POLITECNICO DI TORINO

Master's Degree in Electronics Engineering - Embedded Systems

Master's Degree Thesis

SystemC Transaction Level Modeling of an ASIC CAN controller and simulation performances comparison with an RTL model in VHDL

Supervisor

Prof. Luciano LAVAGNO

Tutors

Ing. Alberto BERTONE

Ing. Riccardo MASANTE

Candidate

Gianmarco RUSSO

A.A.2020-2021

Abstract

The main goal of the thesis work is to verify some of the improvements brought by an innovative digital system modelling approach, SystemC Transaction Level Modelling, which is based on the abstraction of inter-module communication from the architectural details of the functional units and the communication protocols.

The parameter under verification is the simulation performance of the behavioural model of a CAN controller peripheral device, described in SystemC TLM1.0, with respect to a CAN controller described in VHDL. In order to obtain a valid comparison of the simulation times it is needed to have two devices with the same design complexity; therefore, the two versions are similarly described, namely they have the same external interfaces, they include the same type of sub-modules and their main threads, managing the frame transmission, reception and processing, have similar body structures.

Each of the two modules is tested by a functional testbench, performing a determined amount of frame transmissions between two CAN nodes and verifying the correctness of each transmission by collecting the processed frames from the receiving node. Thus, the test applied on the TLM device and on the VHDL benchmark are the same and they run on a common server, in order to measure the simulation performance in terms of the amount of real time corresponding to the same simulation time interval.

Acknowledgements

Da inserire dopo revisione professor Lavagno

Table of Contents

List of Tables

List of Figures

Acronyms

HW	Hardware
SW	Software
ASIC	Application Specific Integrated Circuit
RTL	Register Transfer Level
VHDL	VHSIC Hardware Description Language
TLM	Transaction Level Modelling
SoC	System on Chip
CA	Cycle Accurate
MIPS	Million Instructions Per Second
ISS	Instruction Set Simulator
DUT	Device Under Test
IPC	Inter Process Communication

MPSoC
Multi Processor System on Chip

NoC
Network on Chip

OS
Operating System

AMS
Analog Mixed Signals

OSCI
Open SystemC Initiative

HLS
High Level Synthesis

EFSM
Extended Finite State Machine

ECU
Electronic Control Unit

uC
Microcontroller

CRC
Check Redundancy Check

FIFO
First In First Out

DMI
Direct Memory Interface

CAN
Controller Area Network

CAN FD
CAN Flexible Data-Rate

PDU
Protocol Data Unit

Introduction

The design flow adopted in the development of application specific electronic devices involves, at the first stage, the functional description of HW architecture, which is necessary for the HW tests and validations and the possible SW development. In particular, if the device includes some software running on, or even an OS developed on a custom architecture, also all the SW developing, debugging and validation procedures have to be performed subsequently. For these reasons, the whole process could be optimized if the architecture modelling latency is shrunk, or if there is a way to develop SW and HW in parallel, with the SW running on an abstract platform that functionally replicates the HW behaviour. In addition, with the growing complexity and modularity of electronic systems, an efficient architecture exploration among block layers is fundamental to go deep at different levels of implementation, while considering that ASIC systems reach better performances by integrating the SW to the specific HW. At the time, the most used languages for ASICs are RTL description languages like VHDL and Verilog, allowing to design electronic structures of any complexity at the deepest levels of detail, namely the *Register Transfer Level* (RTL) and *Gate Level.* Nevertheless, even if they provide a satisfying exploration, every system should be developed from scratch because there is no reuse of the code, except for some basic modules, then the embedded SW can be developed and simulated only after the compilation of the validated HW architecture. Therefore, it is not possible to proceed with a preliminary high level formal verification (e.g. application or architectural level tests); moreover, there is no inter-module communication protocol by which the signals between modules are controlled, it represents a boundary for any processor based system. An innovative approach based on SystemC language is gaining momentum in the HW design, intended to solve the practical problems shown above by: - providing a platform for hardware dependent software development in early stages of design flow, - allowing system level design exploration and verification, - creating system level models for block level verification, - reusing the code between projects and between various abstraction levels in the same project The aim of the presented work is to demonstrate the improvements of the TLM modelling w.r.t. the RTL one in terms of simulation performances, by creating an ASIC system (a CAN controller device) in both languages and simulating them for the same amount of time (simulation time) while assessing the computation time it takes to simulate each model. Taking into account that the TLM model will be written in a mixed code style, combining *approximately-timed* modules with *cycle-accurate* ones, we expect the TLM version to have a clearly faster simulation than the RTL.

Chapter 1

TLM paradigm, state of the art and fields of application

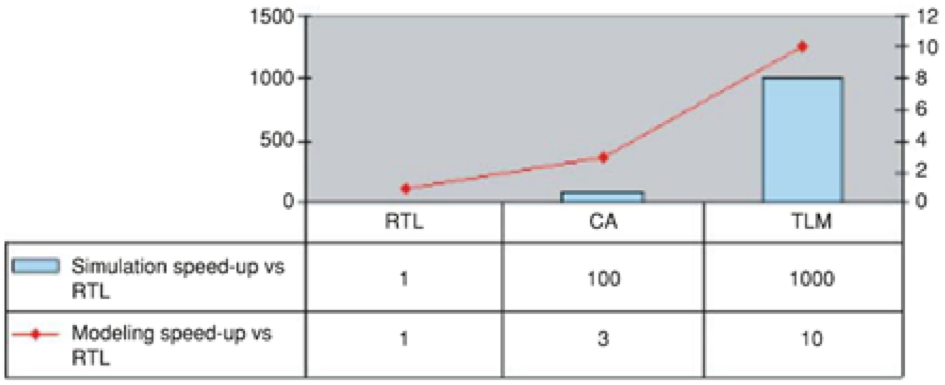
1.1 Comparison between TLM approach and RTL modelling, advantages and use cases

The complexity of System-on-Chip (SoC) systems is growing continuously at the same pace of the scale of integration and, combined with the increasingly reduced time to market, it is becoming a problem for system developers as well as SoC architects because more resources and time are required to design new systems. To overcome this critical issue, it is fundamental to grow the abstraction level for integrated devices. In general, abstraction brings different vantages in the flow of system design:

- the hierarchical structure allows to change between behavioural and structural implementations of each module according to the kind of usage of the system, such as simulation, validation or synthesis, without changing the top module [1];
- heterogeneous designs, which see different kind of processors working in parallel, can be abstracted and implemented more easily in their complexity [2] [3];
- also, software and hardware can be abstracted, leading to a parallel design that overtakes the "software over hardware" limit [4] [3] [2] [5];
- the testbench can be abstracted, thus the test development and validation delays can be reduced.

The new design methodology described above, which joins the system level design on top of the RTL level, in turn on top of layout level, is achieved in this thesis work by adopting the Transaction Level Modelling (TLM) paradigm on SystemC language, it divides the intercommunication details from the architecture of functional and communication modules.

Until now hardware and software design world has been split about TLM and the enhancements it brings: the skeptical think that SystemC behavioral models are a Verilog/VHDL license-free version, gaining the same simulation performance, while the opposite mindset sustains the same theory of this work, namely that TLM simulates faster than RTL and is necessary to speed up the time to market, taking in consideration that if the SystemC design is accurate enough it can be translated into RTL thanks to appropriate compilers. In figure 1.1 is shown the comparison between RTL, Cycle Accurate (CA) and TLM concerning design and simulation. As you can see the TLM latency is decreased up to 10 times for modelling and by 1000 times for simulation. The actual goal of this work is to verify that the innovative description model reasonably leads to this performance improvement.



	RTL	CA	TLM
▭ Simulation speed-up vs RTL	1	100	1000
—◆— Modeling speed-up vs RTL	1	3	10

Figure 1.1: *Efficiency of different modelling strategies.*

Different TLM standards have been released since its creation for SystemC language, but the main changes are seen from TLM1.0 to TLM2.0 [6] [7] [1], that will be described in the current paragraph.
The TLM1.0 standard is based on 3 concepts: interfaces, blocking vs non-blocking mechanism, unidirectional vs bidirectional transfers. Interfaces are the core of the TLM1.0 standard cause they define the type of provided service, they inherit from class `sc_interface`. Designing using interfaces comes from the already existing C++ interface concept, fundamental for the object oriented programming, applied to the TLM design. The OSCI TLM1.0 standard follows the use of the terms "blocking" and "non-blocking" in a strict way: if a TLM interface is labelled "non-blocking", it means that it contains only methods can never call `wait()`, while a "blocking" interface has methods which can contain different `wait()`, as summarised in table 1.1, making possible to suspend some processes running on a module inheriting

the blocking interface. However, the two kind of methods are indistinguishable by the signature, in fact it is necessary to go into the implementation details of each method to recognise if it is blocking or unblocking.

OSCI Terminology	**Contains wait(.)**	**Can be called from**
Blocking	Possibly	only SC_THREAD
Non-Blocking	No	SC_THREAD or SC_METHOD

Table 1.1: *SystemC blocking vs. non-blocking interfaces.*

The concept of bidirectional and unidirectional transactions is implemented since TLM1.0, they are the fundamental transfer for communication protocols of any complexity. In general, a transaction (or transfer) is a method called by one initiator module, possibly passing through a dedicated channel and implemented by the target module; and this approach allows to adapt the transaction based on the communication protocol and the application. Both unidirectional and bidirectional interfaces are based on the `sc_fifo` interface and their methods can be implemented by a channel or directly with `sc_export`.

Considering that, we can have 2 different kinds of unidirectional interfaces:

1. Blocking unidirectional interfaces, they are: `tlm_blocking_get_if`, `tlm_blocking_peek_if` and `tlm_blocking_put_if`;
2. Non-Blocking unidirectional interfaces, they are: `tlm_nonblocking_get_if`, `tlm_nonblocking_peek_if` and `tlm_nonblocking_put_if`. These may fail and return a boolean value indicating whether they succeeded.

The bidirectional interface simply implements methods that can be seen as a join between the unidirectional methods, and there is no bidirectional non-blocking interface. In the following all interfaces implemented in channels or exports (`sc_export`), and defined in the TLM1.0 standard, are defined:

- `tlm_req_rsp_channel<REQ,RSP>`, which extends the unidirectional interfaces by implementing a FIFO for the initiator to target requests and a FIFO for the target to the initiator responses;
- `tlm_transport_channel<REQ,RSP>`, which extends the unidirectional and bidirectional interfaces by implementing FIFOs of size 1 to bound every response to its request.

In figure 1.2 is shown the SystemC TLM modelling template for every system design, which applies the abstraction concept to obtain multi-level system architectures. It is composed of two interfaces, that are the convenience interface and the TLM interface. The former allows the user to use methods like read and write at the application level (or user level), which in turns are implemented at the protocol level by designers, while the latter has the function to protect the transport methods, defined by the TLM standard library, from the protocol layer.

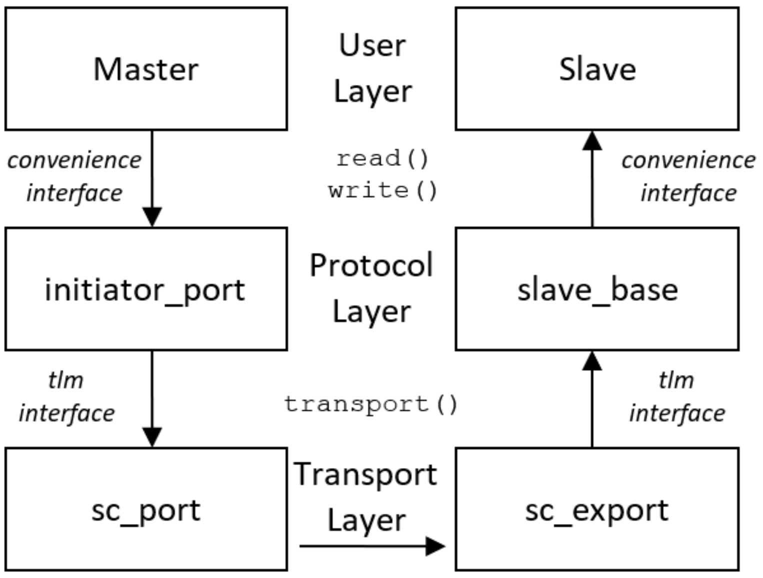


Figure 1.2: *Standard SystemC TLM structure, representing the template for every TLM design.*

The general use of the structure sees the user employing, at the top layer, the initiator ports of the master module, which uses the previously described interfaces to generate transaction calls (unidirectional or bidirectional) which are implemented at the transport layer and arrive at the slaves' exports. All slaves implement their own functions at the protocol layer consisting of requests and responses to the master, used in turn by the user layer methods.

This structure is not compulsory for every design because there are distinct ways to build SoCs with TLM, avoiding the use of some layers at all or using a mixed structure to design the modules and connect them together. As a first example, the master can forward bidirectional read and write transactions to the slave, which is described at the abstraction level, using a bidirectional blocking interface (figure 1.3); but it is possible also to connect master and slave through a TLM channel, describing the slaves with the at the abstract level (figure 1.4)) or implementing them at the RTL level (figure 1.6). In the last case it is needed to have a transactor in order to be compliant to the TLM architecture.

If the master drives more than one slave through a bus-based connection, a structure using the router like in figure 1.5 should be implemented, it is fundamental for the mapping of each slave address to the master one, in order to decode every transaction coming from the master and forward them to the involved slave. When indeed an arbitration between parallel masters is needed, a structure like in figure 1.7 is necessary: the masters requests are stored in the request FIFO of the TLM channels, the arbiter forwards them by priority and the responses are put back in the response FIFO of the relative channels.

