CLAN - A CAN 2.0B Protocol Controller for Research Purposes

Arnaldo S. R. Oliveira, Nelson L. Arqueiro, Pedro N. Fonseca

Abstract – **The CLAN intellectual property core is a CAN 2.0B controller developed at the Electronics and Telecommunications Department of the University of Aveiro, for research and educational purposes and in particular with the aim of providing the adequate hardware support to implement and validate higher layer protocols such as TTCAN or FTT-CAN. It was modelled at RTL level using the VHDL hardware description language, synthesized, implemented and tested on Xilinx FPGAs. However, the model is technology independent and can be synthesized for different implementation technologies from FPGAs to ASICs. The CLAN IP core fully implements the CAN 2.0B specification and it includes also a synchronous parallel microprocessor interface, interrupt generation logic and some advanced features, such as message filtering, single shot transmission and extended error and statistics logs. The data bus width can be 8, 16 or 32 bits wide. For applications where a microprocessor interface is not needed or a different interface is required, the core internal module that implements the protocol can be used separately. The CLAN controller with microprocessor interface logic occupies about 30% of a Xilinx Spartan-IIE XC2S300E FPGA, corresponding to 100,000 equivalent logic gates, approximately. It was tested with other CAN controllers operating at 1Mbit/seg.**

Resumo – **O módulo CLAN é um controlador CAN 2.0B desenvolvido no Departamento de Electrónica e Telecomunicações da Universidade de Aveiro para fins académicos e em particular com o objectivo de conceber um controlador que proporcione o suporte de hardware adequado à implementação de protocolos de alto-nível, tais como o TTCAN ou o FTT-CAN. O controlador CLAN foi modelado ao nível RTL com a linguagem de descrição de hardware VHDL, implementado e testado em FPGAs da Xilinx. No entanto, é importante referir que o modelo é completamente independente da tecnologia podendo ser sintetizado para diferentes tecnologias, desde FPGAs a ASICs. O controlador CLAN implementa completamente a especificação 2.0B do protocolo CAN e inclui também um interface síncrono paralelo para ligação a um microprocessador, circuito gerador de interrupções, filtros de mensagens e vários contadores erros e registos de estatísticas. O barramento de dados pode ser de 8, 16 ou 32 bits. Para aplicações que não necessitem de um interface com processador ou requeiram outro tipo de interface, o bloco interno que implementa o protocolo pode ser usado separadamente. O controlador CLAN ocupa cerca de 30% de uma FPGA Spartan-IIE XC2S300E da Xilinx, correspondendo a cerca de 100.000 portas lógicas equivalentes e foi testado com outros controladores CAN a operar a 1Mbit/seg.**

Keywords – **CAN, TTCAN, FTT-CAN, Protocol controller**

Palavras chave – **CAN, TTCAN, FTT-CAN, Controlador de protocolo**

I. INTRODUCTION

Defined in the late 80's, CAN (Controller Area Network) [1] found wide-spread acceptance in embedded distributed control systems, from automotive to industrial applications. A CAN overview is out of the scope of this paper. The CAN specification available on the web [1] provides a clear description of the protocol.

In spite of its popularity, the application of CAN in safety-critical systems is, nevertheless, impaired by the event-triggered characteristics of the original definition. In CAN, a node can send a message at any time, provided there is silence on the bus (CSMA); the Medium Access Control mechanisms will handle the resulting collisions. As a consequence, a node sending a message has no guarantee in what concerns the delivery time of that message; depending on the message priority, it may loose contention for several consecutive times, thus postponing the effective sending of the message.

For critical applications, time-triggered systems are preferred, due to their scalability, composability and dependability properties [2]. The last few years saw the outcome of some proposals to improve the time characteristics of CAN (e.g., TTCAN [3], FTT-CAN [4]). These take advantage of the fact that Bosch's and ISO specifications define only layers 2 and (partially) 1 of the ISO OSI model. With major or minor changes on the original definition, these new proposals impose some determinism in the message exchange behavior, namely by allowing a node to send its message at well defined instants in time. This is achieved by properly defining mechanisms in the layers above the original definition.

TTCAN (Time-Triggered Communication on CAN) started in ICC'98, the International CAN Conference, where an expert group, including CiA (CAN in Automation), chip providers, users and academia, joined the ISO TC22/SC3/WG1/TF6. The result was ISO 11898-4, part 4 of the ISO 11898 standard, that specifies time triggered communication on CAN [5].

FTT-CAN (Flexible Time-Triggered Communication on CAN) has been proposed at the University of Aveiro as a mean to merge flexibility and timeliness in CAN systems. The aim is to achieve a communication paradigm that allows systems to be both timely, delivering the messages under the specified time constraints, and flexible, by not requiring the message set to be statically defined during system operation.

