was generated using Astro from Synopsys [14]. The resulting area was 2.2 x 2.2 mm². See Figure 7.

![Figure 7. Layout from DigiSeal project.](image)

The DigiSeal project had a shorter development time, with only two employees working for six months. The reasons for this are:

- The complete superposition of RTL implementation and functional verification;
- No need of extra code to do RTL code testing;
- Less time spent in testbench writing because of code reuse;
- Consistency of interfaces between blocks of the hierarchical decomposition easily achieved by systematic code reuse.

The results show that VeriSC is more efficient than a traditional verification methodology.

5. Conclusion

VeriSC methodology proposes the implementation of the testbench before the RTL development, with no extra code and a hierarchical approach. This also permits that the interface TDrivers and TMonitors may be reused in a way to assure interface consistency.
throughout the hierarchical decomposition process. It also shows to be efficient in the solution of problems with asynchronous transactions.

This paper presented the VeriSC functional verification methodology applied to the case study of the project of the DigiSeal circuit. After simulation and synthesis steps were finished, the FPGA implementation was an immediate success. Thus, the efficiency of the VeriSC methodology could be shown.

References