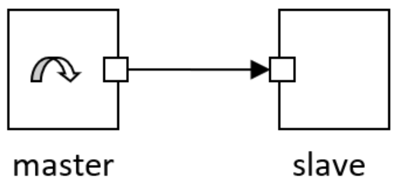


Figure 1.3: *Master and slave produced as loosely-timed models, connected with exports.*

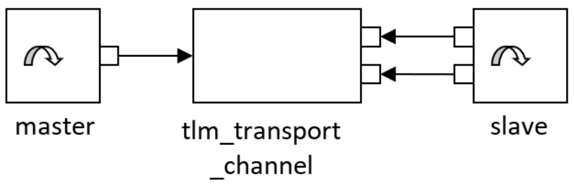


Figure 1.4: *Master and slave produced as loosely-timed models, connected by a transport channel.*

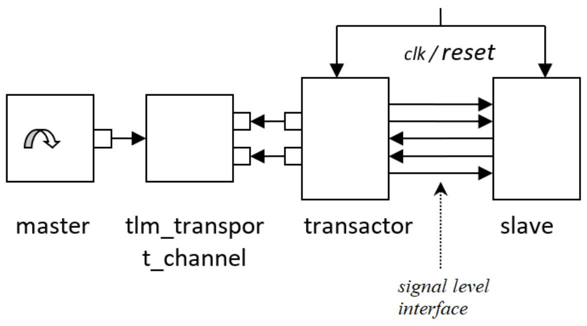


Figure 1.5: *Master produced as loosely-timed model, slave described at RTL and interfaced to the architecture through a transactor.*

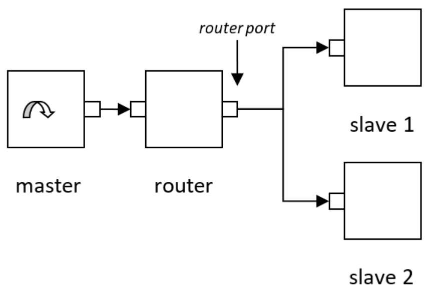


Figure 1.6: *Router insertion to allow master drive multiple slaves with a relative protocol.*

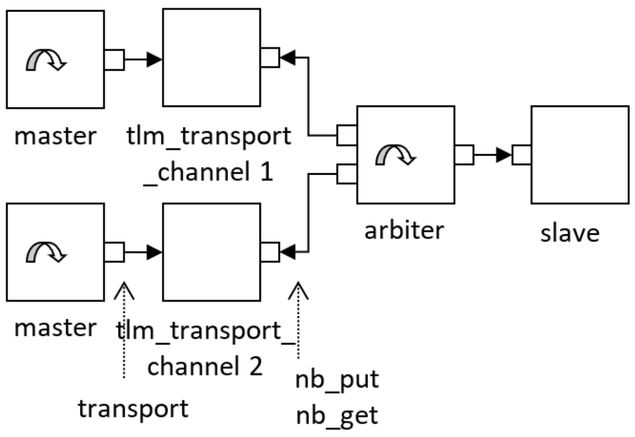


Figure 1.7: *Arbiter insertion to allow multiple masters drive the same slave.*

The next upgrade regarding the TLM standard is TLM2.0, which replaces the TLM1.0 in many designs to overcome the problem of the increasingly high level of abstraction of the computations on which the TLM1.0 becomes too slow in simulation and verification. Moreover, TLM 1.0 does not allow the compatibility between models created by different vendors and used in the same design; thus, if a system would be composed by several interconnected subsystem designed in different ways in TLM, it is not sure that they can interface among themselves due to different design styles adopted using TLM1.0 standard. TLM 2.0 standard inherits the basic libraries from TLM1.0 but, among all enhancements that will be described in the following, solves this boundary defining an API which enables model interoperability and providing several primitives to model arbitrary components and systems.

The SystemC TLM 2.0 has grown up from different solid bases:

- Possibility to implement loosely-timed and approximately-timed coding styles;
- Creating API to define SystemC TLM guidelines;
- Interoperability between models designed by different vendors;
- Unified interfaces and sockets for memory-mapped bus-based models design;
- Inter-module communication based on the generic payload object.

Moreover it provides:

- Direct-memory interface;
- Model synchronisation.

The API suggests data types for low level designs and introduces specific data structures, allowing to model a memory-mapped bus structure template, except for specific protocols that cannot be inter-operable with the TLM structure.

SystemC TLM2.0 standard provides different implementations for each module in a design by defining two coding styles that can be used in every level of abstraction in the system. The first style creates loosely timed models, useful for SW development, SW verification and performance analysis of the systems because they are fast enough to execute more than 1000 MIPS. The other style creates approximately timed models nearer to the real models, suitable for the architecture exploration and synthesis, but the detailed description weighs on performances, which remain under 1 MIPS.

In other terms, the former style defines what are called the software virtual platforms and allow to model timers and interrupts, so that one or more operating systems can run on the platform; it provides temporal decoupling by executing all processes in zero simulation time and synchronising them every time amount which reduces overhead. In this case, the determinism is guaranteed by two factors: the dependency control performed by the scheduler, and the quantum value, the maximum time each process of the design can run before the re-synchronisation of the simulation. The quantum influences the simulation speed because the synchronisation frequency is inversely related to its value.

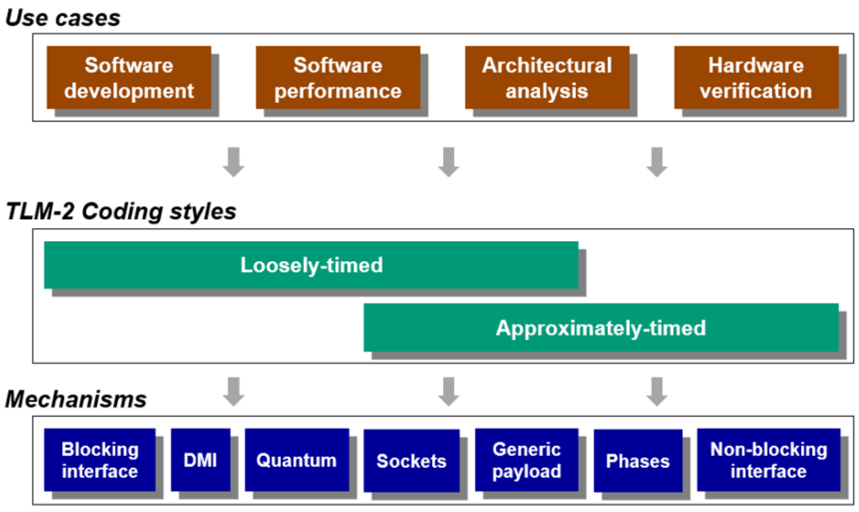


Figure 1.8: *Resume scheme of the TLM2.0 paradigm.*

The approximately timed modelling style defines hardware virtual platforms which tend to be complex, low level and cycle accurate; and it can be realised that the temporal decoupling cannot be applied on this style cause the simulation time advance every time any process generates some events. For this style the transactions are performed in four timing points that correspond to the start and end of request and response. These styles can be used together creating a hybrid system, in fact TLM2.0 allows interconnections between models implemented with different styles and, if the models are described in both ways, the style implementation can be switched before running the system. All use cases, coding styles and mechanisms belonging to TLM2.0 standard are shown in figure 1.8.

The main change brought to the TLM2 standard with respect to TLM1 is the transport mechanism and the structure adopted to accomplish that. As it can be seen in figure 1.9 every transaction pass between two modules, it is started by the initiator module and arrives at the target module and the components are free to act both as initiator and target, this is the case of modules used for interconnecting other initiators and targets (for instance a router, a channel or an arbiter).

As already mentioned, the transaction object passed in TLM2 is the *generic payload*, which is created by an initiator before the starting of transaction; then the transaction is performed passing the object from *initiator* to *target* through the *forward path*, passing from all interconnect components, successively the target can return back the transaction object following two alternative paths: the *return path*, at the return of the transport methods called for the transaction, or the *backward path*, if methods in opposite direction are made on purpose.

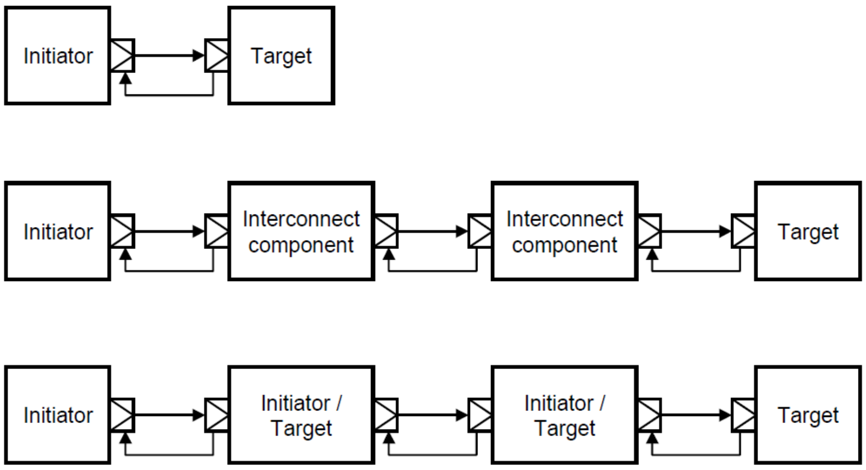


Figure 1.9: *General scheme of TLM2.0 transaction of a generic payload object. From top to bottom: A direct TLM transaction from initiator to target, a TLM transaction made of forward path and return path, a TLM transaction made of forward path and backward path.*

Forward and backward paths supporting generic payload transactions traverse the same modules and in particular the module interfaces for transport are no more simply ports, indeed they are replaced by TLM sockets, which are composite ports supporting the generic payload object as well as DMI and debug transport interface. Going into detail, a socket is composed of a port and an export which must be bound together. There are different types of sockets: they are categorised as initiator socket and target socket because an initiator socket should provide a port to make possible methods calls through the forward path and an export to receive the return object of the transaction, while the reverse structure is provided by the target socket.

The TLM2 paradigm includes the TLM1 core interfaces and provides additional interfaces that are: the blocking interface, the non-blocking interface, the DMI and the debug transport interface.

The transport interface comprehends the blocking and the non-blocking transport interface, the latter has a return value indicating if the slave involved in the transaction has transmitted back the response through the return path.

The TLM2 blocking transport interface allows transactions invoking the `b_transport` method which passes two arguments: the generic payload object, passed by reference, and a timing annotation, the delay the request takes to arrive at the slave starting from the current simulation time; also, at the transaction end the timing annotation is used to return from the blocking transport to indicate the additional delay.

Elaborating on the generic payload details, it is a structure composed variables which can be all used or only in part. They are:

- Command, that specifies if it is a read or a write transaction;
- data;
- address;
- byte enables;
- streaming width, defining the length of the streaming burst;
- response status, a variable indicating how the transaction has gone;
- DMI hint, set to 1 if Direct Memory is supported by the target module;
- extensions.

The sequence chart shown in figure 1.10 explains better a typical blocking transport example, where it is possible that other transactions can be called by different processes running on the initiator while the thread is blocked. You can see in particular the case in which the initiator makes two consecutive calls: the first is returned in zero time cause the target module did not have any wait in the used method, while at the last transaction the target waited 40 ns of simulation time before returning, also stopping the initiator thread involved in the transaction.

As known, both loosely-timed and approximately-timed modelling styles support temporal decoupling, it is implemented both by blocking and non-blocking transport interfaces, using timing annotation provided by `b_transport` and `nb_transport` methods. If the blocking transport interface is implemented when the initiator and the target modules are modelled at the loosely-timed style (blocking interface is suggested for loosely-timed modelling), their threads can run ahead of the simulation time adding a local time offset and calling multiple transactions, until a *wait()* is encountered in a `b_transport` method. Here all the initiator threads re-synchronise with the simulation time and the local time offset is reset, as you can see clearly in figure 1.11.

If indeed the approximately-timed modelling style is adopted, the non-blocking transport interface should be used in order to exploit the transactions pipelining and model every transaction step and detail; in fact the non-blocking interface can split a transaction into different phases and annotate the time of each one of them, moreover it is composed of forward and backward transactions between initiator and target that are not required to be called one after the other, thus allowing pipelining.

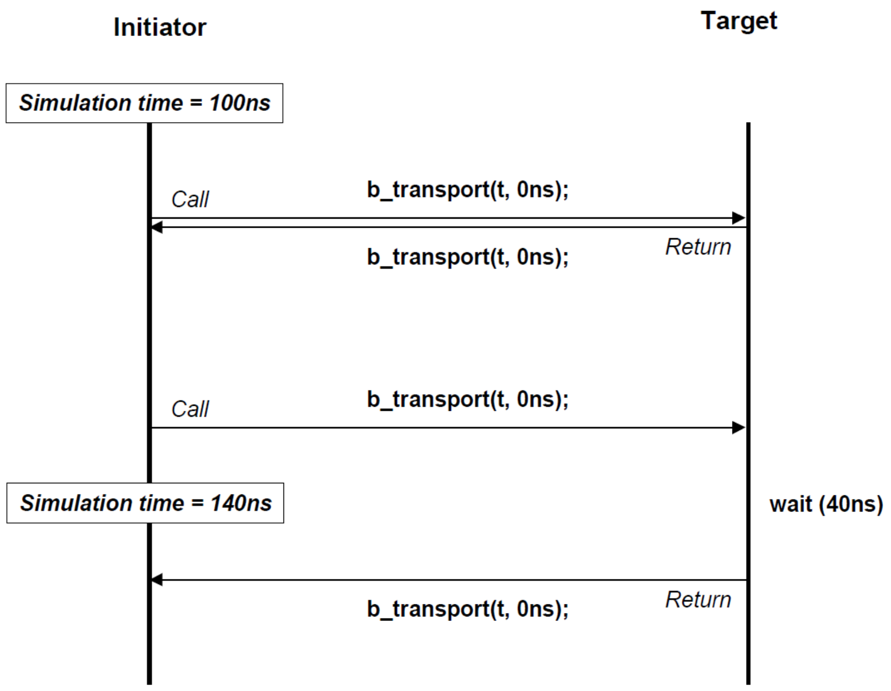


Figure 1.10: *Simulation window using blocking transport interface.*

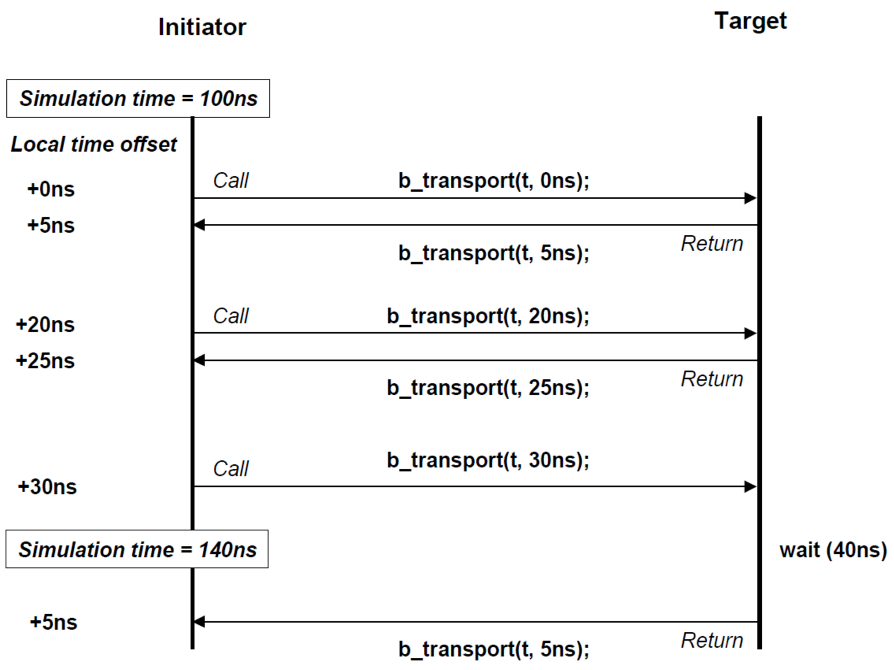


Figure 1.11: *Simulation window using blocking transport interface with temporal decoupling.*

The non-blocking methods pass one additional argument with respect to the blocking method, that is the transaction phase; it gives information about the state of the considered transaction, which can be: **BEGIN_REQ**, **END_REQ**, **BEGIN_RESP**, **END_RESP**.