II. MOTIVATION AND OBJECTIVES

Both proposals for time triggered operation of CAN are built on top of the existing protocol with little or no mod-

ifications (the aim of FTT-CAN is also to provide timely behavior with standard CAN controllers).

The development of a new communication protocol requires its validation. Although simulation and formal validation play an important role here, they are not, on their own, sufficient. The last step in validation is always field tests, and these have to be performed with hardware devices. These tests should also involve the verification that the adopted solution is better than the alternatives. Ideally, we should have a flexible communication controller that can be programmed to follow some specification and that can be modified. Another issue to test is the robustness of the new protocols, mainly in what concerns fault tolerance. To do this, faults have to be introduced in the system in a controlled and predictable way. Again, we meet the need for a controller that we can modify to our desire.

The above requirements cannot be easily fulfilled with the CAN products commercially available [6] because they are hardwired ASIC products or flexible but expensive synthesizable cores. Thus, a CAN controller was developed based on the *CANSim* simulator [7]. The initial requirements for the CLAN project were the following:

- Complete CAN 2.0B implementation;
- Internal status fully visible to effectively support the implementation of higher layer protocols;
- Enhanced logging capabilities (individual error counters and flags) and statistics logs (message counters);
- Flexible message filtering capabilities;
- Customizable interface - parallel/serial, (a)synchronous, (de)multiplexed buses.

III. ARCHITECTURE

The developed CAN controller fully implements the CAN 2.0B specification. The developed Intelectual Property (IP) block was split in two modules, to separate the logic that implements the protocol from the interface:

- The *CLAN Core* module which implements the CAN 2.0B protocol;
- The *CLAN Controller* module which provides a synchronous parallel interface with non-multiplexed buses.

The interface provided is adequate for integration of the *CLAN Controller* with a processor core into a single FPGA or a System-on-Chip.

A. CLAN Core Module

The *CLAN Core* module contains all the circuits required to implement the Medium Access Control (MAC) and the Logical Link Control (LLC) layers of the CAN 2.0B specification. It can be used separately directly connecting to sensors and/or actuators in a CAN node without a microprocessor. Alternatively, it can also be used as a building block to create a controller with a customized interface.

A.1 Interface Ports

The external interface of the *CLAN Core* module is shown on Figure 1. The ports are divided into the following

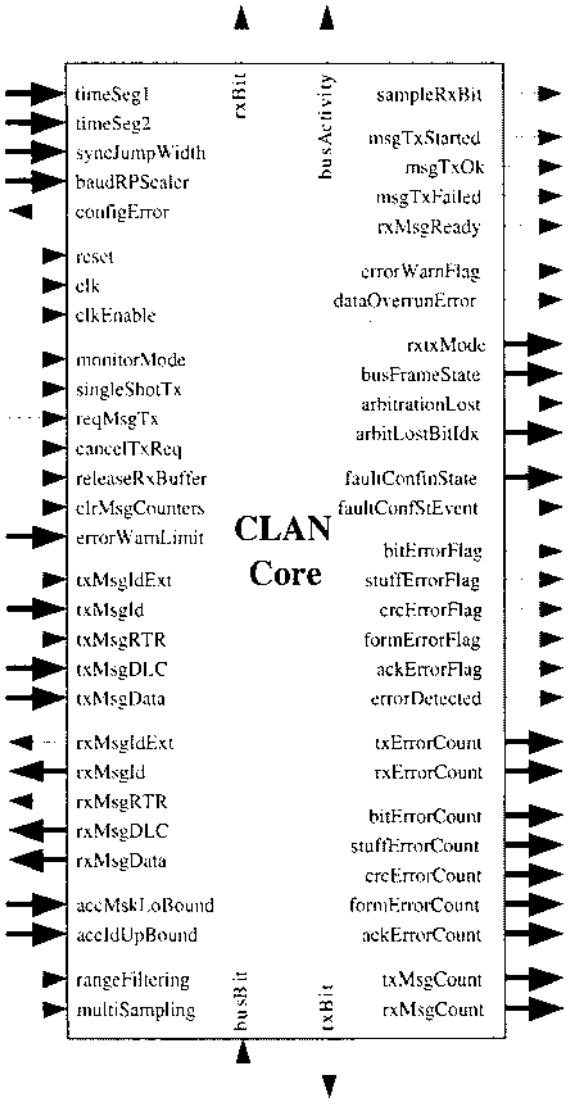


Figure 1 - *CLAN Core* module interface.

functional groups: *Synchronization and Initialization* (Table I), *Timing Configuration* (Table II), *Mode Setup* (Table III), *General Status and Statistics* (Table IV), *Transmission and Reception Data* (Table V), *Transmission and Reception Configuration* (Table VI), *Error and Fault Confinement* (Table VII), *Message Filtering Setup* (Table VIII) and *Bus Interface* (Table IX). A short description of each port is given into the tables below.