In addition, the non-blocking methods returns an enumerated variable which can assumes a value between: **TLM_ACCEPTED**, returned by the destination module when it has received and accepted the transaction call; **TLM_UPDATED**, returned by the destination module when it has received the transaction call, has possibly modified the transaction object (generic payload) and has added a delay on the timing annotation; **TLM_COMPLETED**, when the transaction object has been modified by the destination module and the transaction is completed.

Due to the fact that at each transaction phase, the scheduler takes control from the processes of the caller in order to proceed in simulation time, the non-blocking transport mechanism is simulated slower than the blocking one.

Figure 1.12 shows a typical example of a non-blocking transport sequence, where the difference with respect to the blocking one is highlighted on the number of calls, while in figure 1.13 the temporal decoupling is applied through the timing annotation variable.

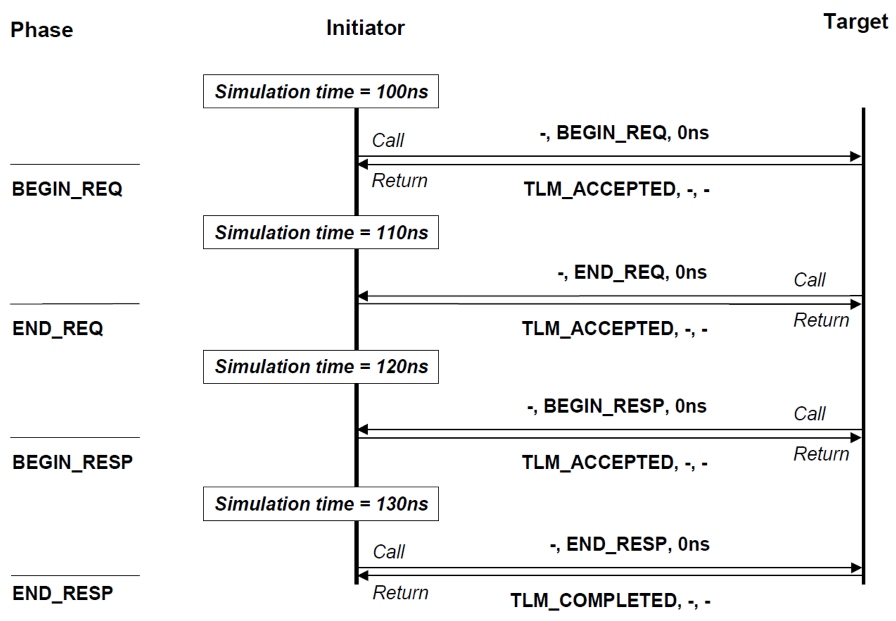


Figure 1.12: *Simulation window using non-blocking transport interface.*

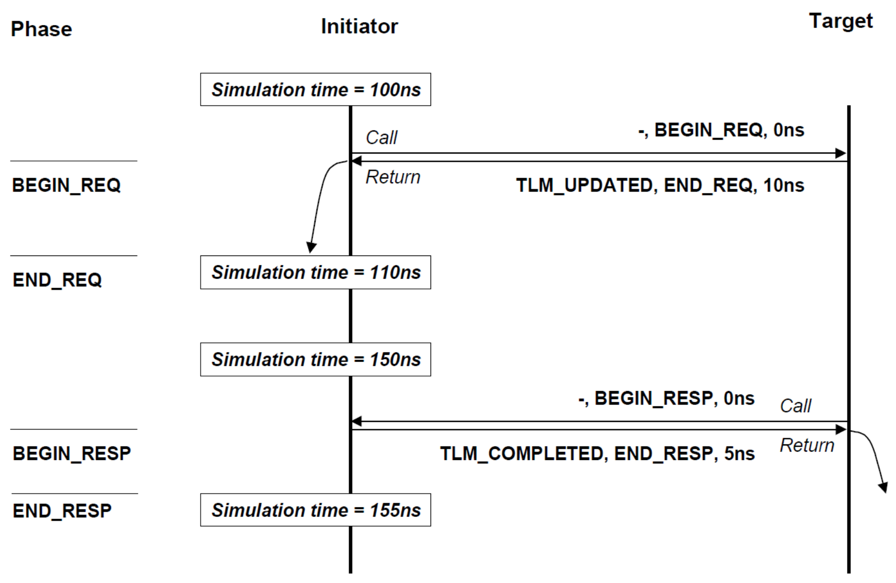


Figure 1.13: *Simulation window using blocking transport interface with temporal decoupling.*

1.2 Overview on Controller Area Network: protocol and applications

1.2.1 CAN standard

Controller Area Network (CAN) protocol is a standard (ISO-11898-1:2015) that manages a multi-cast, multi-master, serial communication system for field bus. It was developed by Bosch to be used in the automotive industry because it allows the connection between different ECUs by two wires and over time the characteristics of CAN have spurred its use to many industry fields like medical, building automation and manufacturing.

The key peculiarities which make CAN protocol a ductile standard for system design are: the ability to detect and correct errors on different layers of the OSI hierarchy, the immunity to electromagnetic noise and interference, the low cost coming with relative high performance.

As previously mentioned, the CAN protocol respect the ISO/OSI architecture and provide the communication service for the two bottom layers, that are data link layer and physical layer. As it is shown in figure 1.14, also the application level is considered in the CAN, intended as an interface with the specific protocol used at the higher level.

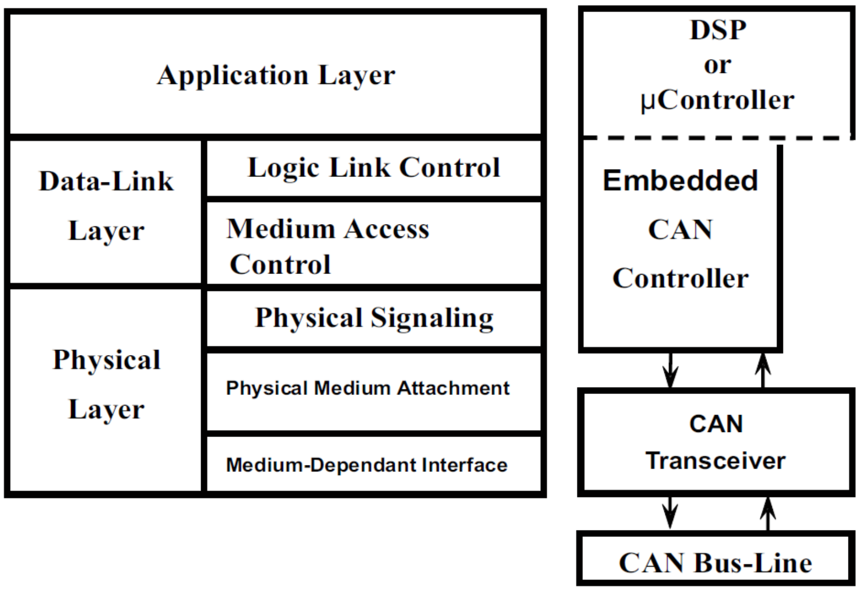


Figure 1.14: *CAN stack compliant to the OSI standard layer architecture.*

CAN protocol is of type CSMA/CD+AMP, meaning that it is a multiple access carrier-sense with collision detection and arbitration on message priority. Giving a better description of the protocol, every time a frame transmission is started by a node, the first transmitted bit is sensed by all other nodes that detect the start of frame and keep reading the bus without driving it, in order to receive the frame and avoid collisions, in addition the protocol imposes the presence of a timing interval between two transmissions, during which the bus must be idle before sending a new frame, moreover every conflict of simultaneous transmissions is resolved by the arbitration thanks to the message priority.

The synchronous bus communication is based on packet transfer, also called frame transfer or message transfer; where each frame follows a standardised structure divided in fields. In particular there are two frame formats: the standard CAN frame and the extended CAN frame, where the last one is an improved version of the former with the main difference of containing a 29-bit identifier that allows to get 2^{29} identifier values.

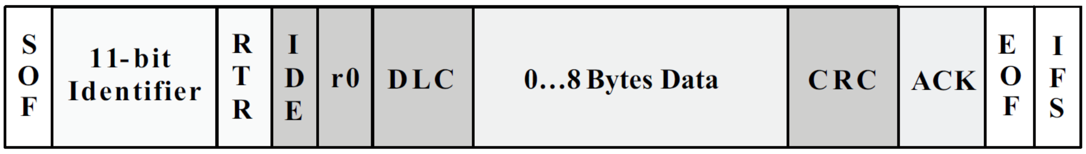


Figure 1.15: *CAN frame structure in its standard format.*

The standard CAN frame is composed as shown in figure1.15

- SOF bit, a recessive bit signalling the start of transmission of a new message;
- The standard identifier that has double purpose: storing the node ID value and giving a priority to that node;
- Remote Transmission Request (RTR) bit, which is recessive if it is intended to transmit data and it is dominant if data are requested from the node having the same identifier of the frame;
- Identifier Extension (IDE) bit;
- a reserved bit r0;
- Data Length Code (DLC) field, contains the bytes of data to be transmitted;
- Data field, the core message occupying from 0 to 8 bytes
- Cycling Redundancy Check (CRC), a 15-bit checksum for error detection on the preceding fields plus a delimiter recessive bit;
- ACK bit, transmitted by the node that receives the frame. It assumes a recessive value if there are no errors in the transmission, otherwise it is dominant;
- ACK delimiter bit, always dominant
- End Of Frame (EOF), a sequence of 7 dominant bits without stuffing, signalling the end of transmission;

- Inter Frame Space (IFS), same as EOF with the goal of giving enough processing time to all CAN nodes.

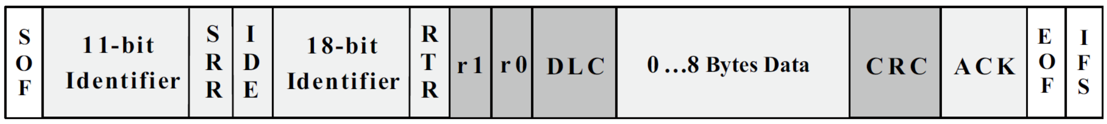


Figure 1.16: *CAN frame structure in its extended format.*

As you can infer from figure 1.16, the extended format is similar to the standard one, but it presents some changes: 18-bits of identifier extension are added after IDE, the SRR takes the position of the standard frame RTR, the IDE assumes the recessive value and there is an extra reserved bit r1 after r0.

Giving a look at the physical layer, data and clock are encoded together and transmitted on a unique differential signal carried by the CAN bus line (a twisted pair cables); in particular a specific technique is adopted to keep synchronized the clock of each node of the network, avoiding sending the clock through a dedicated signal: the bit stuffing. This consists in inserting some bits, during the frame transmission, with the goal of having at least one commutation every 5 transmitted bits to trigger the synchronization into all nodes. In this way a destuffing must be performed during the frame reception in order to extract the frame information. The only fields not subject to stuffing are EOF and IFS, helping the receivers to detect a transmission ending.

The physical CAN bus is composed of CANH and CANL signals, they are at high impedance if there is no node sending a dominant bit and they are short circuited in the opposite case, implementing a wired-AND behaviour and avoiding any collision between multiple transmissions. On the other side, it is not possible to detect on the bus if more than one node is transmitting and each node must compare its exiting bit with the read value from the bus at each clock cycle; therefore, the arbitration can be applied by halting the transmission of all except one frame per time. In fact, during the identifier transmission, each node that read a bus value opposite to the last wrote bit stops transmitting cause there is another message with higher priority on the CAN bus. Knowing that all nodes are in wired-AND, the node with lower identifier value has higher priority on the bus and can continue the transmission. The interfacing between the TTL logic and the physical bus is performed by the CAN transceiver.

A special message format that goes outside the rules of CAN frame format, the error frame, can be sent if a node notices an error in a message, also triggering the other nodes to send an error frame through the bus.

The CAN wide use in industry is overall justified by its robustness, guaranteed by five methods of error checking; in addition, fault confinement is implemented in each controller by disabling the transmission when the errors upper limit is exceeded. Error checking is implemented at the message level with CRC, ACK and all bits that should have a constant value, such as SOF, EOF and delimiters; error checking at the bit level are: firstly, the comparison between the transmitted bit and the one read from bus, at last the checking of the bit-stuffing by verifying that, during a transmission or a reception, after five consecutive bits carrying the same value there is an opposite value on the bus.

The error detection procedure applied during transmission and reception is, as mentioned, a Cyclic Redundancy Check on the message, in particular the CRC-16 polynomial implementation; it is simple to design in HW while provides a strong guarantee for data reliability and integrity. The characteristics of CRC are single bit error detection, double bit error detection and adjacent bit error detection, therefore its use represents a great advantage with respect to other techniques as parity bit check, which detects only single bit errors; moreover, it has a simpler application than other algorithms as the Hamming code.

The generic algorithm calculates the CRC code dividing the considered data by the polynomial and taking the rest for the transmission, while at the reception an error check is performed, dividing the entire message (data + CRC) by the polynomial and verifying that the result is zero. The XOR operation of the two numbers is equivalent to dividing one for the other, because the division is performed with Modulo-2 arithmetic.

Knowing how the division can be simplified in modulo-2, the HW design sees only XOR gates and a shift register; in particular, the binary data is serially shifted in the register and a XOR operation is performed at each position before shifting where the polynomial contains a 1, while a simple shift is performed where a 0 is present at the polynomial. For example, if the polynomial is 1000000000000101_2, the HW implementation is the one in figure 1.17 . The LSB is always 1, so that the bit exiting from the MSB is looped and XORed.

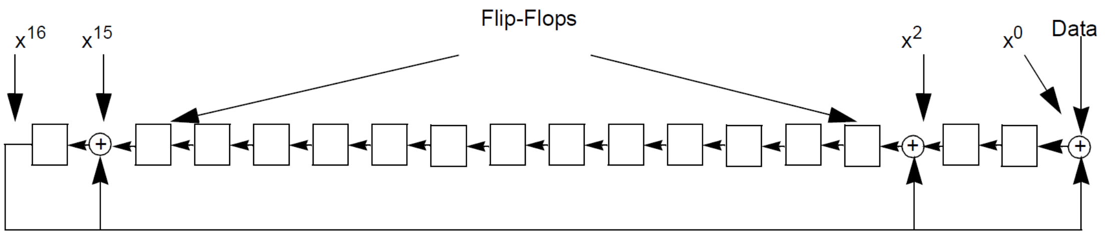


Figure 1.17: *Hardware CRC-16 generator.*

In case of the CAN protocol, the CRC polynomial, also called CRC key, assumes the binary value 1100010110011001, and the computation is performed as following:

- at the transmission, the data used for CRC computation is the frame fragment before the CRC field, left shifted of 16 bits to have 16 zeroes as LSBs;
- at the reception, the CRC check is performed on the frame portion from SOF to CRC included.

CAN is used in application where relative short data is needed to be transmitted between all nodes of the network with high data reliability and where some nodes have high priority to transmit over the majority, in order to send and request messages and take control over the system.

1.2.2 CAN Flexible Data-Rate standard: an faster and more secure communication

The main limitation of CAN protocol is the small size of data information contained in a frame, which can reach a maximum of 8 bytes. CAN FD (Flexible Data-Rate) protocol overtakes this boundary by extending to 64 bytes the upper limit of data size, but this is not the only upgrade included in the protocol. FD stands for flexible data rate, citing the feature of the CAN FD controller to choose between slower and faster data rate. As well as the CAN protocol, also the CAN FD protocol covers two different layers in the ISO/OSI architecture: the data link (DL) layer, subdivided into Logical Link Control (LLC) sublayer and Medium Access Control (MAC) sublayer, and the physical layer.

The LLC sublayer works on top of the MAC sublayer, it selects which of the arriving messages to accept, other than forwarding the exiting messages to the MAC sublayer, managing the recovery and rising overload warnings; while the MAC sublayer works at the lower level and it generate the frames from the messages provided by LLC sublayer, then it transmits the frames, manages the arbitration, the acknowledgement and the error detection and notification.

On the other way, the clock synchronization, the transmission/reception bit timing and the related stuffing/destuffing.

CAN FD benefits from many enhancements with respect to CAN standard, showed up as follows:

- the Identifier field has double purpose: other than addressing every node and setting priority to the messages, it contains information about the messages for classification and, accordingly, filtering at the receiving;
- the bit rate during arbitration is different than the one during data field transmission, transmitting data with speed up to 5 Mb per sec;
- Extended Data Length (EDL) bit takes the position of r0 CAN bit; it is forced to recessive to signal that a CAN FD message is transmitted on CAN bus;
- Bit Rate Switch (BRS) bit which, when recessive, signals that the bit rate at data field is different from the one at the arbitration phase;
- the fault confinement is performed by error counters and the Error State Indicator (ESI) in this way: there is a transmit counter and a receive counter, relatively incrementing when there is a transmission fault or a reception fault; every node has the ESI bit dominant indicating the error active state of the node, but if one of the counters exceed 127, the node goes to the error passive state and once a counter reaches the 255 value, the node switches to bus-off state;

The features described are implemented on the CAN FD frame, which standard format and extended format are shown in figures 1.18 and 1.19.

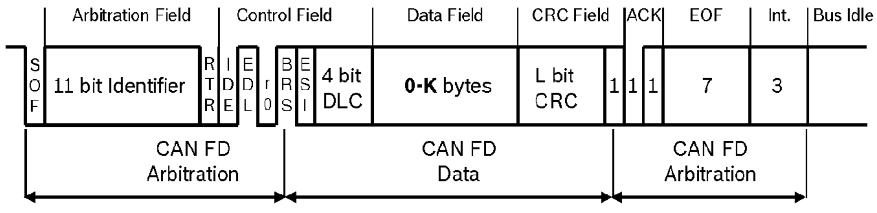


Figure 1.18: *CAN FD frame structure in its standard format.*

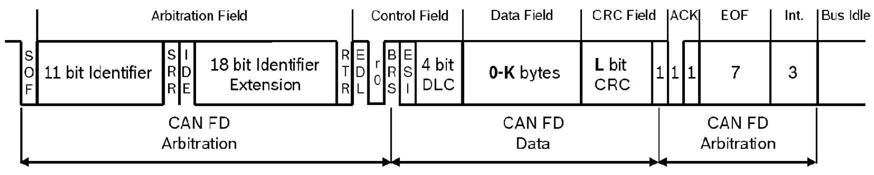


Figure 1.19: *CAN FD frame structure in its extended format.*

At the present, both CAN and CAN FD standards are used in all fields of the industry because they have their own advantages and electrical characteristics that can be adopted on different systems. That is, if the application requirements (like the bit rate, the number of nodes or the bus length) can be provided by CAN, it is usually preferred for its simplicity with respect with the CAN FD because the nodes are cheaper, vice versa the latter should be considered for better performances of the network. In general, in a system which contains multiple networks connected in an appropriate topology and interacting each other, different standards can be adopted to manage the communications and some special nodes can act as interface between them, as you can see in figure 1.20.

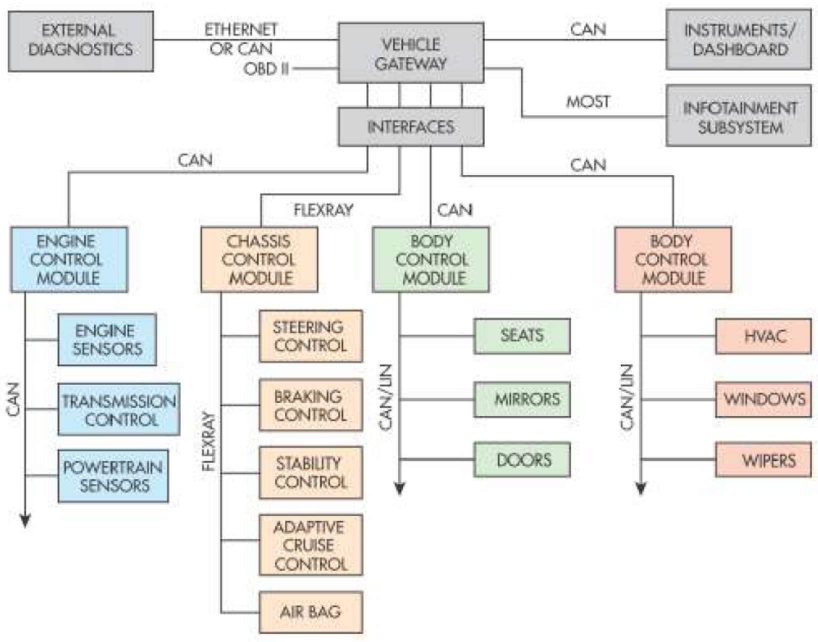


Figure 1.20: *Typical embedded system inter-communication network.*

1.3 State of the art in literature

In the last decade many studies have been carried out on the applications and improvements that the SystemC TLM brings with it in the field of SoC design, high level synthesis (HLS), system level simulation and RTL simulation, with optimal results in all sides.

The TLM effects on simulation performances has been demonstrated by Boukhechem et al. [2] who, exploiting the interoperability provided by TLM, presented a TLM co-simulation methodology to design, validate and verify an open-source Multi-Processor SoC, thanks to the integration of two Instruction Set Simulators (ISS) with DUT composed of the MPSoC hardware surrounded by the SystemC simulation environment, as shown in figure 1.21; aiming to reach a fast and efficient architecture exploration at high level.

On the other side, ISSs are interfaced by SystemC interface wrappers and communicate with the DUT via abstract channels, as is shown in figure. The aim of the work is to compare three HW implementations during software prototyping: RTL modelling and TLM at instruction accurate level and cycle-accurate level; and the results highlights how the design and validation times are reduced and there is no significant overhead due to the wrappers.

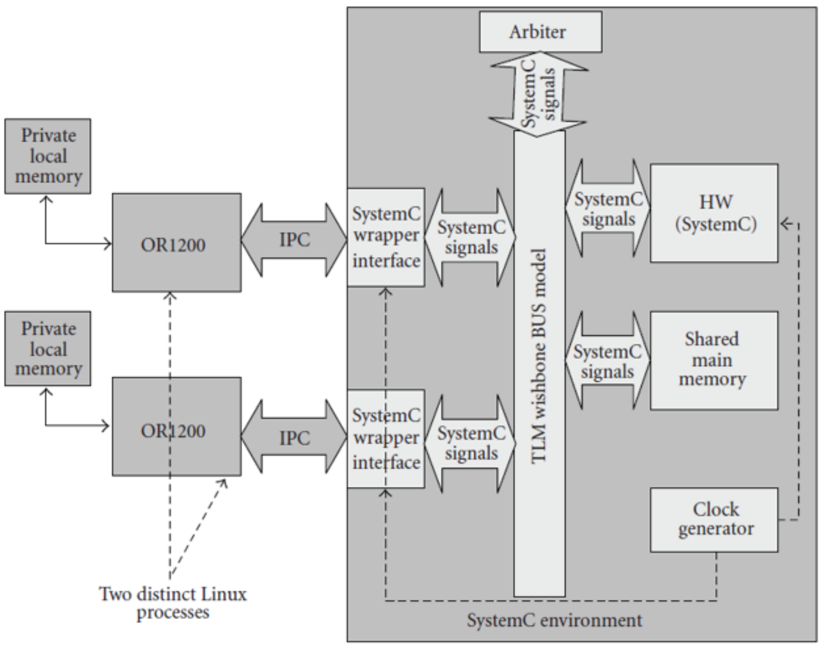


Figure 1.21: *Interface between the two Instruction Set Simulators and the Multi Processor SoC platform.*

Other works have been used the TLM paradigm to evaluate the performances of a Network on Chip (NoC) with different routing algorithms running on it. In this case Escobar et al. [4] used TLM to early validate the network before integrating it with other devices, thus saving time and effort resources; therefore, they used the high level description which turned out to be also useful for HW/SW co-design and fast simulation and proved to be the best choice for creating complex systems.

Different design and simulation tools were developed for academic and industrial purposes, like QEMU-SystemC emulation framework exploited by Yeh's working team [8], connected with the involved SystemC virtual platform through TLM2.0 interface and capable to evaluate its performance, as well as other statistic parameter, while an OS is running; showing the possibility to simulate the designs at the instruction level with external tools through TLM.

As previously described, the main goal that brought TLM paradigm to be the step forward for complex systems design flow is the modelling speed-up; and it has been reached by Jerraya et al. [3], who meant to abstract hardware and software interfaces of heterogeneous MPSoCs (Multi Processor SoCs with different kinds of programmable processors) via the use of parallel programming models.

Since neither the general parallel programming that the software mapping tools are efficient for heterogeneous MPSoCs, the HW and SW interfaces should be abstracted as shown in figure for a concurrent developing.

An interesting result of the Jerraya team's work is that the virtual architecture model, when simulated, has a very precise timing estimation that allows hardware exploration.

Anyway, today's embedded systems often include the analog domain and, knowing that SystemC provides the Analog Mixed Signals system-level design extension, connecting analog domain and digital TLM design allows modelling, simulating and validating whole heterogeneous systems before even generating the RTL architecture [9].

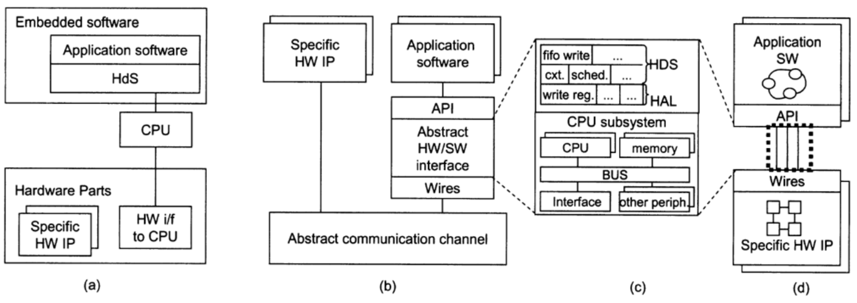


Figure 1.22: *SoC abstraction views: (a) Classic view (b) Abstract HW-SW view (c) Implementation view (d) Functional view.*

In order to give TLM design the same completeness of RTL design, it should be allowed to perform all levels of tests and verifications on both hardware and software. A first approach to fault tolerance requirements verification was attempted by Da Silva et al.[10], who presented a SystemC TLM wrapping library that can be used for the assertion of system properties, protocol compliance, or fault injection; thanks to the fact that virtual platforms allow to access and modify the internal state of the virtual prototypes, allowing to conduct fault injection and fault tolerance assessment procedures. The working group succeeded to create a generic framework that provides dynamic binary instrumentation to TLM2.0 models, able to insert wrappers into the transaction path which allows validating third party TLM cores and modules.

Although TLM paradigm leads to all optimizations described, in any case RTL implementation is needed to proceed toward the gate level synthesis, so if the code cannot be synthesised into RTL it cannot be reused and the RTL model should be described from scratch. As far as it is known, a manual approach could be adopted in TLM to RTL synthesis, but this leads to time waste and error insertion; but thanks to different research groups, TLM HLS (High Level Synthesis) is possible by automated or semi-automated methodologies. A work carried out by Bombieri's team [11] applied the high-level synthesis (HLS) theory to TLM, adopting a protocol synthesis procedure based on the extended finite state machine (EFSM) that starts from TLM functionalities, creates a first cycle accurate implementation before mapping the final design compliantly to the target bus protocol. The working group has successfully implemented an automatic synthesis methodology, shown also in figure 1.23, and they guarantee the correct mapping from TLM transactions to RTL bus transfers, with an RTL interface compliant with the bus protocol.

Just as the TLM code reuse for RTL architecture synthesis reduces design time of complex systems, also the validation effort could be shrunk in case TLM level tests could be translated into RTL validation tests, because different RTL test can be extracted instead of being written for the relative SoCs.

This was the aim of Chem et al. [12] that succeeded on automatically extracting RTL tests from SystemC TLM specifications thanks to a transition-based coverage metric. The methodology starts from the TLM specifications which are then formalized so that TLM tests for the system can be generated; finally, transformation rules are applied to transform TLM tests into RTL tests. Many enhancements can be appreciated from this approach to RTL test generation, other than the ones described yet: system level requirements, difficult to extract starting from RTL level, are intrinsically contained in TLM tests, besides the RTL test can be generated before the RTL model, allowing to be modified compliantly to the low-level validation goals.