All interface control and status signals are sampled or switched at the falling edge of the *clk* synchronization clock. The *sampleRxBit* port (Table I) provides a clock signal synchronous with the *Sample Point*. This signal combined with the frame state available on the *busFrameState* port is useful for clock synchronization on TTCAN implementations. The single shot transmission capability provided is useful for disabling automatic message retransmission in case of an error within TTCAN and FTT-CAN synchronous windows. The values applied to the *Timing Configuration* ports (Table II) must be stable to ensure a correct operation of the core.

Name	Type	Description
reset	In	Asynchronous reset input
clk	In	Main synchronization signal
clkEnable	In	Enable input for the core "clk" signal
sampleRxBit	Out	Clock signal synchronous with the Sample Point

Table I
SYNCHRONIZATION AND INITIALIZATION PORTS.

Name	Type	Description
timeSeg1	In	"Length - 1" of Time Segment 1 (in time quanta)
timeSeg2	In	"Length - 1" of Time Segment 2 (in time quanta)
syncJumpWidth	In	Synchronization Jump Width value (in time quanta)
baudRPScaler	In	Baud Rate Pre-Scaler value used for "clk" frequency division

Table II
TIMING CONFIGURATION PORTS.

Name	Type	Description
singleShotTx	In	When active disables automatic message retransmission in case of error
multiSample	In	Sampling mode for improved noise imunity
monitorMode	In	When active sets the output driver permanently to "recessive" level

Table III
MODE SETUP PORTS.

Name	Type	Description
rxtxMode	Out	Current RX/TX mode (None, Rx, Tx, Arbitration)
busFrameState	Out	Current state of the frame present on the bus
arbitrationLost	Out	Activated during one CAN bit time in case of arbitration lost
arbitLostBitIdx	Out	When "arbitrationLost" = 1 this output indicates the bit where arbitration was lost
txMsgCount	Out	Number of successfully transmitted messages
rxMsgCount	Out	Number of successfully received messages
clrMsgCounters	In	When activated clears the "txMsgCount" and "rxMsgCount" counters
busActivityFlag	Out	Indicates the presence of bus activity

Table IV
GENERAL STATUS AND STATISTICS PORTS.

Name	Type	Description
txMsgIdExt	In	Tx message extended identifier flag
txMsgId	In	Tx message identifier
txMsgRTR	In	Tx message RTR flag
txMsgDLC	In	Tx message DLC value
txMsgData	In	Tx message data bytes
rxMsgIdExt	Out	Rx message extended identifier flag
rxMsgId	Out	Rx message identifier
rxMsgRTR	Out	Rx message RTR flag
rxMsgDLC	Out	Rx message DLC value
rxMsgData	Out	Rx message data bytes

Table V
TRANSMISSION AND RECEPTION DATA PORTS.

A.2 Internal Structure

The internal structure of the *CLAN Core* module is shown on Figure 2. A short description of each block is given into the following subsections.

Bit Stuffing Unit

The *Bit Stuffing Unit* is used to:

- insert stuff bits on the transmitted bit stream;
- check and remove stuff bits from the received bit stream.

The *Bit Stuffing Unit* is shared by the transmission and reception parts of the core, because unless an error has occurred or the transmitter looses arbitration, within the stuffed fields the transmitted and the received bits should match.

CRC Unit

The *CRC Unit* calculates and checks the CRC sequence included in the frame. Similarly to the *Bit Stuffing Unit*, it is shared among the transmission and reception parts of the controller. In transmit mode, it calculates the CRC sequence during the *Start of Frame*, *Arbitration*, *Control* and *Data fields*. During the *CRC Sequence* field, the calculated sequence is shifted into the bus. In reception mode it compares the received sequence with the locally computed sequence in order to detect errors on the received bit stream.

Reception Unit

The *Reception Unit* latches the bus bit at the *Sample Point* and performs the de-serialization of the reception bitstream. It also acknowledges a correctly received frame during the *Acknowledge Slot* field.