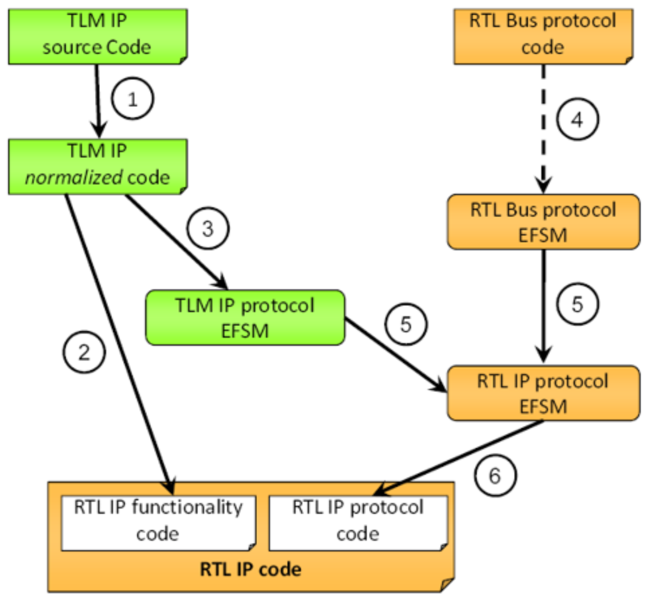


Figure 1.23: *Flow diagram of the TLM to RTL automatic synthesis algorithm.*

1.4 Introduction of the design to test and TLM versioning

The main goal of the thesis work is to prove, over all enhancements exposed in this section, the simulation speed-up of SystemC TLM models with respect to the equivalent RTL model, whether this is a behavioural or structural implementation.

In specific, a CAN controller device is described in both hardware description languages and, regarding the SystemC TLM language, the TLM1 version is chosen rather than 2 because the features and the enhancements bring by the latest version, as the capability of choosing the coding style or the module interoperability or the generic payload usage, are not a discriminant for the simulation performance gap, even if they contribute to increase more the simulation speed. In addition, TLM1 enables the early design of custom bus-based designs and the bridge with pure RTL modules (written in SystemC) is simpler than in the case of TLM2. The specific interfaces towards the bus and the clock-triggered functions suggest that a loosely-timed TLM2 model cannot be implemented in the full architecture, thus all modules that have to work synchronously with the CAN bus should be coded at cycle-accuracy. Tough, the master module driving all peripheral functions can be implemented as an abstract module, that should give a speed boost to the simulation performance. The use of sockets for intercommunication by generic payload is not needed and the communication with exports and TLM channels is sufficient, as well as the direct memory interface will not be exploited. For all the reasons described above, the compilation overhead of the TLM2 library can be avoided by including only the TLM1 standard.

Chapter 2

Simulation of a real case application: TLM design against VHDL benchmark

In this chapter it will be described the development of a CAN controller, implemented in SystemC by TLM1.0 standard; successively it will be introduced the benchmark, corresponding in this case to a VHDL implementation of the CAN controller specifically created to obtain the same behaviour and functionalities of the TLM version. After that, the TLM architecture is compared with the benchmark in terms of simulation performances, thanks to a simulation tool that can simulate VHDL designs as well as SystemC ones.

2.1 The TLM CAN Controller

The CAN controller is designed to be a device implemented as a peripheral of a microcontroller system, able to receive control signals and data from a logic part, like a core processor, and to transmit back the received data from CAN bus. The TLM structure follows the TLM1 design rules detailed in chapter 1; it is composed by a master module, implemented as an abstract module and connected with three slaves by means of a router, which implements address mapping of the slaves. Two slaves are implemented as pure TLM modules, while the module which is interfaced with the CAN bus (the transceiver driver) is designed at RTL; thus the transactor is needed to act as a bridge to interconnect the RTL designed module to the TLM structure.

The slaves implemented in the controller are:

- the CRC codec, which encode the CRC for every frame to be transmitted and checks the coherency between CRC and the remaining bits for each received frame;
- the transceiver transactor, connected to a TLM channel as shown in figure 2.1, that processes the TLM transactions and transforms them into RTL signals and vice versa by running an EFSM;
- the CAN clock generator, naming the module providing the clock for the can bus, that is the internal clock scaled in frequency and, If needed, resynchronised during reception phases.

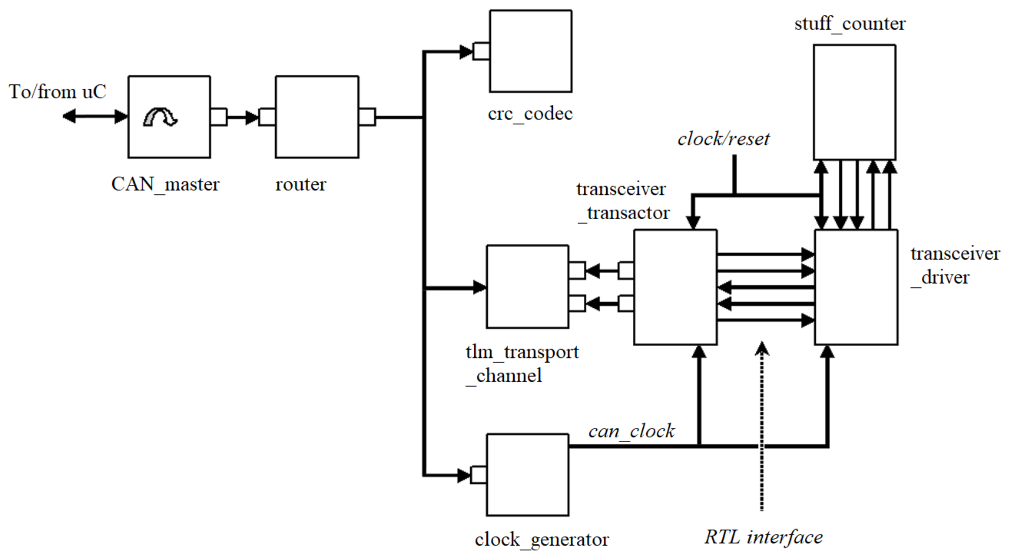


Figure 2.1: *Diagram of the CAN controller modular architecture described with TLM.*

The transactor is in turn connected to the cycle-accurate transceiver driver, which runs processes triggered by the internal clock and the CAN clock in order to correctly perform the protocol functionalities. At last, the stuff counter module is directly connected to the transceiver driver and signals when the stuffing/unstuffing should be performed during a frame transmission or a frame reception.

The management of frames coming from the uC core or from bus is possible providing the master 2 FIFO registers, for temporary storing all fields of each frame. Going into details, when a frame is sent from uC core to the controller, a signal is raised, the frame is stored into a TX_FIFO waiting to be transmitted to bus; in opposite direction, a frame read from bus is first checked for errors and, in case it passes all checks, it is stored into a RX_FIFO before being sent to the uC core. If a FIFO is full, the write is not performed, and a signal notifies the status to the core by a specific signal.

In order to better understand how the message coming from the input interfaces are managed and transmitted as frames, we should go into details of the internal architectures of each module starting from *CAN_master*.

This module contains different ports which are interfaced with the device which sends the messages to the CAN controller. The input ports are: *control_word*, a 10-bit port which receives the bus clock prescaler (but could be used for different controls in future developments); *write_int*, signalling that a message is ready at the *frame_in* port and can be processed. The output ports are: *read_int*, rising every time a message has been extracted from a received frame and it is ready at *frame_out*, *overload_int*, signalling that the FIFO where message to be transmitted is full, and the set of error flags.

There are four different threads running on *CAN_master* committed to write and read messages to/from logic, but also to transmit and receive frames on the CAN bus.

In particular, one thread reads the messages from *frame_in* port and stores them in the *TX_FIFO*, other than signalling a possible overload of the FIFO; the second thread reads the *RX_FIFO* and, if not empty, it writes the extracted messages on *frame_out* port while rising the interrupt.

The pseudo-codes in the following represent the processes described above:

```
receive_uc(){
    OVERLOAD = 0
    while loop
        if (write_int = 1)
            read frame_in
            if TX_FIFO.full = 1
                OVERLOAD = 1
            else
                push(TX_FIFO, frame_in)
            end if
        end if
    end loop
}

transmit_uc(){
    read_int = 0
    while loop
        if RX_FIFO.empty = 0
            frame_out = read RX_FIFO
            read_int = 1
            delay quantum
            read_int = 0
        end if
    end loop
}
```

The remaining two threads, instead, manage the frame transmission and reception through the CAN network alternating the reception and the transmission thanks to a Mutex (mutual exclusion) mechanism, outlined in figure 2.2 , meaning that alternatively only one procedure between reception and transmission is executed. This is made up by two Boolean variables, **TRANSMIT** and **RECEIVE**, which are true one at a time and, other than synchronising the processes, they signal whether a transmission or a reception are going to be pursued.

The body of *receive_bus()* thread is structured as follows: if no transmission is being executed (**TRANSMIT** = FALSE) a read transaction is sent to the transceiver transactor module know if a SOF bit has been detected on bus and, if that happens, the master sends other read transactions to get the frame received from CAN bus, subsequently the frame is sent to the *CRC_codec* module in order to check for errors; then, if CRC check is passed, all frame's fields are extracted following the format rules (different between standard and extended format) and then stored in the specific FIFO, ready to be sent to the processor.

Specularly, the thread *transmit_bus()* reads a new message from the relative FIFO, packets the message into the frame format, so the CRC code can be created and, if no reception is happening yet, the frame can be written to the transceiver transactor in order to start the transmission. In the following, the arbitration bit is read back from transceiver transactor module: if arbitration is won, the transmission continues, otherwise the reception procedure starts, similarly to the *receive_bus()* thread.

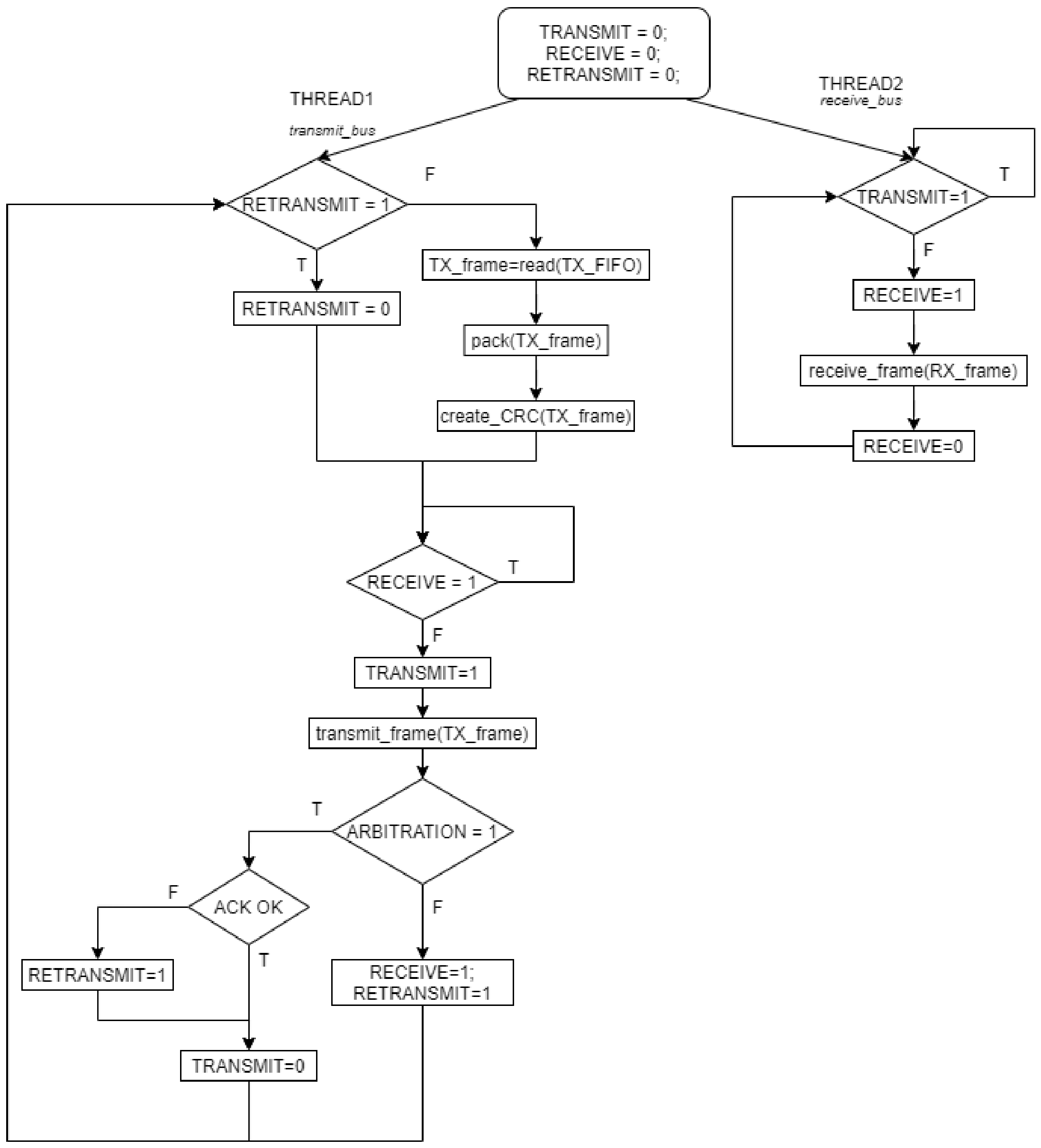


Figure 2.2: *MuTex logic running on master module.*

The peculiarity of the master module is its abstraction, indeed it simply sends transactions to the slaves through the export and there is no implementation detail of the methods governing the transmission or the reception that must follow the CAN protocol.

The module called *clock_generator* is a slave implemented with TLM with the purpose of modelling the clock signal synchronous to the CAN bus clock and thus to all nodes of the CAN network. The clock is generated from the internal clock, but the clock period depends also by the prescaler value read at the control port; moreover, the module is responsible of the CAN clock synchronisation every time a transition is detected from CAN bus.

An important effort has been made to model the transceiver driver because, as known, it is not described using TLM and it is implemented at the lower level of RTL detail; due to this, the transceiver transactor is needed to connect the TLM architecture with the driver. The transactor is connected to the master through a TLM channel (as the TLM1 standard example shown in figure 1.7) thanks to the proper TLM interface ports, on the other side it contains the input and output ports allowing the connection with the transceiver driver's signals.

The module inspects the incoming transactions and it manages the signals provided to the driver through an FSM, which state diagram is shown in figure 2.3. In particular, at the EXECUTE state the process reads a new request (if present) other than switching some output signals in function of the type and address of the request; while at the **WAIT_RESPONSE** state, the done signals coming from the *transceiver_driver* are read because they signal if any transmission/reception phase has finished, in this case a response can be transmitted back on the TLM bus and other requests can be processed; at last, the WAIT status deals with failed response writes.

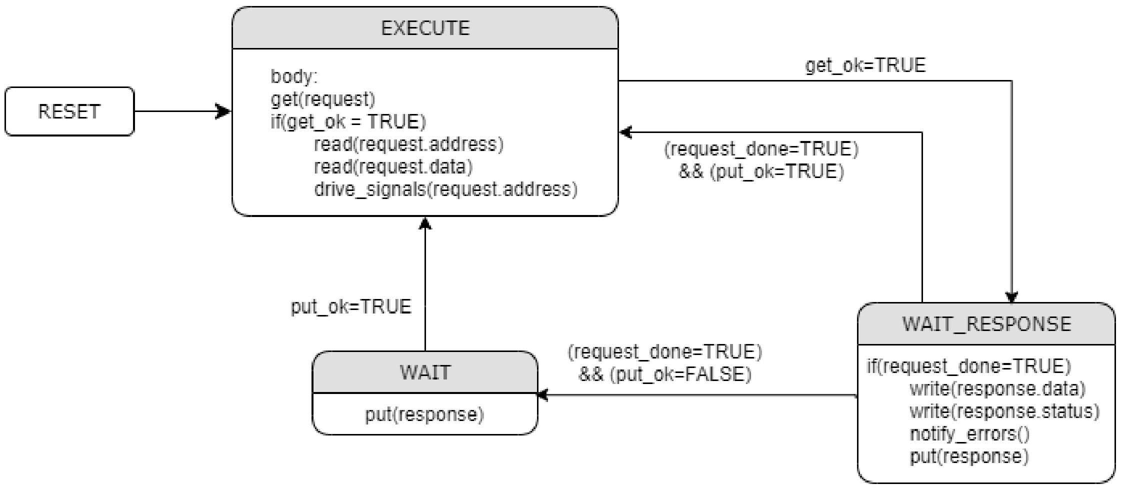


Figure 2.3: *State Diagram of the Transceiver_Transactor FSM.*

In simultaneous with the transceiver transactor FSM, the process running on the transceiver driver performs the CAN protocol at low level as follows:

- if *start_TX* signal is active, the serial transmission of *TX_frame* starts by writing on CAN bus with arbitration; thus, at the end of arbitration the relative flag is written on output port and then either there is a transmission

proceeding or a frame reception;
- if there is no transmission and *start_RX* is active, meaning that a SOF has been detected from bus previously, a frame is read serially from CAN bus and sent to the transceiver through the output ports, in order to be handled by the master;
- if there is no transmission nor reception running and *sof_rx* is active, the CAN bus is read to check for a SOF bit, which result is sent back through port;

The behaviour of the transceiver driver is described with the following pseudo-code:

```
while loop
    if start_transmission = TRUE
        read frame_TX
        transmit_with_arb(frame_TX, frame_RX)
        write transmit_arb_finished

        if arbitration_won = TRUE
            transmit_continue(frame_TX)
            write transmission_finished
        else
            receive(frame_RX)
            write reception_finished
        end if

    else if polling_SOF = TRUE
        receive_SOF
        write SOF_flag

    else if start_reception = TRUE
        receive(frame_RX)
        write reception_finished
    end if
end loop
```

As known, the clock is integrated in the data signal and therefore the bit stuffing technique must be implemented during transmission to guarantee the correct resynchronisation of every node of the CAN. The module supporting the bit stuffing/destuffing to the transceiver driver is the stuff counter; it is implemented as a TLM slave that, during a frame transmission or a frame reception on the transceiver driver, rises the stuff signal to notify that a stuff bit should be written on bus (during a frame transmission) or a stuff bit is read from bus (during a frame reception).

2.2 Developing the benchmark: VHDL version of CAN controller

The simulation performance comparison, that is meant to be carried out between the TLM-developed device and the benchmark described with VHDL language, is considered reliable only if the two models have identical functional behaviours and their architectures are comparable; therefore, the benchmark should follow the overall design structure of the TLM device.

Different open-source CAN controller devices are made available for academic or development use, but unfortunately the structures and the functional behaviours are by far different cause they have, for instance, higher complexity, different processes managing the protocol and dissimilar data structures involved. For this reason, it has been decided to create the benchmark from scratch, so that it can be modelled in the most similar way to the considered device.

The configuration is pretty the same: the core module, having connections with all other modules and running an EFSM to have a complete control on the system, a transceiver driver, a CRC codec, a stuff counter and a clock generator; but in the case of an RTL description there is no abstraction level to be implemented, so the modules are connected by low level signals or signal arrays, moreover the *Transceiver_Driver* EFSM is collapsed into the main EFSM and the *ctrl_proc* is not implemented in the benchmark. Nevertheless, the complexity of the architecture is the same as the TLM version of the device, as will be described below.

The device's main module is the *Stream_processor* that corresponds to the logic unit of the architecture, this component contains an EFSM which manages both the transmission and the reception of the CAN frames, in addition it manages the interfacing with all other modules of the device thanks to discrete control and data signals, it deals with packing the incoming messages to be transmitted and with unpacking compliantly the read frames to extract the messages.

As mentioned, the main component of the *Stream_processor* is the EFSM which includes two parallel interactive FSMs and they are developed such that the transmission and reception are never simultaneously performed (the Mutex mechanism), in accordance with the CAN protocol.

The transmission status, which FSM is shown below in figure 2.5, corresponds to **PICK_FRAME** when reset, to check if a new message to be transmitted is present on **TX_FIFO**; in that case the CRC of the message is created and then, if no reception from bus is active, the transmission starts in the same way of the TLM implementation. Indeed, if the frame has been transmitted without obstacles, the status returns to **PICK_FRAME**; if instead the node has lost arbitration, the message extracted from the frame read on CAN bus is stored on **RX_FIFO** and a re-transmission should be performed without reading new messages from **TX_FIFO**; at last, also in case the transmission has been performed but there have been some errors a re-transmission of the same message should be done.

On the other side, the reception status starts from the idle state (**WAIT_RX**) and, only if there is no frame transmission, the status switches to **WAITFOR_SOF** to check if a SOF bit is present on bus; then, if this is detected the reception is started, otherwise the status goes back to **WAIT_RX** to check for the next SOF bit. At the end of reception, the status goes to **TRANSMIT_ACK** where communicates if there have been some errors; in case of errors in reception the status switches to **END_RX**, otherwise the device receives the EOF and IFS fields from bus, stores the received message in the **RX_FIFO** and returns to the default status.

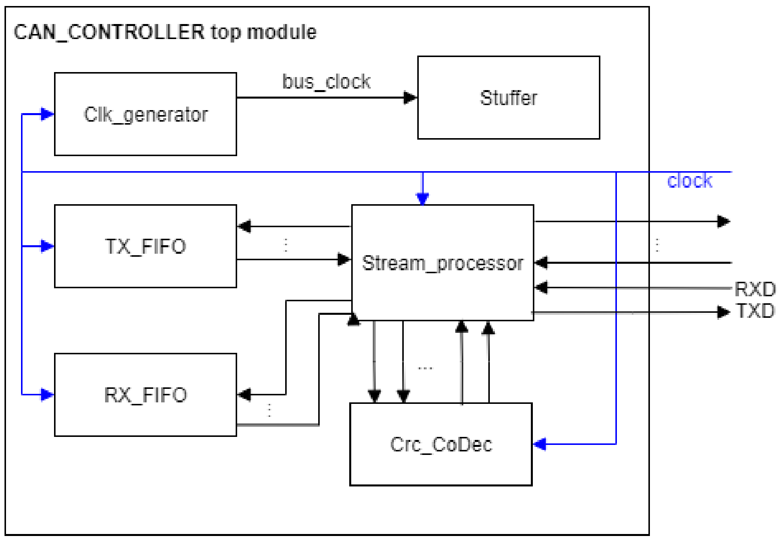


Figure 2.4: *Block diagram of the VHDL benchmark.*

The thread *FRAME_GEN* is responsible for assembling the frame to be transmitted after a new message is picked from **TX_FIFO**, in relation to the length and the frame type, it also concatenates the CRC generated by the module *CRC_Codec*; while the thread *CRC_PROC* generates the correct serial CRC data signal depending on the status of the EFSM, knowing that the *CRC_Codec* is used for CRC key generation for a received frame from bus and for a frame to be transmitted.

As it can be seen from figure 2.4, unlike the TLM device, the VHDL benchmark does not implement the transceiver driver module cause its functionalities are integrated in the *Stream_processor*, in particular the parallel FSMs described above drive the writing of each frames' bits at CANbus_clock's rising edge, based on the arbitration status, the stuffing and the possible transmission errors, on the other side they perform the bit reading at the falling edge and check for any errors and incoming signals.

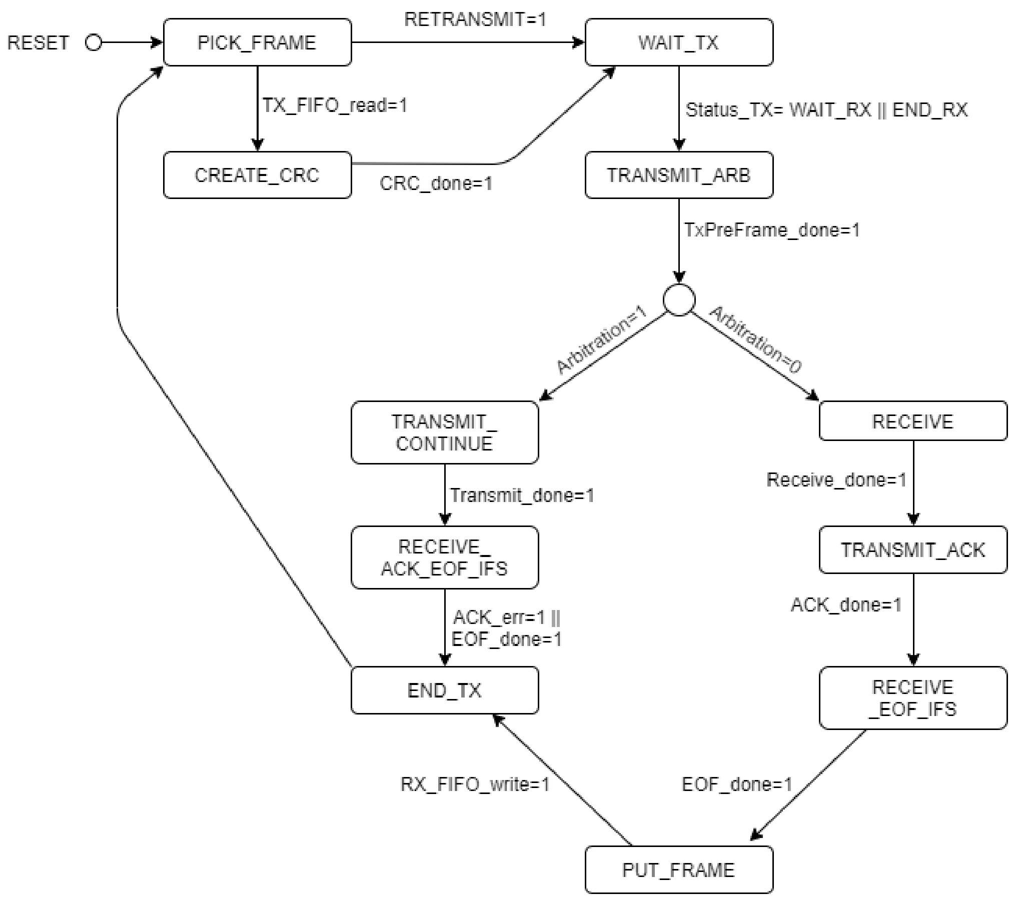


Figure 2.5: *State diagram of the benchmark transmission FSM.*

The clock generator is a simple module, generating a clock signal for the CAN bus from the internal clock of the architecture; it is implemented with a counter that scales the period of the internal clock. In this case the scaling factor is taken by the prescaling signal, moreover the counter can be re-synchronized when an edge on the bus is detected.

The *CRC_Codec* module is realized with a shift register and XOR gates cause the input data is received serially. The register is reset during the initialization phase, then the CRC generation can be pursued on the incoming data by performing sequential left-shifts and eventually modulo-2 subtractions, in function of the application specific CRC key. As known, it is needed in both CRC code generation and CRC check, allowing detection of single and double bit errors.

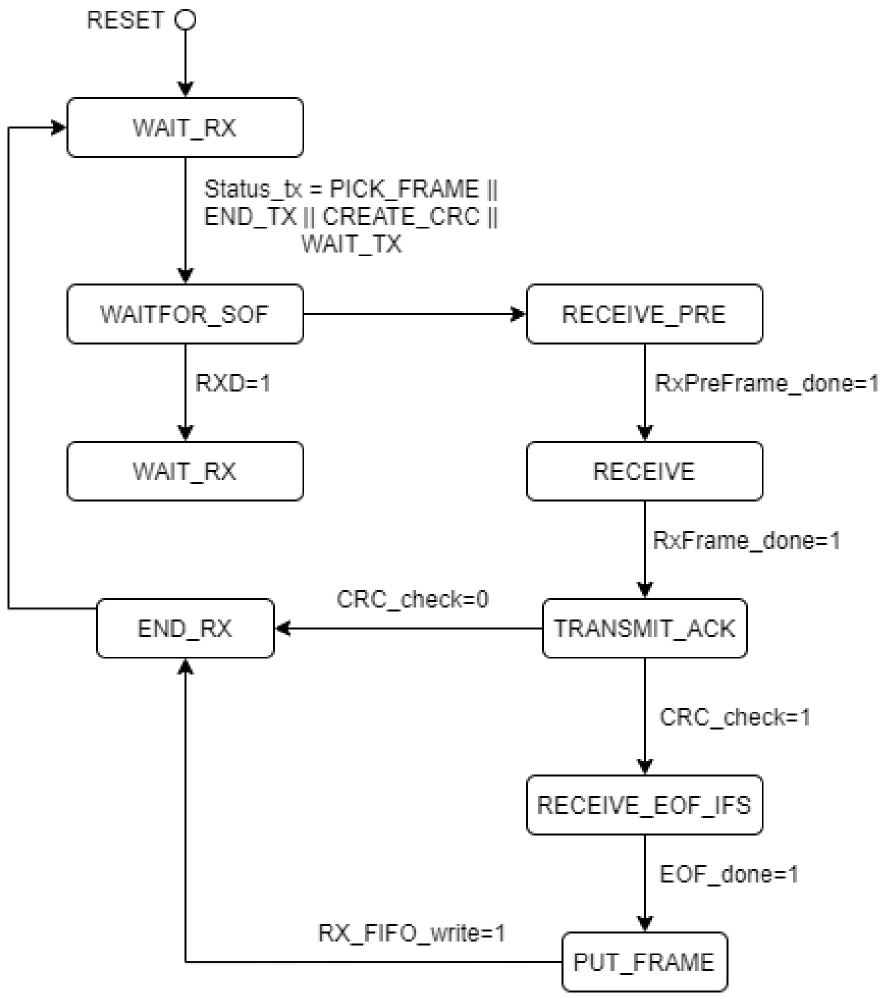


Figure 2.6: *State diagram of the benchmark reception FSM.*

2.3 Simulations in comparison

Based on the evident similarity between the architecture of the two models, that entails a valid performance analysis between simulations without affecting the final comparison, it is necessary to make their testbenches equivalent, so it has been chosen to include the same HW components, configure them equally and provide the same test stimuli step by step.

Therefore, the test environment of both the DUT and the benchmark are structured as follows: there are two CAN nodes exchanging frame transfers through the CAN bus, then a FIFO is instantiated downstream the receiver node, in order to store the read frames. The test first step is a global reset, afterwards the bus clock synchronization is performed and subsequently a series of frames are sent to the node 1 so they can be written on the bus and received by node 2. The test is run for 100 ms in terms of simulation time, while the real time delay is measured to evaluate the simulator performances.

The simulations have been run on the web application EDA Playground, which allowed us to run VHDL simulation with "Aldec Riviera Pro 2020.4" and SystemC simulation with "Accelera SystemC 2.2 compiler", which are tools mounted on the same server, therefore a reliable performance comparison has been carried out. Moreover, the VHDL benchmark has been simulated on ModelSim to trace the signals of interest involved during the different phases of the frame transmission and reception, essential for debugging.

2.3.1 TLM design simulation

In this section, the outputs generated by the simulator terminal will be extracted and analyzed in order to have a better comprehension of the device behaviour during the test case. The TLM design has been generated thanks to an embedded C++ compiler which includes SystemC 2.2 libraries and, in order to exploit the necessary TLM paradigm software architecture, all TLM libraries from the OSCI TLM1.0 standard release are compiled along with the design.

As follows is shown the printing from the internal modules of the DUT, other than the testbench environment, that allows to see in detail the dataflow in both the transmitter node and the receiver node during a frame transaction.

```
Finished binding
Clock generator: Changing prescaler value to 0 at 0 s
TB: Starting initialization
TB: Initialization finished
TB: Starting CAN bus clock synchronization
Clock generator: Changing prescaler value to 40 at 10 ns
Clock generator: Changing prescaler value to 40 at 10 ns
TB: Synchronization finished
TB: reset disabled, simulation started
TB: Sending to DUT a 5 byte message with RTR = 0, IDE = 1 @ 3220 ns
CAN_master_1: Starting CRC encoding
crc_codec_1: CRC = 59338, frame = 13298584346136117472146 2272
CAN_master_1: CRC done
TB: Sending to DUT a 2 byte message with RTR = 1, IDE = 0 @ 3230 ns
TB: Sending to DUT a 3 byte message with RTR = 0, IDE = 0 @ 3240 ns
CAN_master_1: Start transmission of frame: ID = 13, RTR = 0, IDE = 1,
    ID_EXT = 434, DATA = 5210
```

In the first phase, all modules are instantiated and the signals are bound, then the test environment starts the initialization: the DUT is reset, the selected baud rate is sent to the nodes and the synchronization of the bus clock with all nodes is performed. As it can be read, the simulation starts when reset is disabled and the first message is sent to Node 1, which generate the corresponding frame and launch the transmission to the bus. In particular, the master drives the *CRC_codec* and the *Transceiver_transactor* modules to proceed with the Crc encoding and subsequently with the coherent collision-less serial transmission on bus.

```
TXRX: SOF detected at 6200 ns
CAN_master_2: SOF detected, start receiving
TXRX: STARTING RECEIVING at 6300 ns
TXRX: Pre-frame read 1845549317 , it is expected a 5 byte long data
    from node 434
TXRX Trans: Reading FrameH at 88280 ns
TXRX: Frame 12682518335472218963 7 received at 170200 ns
TXRX Trans: Reading FrameL at 170280 ns
CAN_master_2: Frame received, starting CRC check
CAN_master_2: Sending ACK at 170320 ns
Master: End receiving
TB: Received data form CAN master 58255736832
```

As it can be noticed from the output shown above, at 6,2*ns*, the Start Of Frame bit is sensed from node 2, thus it receives the frame through the *Transceiver_driver* and, when the last CRC bit is caught, the frame is sent back to the master, the *CRC_encoder* performs the error check and then the acknowledge bit is written on bus by Node 2.

The received payload, with the metadata extracted from frame SDU, is then written on FIFO and read from testbench, as printed at the output.

You can observe from the following report that the simulation proceed with the messages sent from the testbench to Node1 which continue transmitting on bus without need of arbitration, cause it is the only master node on the CAN network under test. The test ends after 100*ms* of simulation time without errors on bus neither on the message contents, validating the design.

```
TXRX: Sending on bus ack = 0 at 171200 ns
TXRX: SOF detected at 202200 ns
CAN_master_2: SOF detected, start receiving
TXRX: STARTING RECEIVING at 202300 ns
TXRX: ACK = 1, transmitting EOF and IFS
TXRX: The bus is IDLE again at 242200 ns
TXRX Trans: ERROR in transmission occurred
TB ERROR: TX error = 1, RX error = 0 at 242240 ns
TXRX Trans: Reading arbitration flag = 1 at 242280 ns
CAN_master_1: arbitration won
CAN_master_1: ACK not received, the frame will be retransmitted
CAN_master_1: End of transmission
CAN_master_1: Start transmission of frame: ID = 13, RTR = 0, IDE = 1,
     ID_EXT = 434, DATA = 5210
TXRX: Pre-frame read 369366142400 , it is expected a data request
    from node 27
TXRX: Frame 25768758720 received at 288240 ns
TXRX Trans: Reading FrameH at 288280 ns
TXRX Trans: Reading FrameL at 288400 ns
TXRX: Sending on bus ack = 0 at 289200 ns
```

```
TXRX: ACK = 1, transmitting EOF and IFS
TXRX: The bus is IDLE again at 482200 ns
TXRX Trans: Reading arbitration flag = 1 at 482280 ns
CAN_master_1: arbitration won
CAN_master_1: ACK not received, the frame will be retransmitted
CAN_master_1: End of transmission
CAN_master_1: Start transmission of frame: ID = 13, RTR = 0, IDE = 1,
    ID_EXT = 434, DATA = 5210
TXRX: ACK = 1, transmitting EOF and IFS
TXRX: The bus is IDLE again at 722200 ns
TXRX Trans: Reading arbitration flag = 1 at 722280 ns
CAN_master_1: arbitration won
CAN_master_1: ACK not received, the frame will be retransmitted
CAN_master_1: End of transmission
CAN_master_1: Start transmission of frame: ID = 13, RTR = 0, IDE = 1,
    ID_EXT = 434, DATA = 5210
TXRX: ACK = 1, transmitting EOF and IFS
TXRX: The bus is IDLE again at 962200 ns
TXRX Trans: Reading arbitration flag = 1 at 962280 ns
CAN_master_1: arbitration won
CAN_master_1: ACK not received, the frame will be retransmitted
CAN_master_1: End of transmission
CAN_master_1: Start transmission of frame: ID = 13, RTR = 0, IDE = 1,
    ID_EXT = 434, DATA = 5210

TB: Stopping simulation at 100 ms
```

2.3.2 VHDL design simulation

In this section will be shown the signal waves during simulation, displayed in Modelsim graphical environment to debug and analyze the behaviour of the VHDL benchmark; in order to have a top view of the internal signals and the input/output interfaces of the device.

Figure 2.7 contains the first set of clock cycles when a message PDU is picked from the *TX_FIFO* and the CRC code is generated in order to arrange the frame to transmit; before starting the coding, the signal *CRC_init* resets the internal FFs of the codec in order to read the new input serial data *CRC_data*, that is provided together with the enable activation. At each cycle the partial result is present as a binary value of *Crc_computed* and the algorithm runs until *Crc_count* reaches the frame's length, after which the enable is deactivated and the output keeps the result value to be joined with the frame PDU.

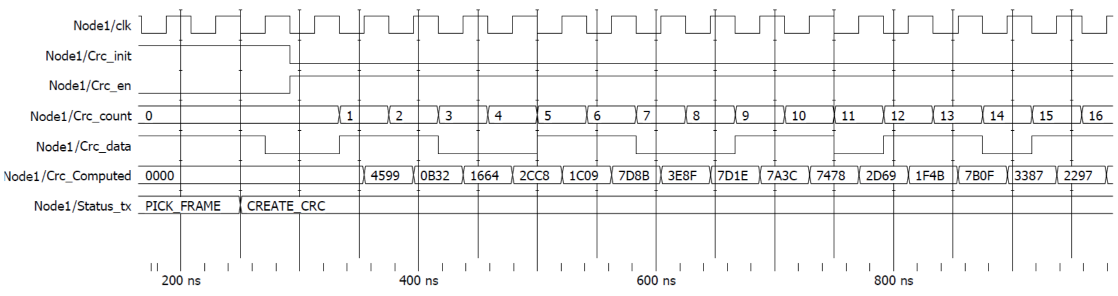


Figure 2.7: *CRC generation, from logic simulation of the benchmark.*

The CRC computation delay depends on the frame length and format, specifically the worst case is encountered when an extended CAN frame with an 8-byte payload is transmitted, since it means that the CRC code is available after 119 clock cycles. The delay, in terms of bus clock cycles, is function of the scaling between the internal clock and the bus clock periods; for instance, during the performed simulations the bus clock period is 40 times long the internal clock period, meaning that the CRC is always generated in less than 3 bus clock ticks. Therefore, all computations performed to obtain the appropriate frame can be done between two consecutive CAN transactions on bus, which keep the bus idle for 14 cycles.

Once the Crc is ready, the device goes in *WAIT_TX* status for on bus cycle, in order to check if a transmission is already ongoing on CAN bus, before starting the transmission. As known, the transmission of the frame's header is lead with the arbitration mechanism and, as it can be verified on figure 2.8, the index *tx_pointer* is used to write sequentially the frame bits, starting from SOF.

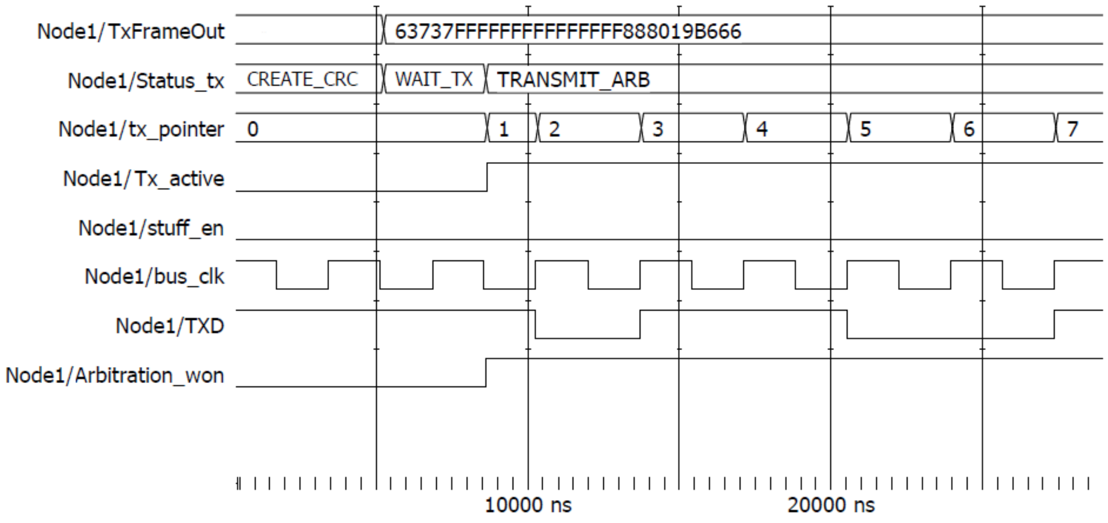


Figure 2.8: *Frame transmission (arbitration phase), from logic simulation of the benchmark.*

Bit write is executed in the rising edge of the bus clock, while bit read is executed on the falling one, therefore the flag *Arbitration_won* is updated on the latter one; while the control signal *stuff_en* is needed to block the index's increase when the stuffing bit has to be written on bus. The simulation does not encounter any collision between transmitting nodes, thus the header transmission is completed and the whole frame is sent from node 1.

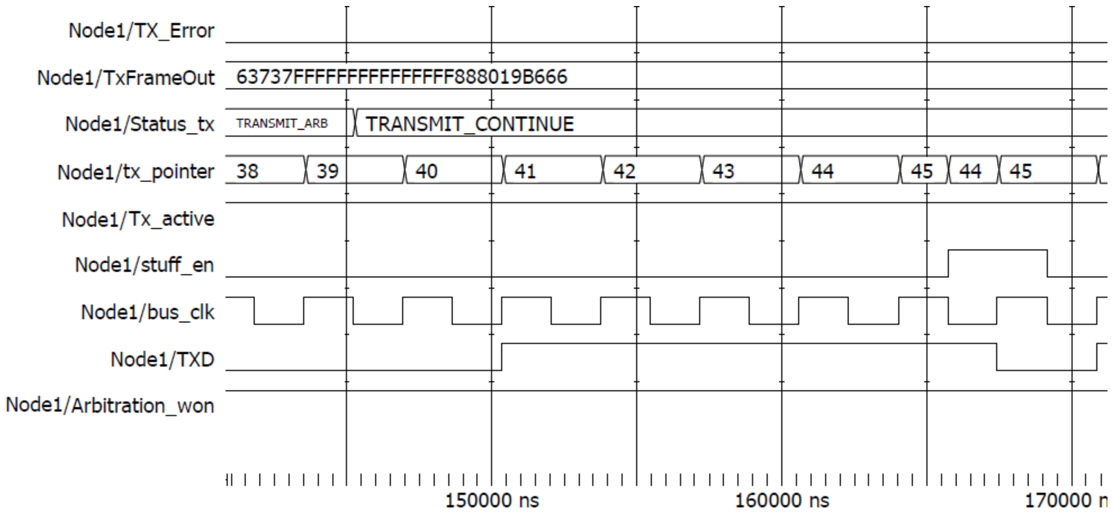


Figure 2.9: *Frame transmission (after arbitration) from logic simulation of the benchmark.*

As shown in figure 2.9, in the test case there is no conflict on bus, thus when it's turn to transmit the payload, the FSM switch to *TRANSMIT_CONTINUE* state in order to turn off the dependence from *arbitration_won* signal. In the same figure can be also appreciated how the stuffing affect the behaviour of the internal signals, in particular way the value of *tx_pointer*. As expected, at the end of transmission the transmitter node waits for the acknowledge and terminate the frame transmission by holding the bus in idle state (transmitting recessive signal) at least for 14 bus clock cycles.

Moving on the receiver (node 2) behaviour, considering the same frame transmission, the concerned group of signals is reported in figure 2.10; it can be noticed that initially the receiver sniffs the bus to sense SOF at the falling edge right after it is sent from the transmitter (node 1), at the same time the Crc codec is keep reset. The figure actually shows how at the second falling edge of *bus_clk*, the node 2 detects the dominant bit on the bus thus its status changes, the Crc codec is enabled to compute the frame during reception and *rx_pointer*, the index needed to sequentially buffer the frame, starts incrementing.

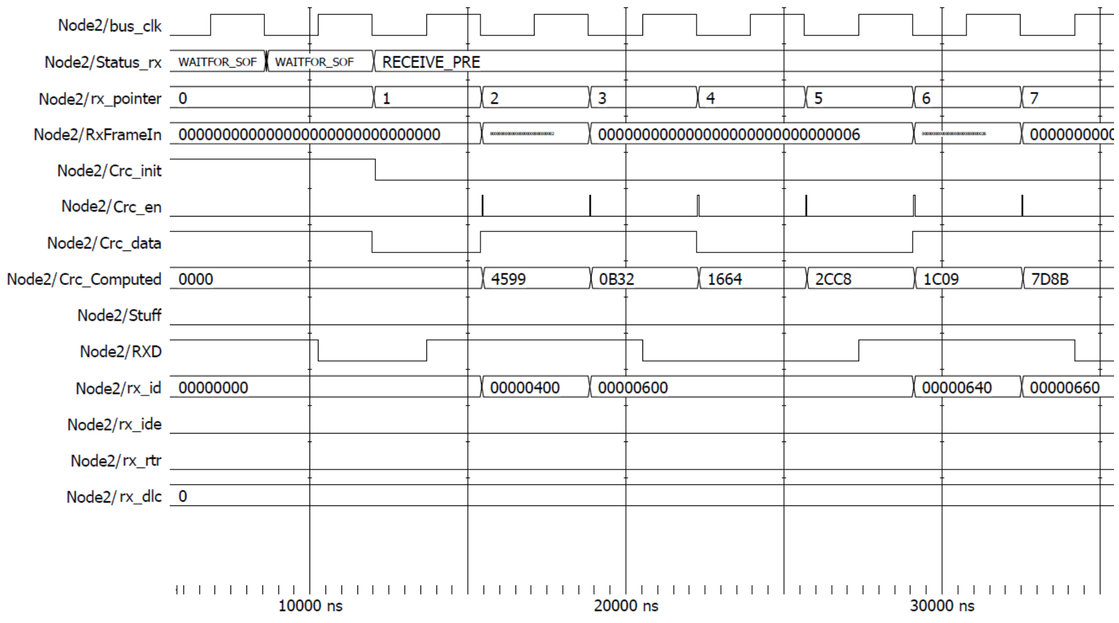


Figure 2.10: *Start of frame reception, from logic simulation of the benchmark.*

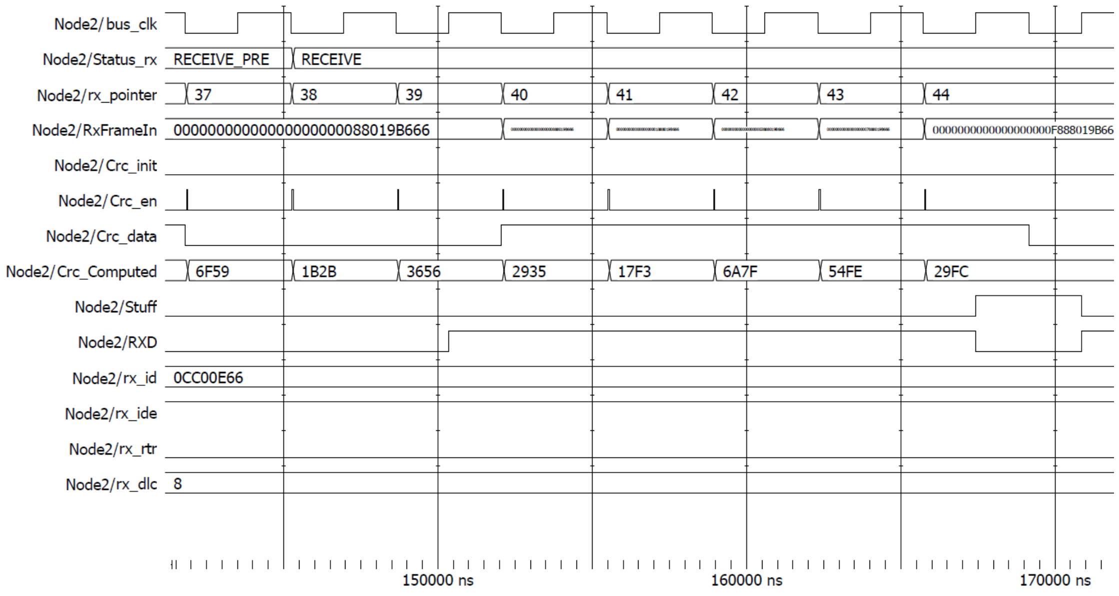


Figure 2.11: *Frame reception phase from logic simulation of the benchmark.*

The reception phase is split in two because the payload length is defined in the header, which in turn should be correctly decoded based on the frame format (standard or extended); hence firstly the extraction of the header fields is handled, then, if no reception error occurs, the receiver switch to *RECEIVE* status to read the frame PDU.

As it can be seen on figure 2.11, in the considered frame transmission the header reception phase is 39 bits long, thus the frame is in the extended format; at this point the DLC value is extrapolated and the reception continues. The destuffing mechanism can be verified from an internal point of view: when the *stuff_en* flag is raised, the index dies not increment in order to overwrite the stuffing bit, which is neither considered in the Crc computation; indeed the *Crc_en* signal is not raised during the destuffing.

When the transmitted frame's PDU is received, the Crc field extracted from this and the Crc computed by the codec during reception are compared and the *Crc_check* flag is generated (figure 2.12), from which depends if an ACK or a NACK is transmitted on bus. Moreover, if the Crc check fails or a reception error occurred, node 2 transmits a NACK and the read frame is discarded. After the ACK/NACK transmission, the reception is finished and also node 2 waits for the EOF and IFS before transmitting or sniffing again the bus.

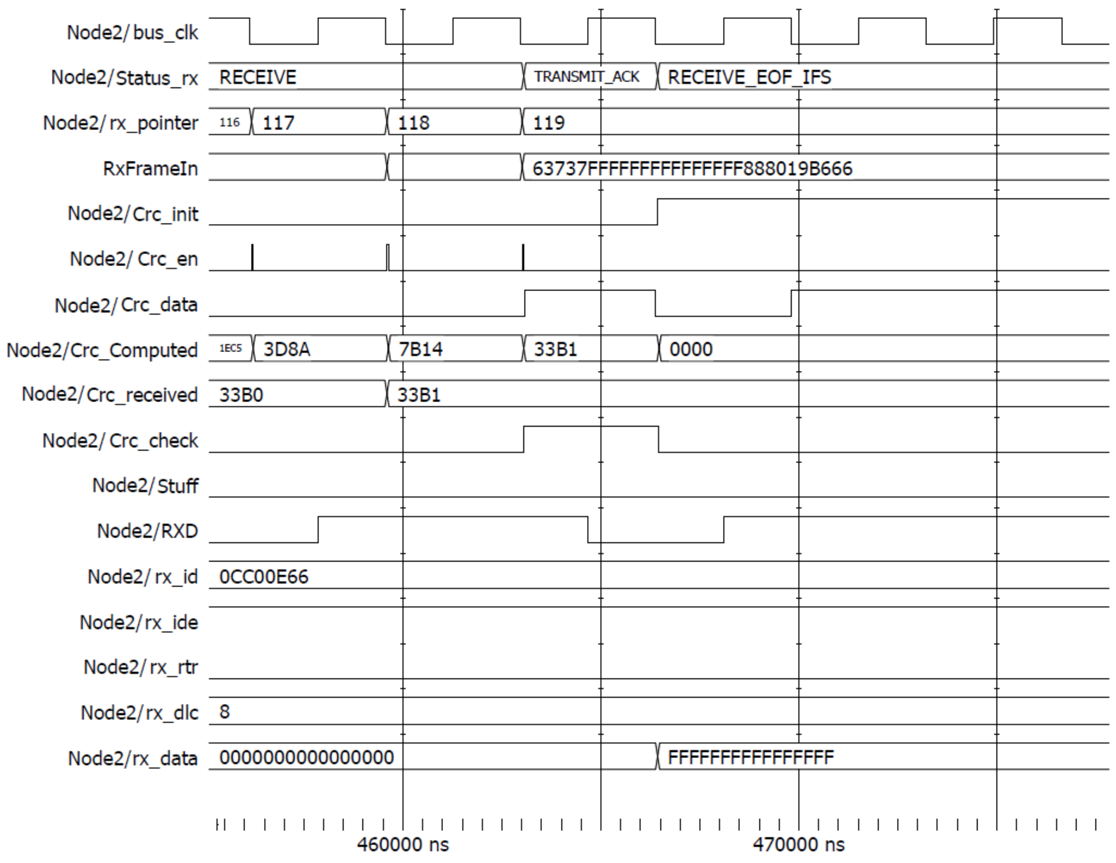


Figure 2.12: *End of frame reception from logic simulation of the benchmark.*

Chapter 3

Conclusions: Result analysis and possible developments

A standalone CAN transceiver, exploitable as a peripheral module compatible with a microcontroller based architecture, has been implemented in a high-level language and compared with a low-level hardware description language.

The comparison results demonstrate a reasonable simulation performance enhancement although the optimization does not reach the most excellent outcome, as discussed in section 1.1.

Indeed, the real time amount employed to run 100ms of VHDL simulation by the "Aldec Riviera Pro 2020.4" simulator is 23sec, whereas the real time employed to run 100ms of SystemC simulation is equal to 1.65sec (both values are an average on 10 repeated simulations).From these results the time ratio results to be equal to 14, thus the latency is decreased by over 10 times; against the values of simulation time improvement in the order of 100 times for cycle accurate models.

The dissimilarity has been analyzed and several design options are identified as starting points for a different implementation that bring to further optimizations. In particular, every process of the developed TLM modules has been implemented as an *SC_THREAD*, meaning that the use of different *wait()* statements rather slower the simulation because the dynamic sensitivity list is triggered in threads; instead the use of *SC_METHOD*s and static sensitivity lists lighten the real time emulator job.

A further consideration to take into account is that the VHDL benchmark has been developed with a behavioural implementation, that brings to a faster functional simulation with respect to a structural implementation of each component.

Considering as well that the standard used to develop the device is the TLM1.0, an additional margin of improvement can be introduced by adopting the new TLM2.0 standard which allows the use of the *loosely timed modeling*, known as the coding stile used for fast simulations.

In conclusion, this comparison clearly reveals that the high level description approach yields significant results and show the way forward to an innovative hardware and software modeling methodology that influence the development process of complex systems.

References

[1] James Aldis Bart Vanthournout and the TLM OSCI Working Group. *OSCI TLM-2.0 User Manual.* Tech. rep. June 2008, p. 151 (cit. on pp. 3, 4).

[2] Sami Boukhechem and El-Bay Bourennane. «SystemC Transaction-Level Modeling of an MPSoC PlatformBased on an Open Source ISS by Using Interprocess Communication». In: *International Journal of Reconfigurable Computing* 2008 (Aug. 2008), p. 10. DOI: 10.1155/2008/902653 (cit. on pp. 3, 22).

[3] Aimen Bouchhima Ahmed A. Jerraya and Frederic Petrot. «Programming models and HW-SW InterFaces Abstraction for Multi-Processor SoC». In: *Proc. ACM/IEEE 43rd Design Automation Conference.* California, USA, July 2006, p. 6. DOI: 10.1109/DAC.2006.229246 (cit. on pp. 3, 23).

[4] Lorena Garcia Posada Fernando A. Escobar Mauricio Guerrero Hurtado and Antonio Garcia Rozo. «Performance Evaluation of a Network on a Chip Router Using SystemC and TLM 2.0». In: *IEEE* 2011 (Apr. 2011), p. 4. DOI: 10.1109/LASCAS.2011.5750309 (cit. on pp. 3, 23).

[5] Pablo Parra Sebastián Sánchez Prieto Oscar R. Polo and Antonio da Silva. «HW/SW Co-design of the Instrument Control Unit for the Energetic Particle Detector on-board Solar Orbiter». In: *Advances in Space Research* 2013 (Mar. 2013). DOI: 10.1016/j.asr.2013.05.029 (cit. on p. 3).

[6] John Pierce Adam Rose Stuart Swan and Jean-Michel Fernandez. *Transaction Level Modeling in SystemC.* Tech. rep. Apr. 2005, p. 15 (cit. on p. 4).

[7] Marcelo Montoreano. *Transaction Level Modeling using OSCI TLM 2.0.* Tech. rep. Mar. 2007, p. 5 (cit. on p. 4).

[8] Zin-Yuan Lin Tse-Chen Yeh and Ming-Chao Chiang. «Enabling TLM-2.0 Interface on QEMU and SystemC-based Virtual Platform». In: *IEEE* 2011 (May 2011), p. 4. DOI: 10.1109/ICICDT.2011.5783207 (cit. on p. 23).

[9] Christoph Grimm Markus Damm and Jan Haase. «Connecting SystemC-AMS Models with OSCI TLM 2.0 Models Using Temporal Decoupling». In: *Forum on Specification, Verification and Design Languages*. Stuttgart, Germany, Sept. 2008, p. 6. DOI: `10.1109/FDL.2008.4641416` (cit. on p. 23).

[10] Oscar R. Polo Antonio da Silva Pablo Parra and Sebastián Sánchez Prieto. «Runtime Instrumentation of SystemC/TLM2 Interfaces for Fault Tolerance Requirements Verification in Software Cosimulation». In: *Hindawi Publishing Corporation* 2014 (Sept. 2014), p. 15. DOI: `10.1155/2014/105051` (cit. on p. 24).

[11] Franco Fummi Nicola Bombieri and Valerio Guarnieri. «Automatic Synthesis of OSCI TLM-2.0 Models into RTL Bus-based IPs». In: *Proc. ACM/IEEE International High Level Design Validation and est Workshop (HLDVT)*. Anaheim, Florida, USA, June 2010, p. 8. DOI: `10.1109/HLDVT.2010.5496652` (cit. on p. 24).

[12] Prabhat Mishra Mingsong Chen and Dhrubajyoti Kalita. «Towards RTL test generation from SystemC TLM specifications». In: *Proc. ACM/IEEE International High Level Design Validation and est Workshop*. Irvine, California, USA, Nov. 2007, p. 7. DOI: `10.1109/HLDVT.2007.4392793` (cit. on p. 24).