Transmission Unit

The *Transmission Unit* performs the serialization of the message to send, determining the bit to be transmitted by

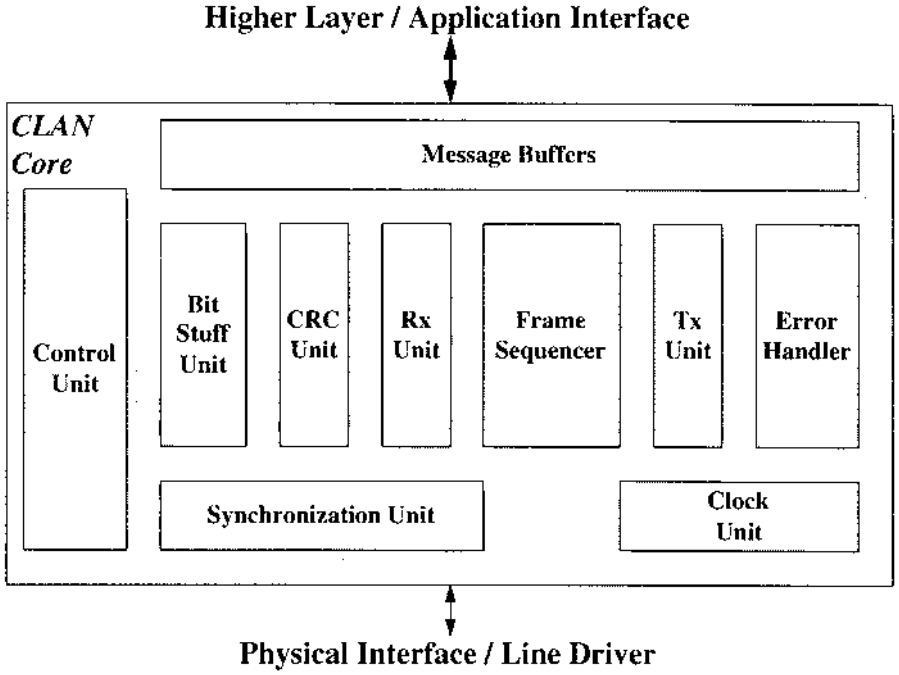


Figure 2 - *CLAN Core* internal block diagram.

the node and setting it at the beginning of the bit time. The sources for the transmitted bit are the following:

- A message bit from the *ID*, *RTR*, *DLC* or *DATA* fields;
- A stuff bit;
- A *CRC* bit;
- A fixed polarity bit (*recessive* or *dominant*);
- An acknowledge bit generated by the *Reception Unit*;
- An error frame bit produced by the *Error Handler*.

Frame Sequencer

The *Frame Sequencer* plays a central role within the controller, performing the following tasks:

- Arbitration;
- Accepting requests to transmit messages;
- Detecting a *Start of Frame* field on the bus;
- Sequencing fields in *Data*, *Overload* and *Remote Transmission Request* frames;
- Signaling the successful transmission of a message and the end of a message reception;
- Responding to *Overload* frames.

Figure 3 shows a partial behavioral specification of the *Frame Sequencer*. It consists of two parallel state machines: the *Rx/Tx Mode State Machine* and the *Frame State Machine*. The former defines the operating mode of the controller. The last establishes the sequence of fields for all frame types except *Error* frames, which are generated directly by the *Error Handler* and applied to the *Transmission Unit*.

Error Handler

The *Error Handler* performs all activities related to fault confinement, error detection, counting and signaling. Internally it implements the mechanisms to detect the different error types and the error counters specified on the standard.

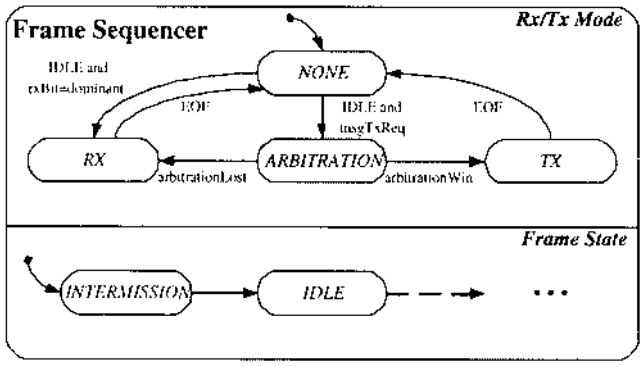


Figure 3 - Partial behavioral specification of the *Frame Sequencer* module.

When an *Error* frame has to be sent, the transmission is performed through the *Transmission Unit* and the *Frame Sequencer* is disabled until the end of the *Error* frame.

Message Buffers

As the name implies, the *Message Buffers* are used to store messages. Two buffers are accessible from the outside of the module: one for transmission and the other for reception. However, internally the *Transmission* and *Reception* units contain shift registers for message serialization and de-serialization that act as temporary buffers.

Clock Unit

The *Clock Unit* generates all the clocks required to control and synchronize the activities of the other core components. Behavioral modelling of the CAN controller has shown that two clock signals are required for such purposes [7]: a *Synchronization Clock* with frequency f_{SYNC} and *Control Clock* with frequency f_{CTRL}:

$$f_{SYNC} = \frac{1}{T_Q}$$

Name	Type	Description
reqMsgTx	In	When activated, requests the transmission of the message applied to the txMsg(IdExt, Id, RTR, DLC, Data) ports
cancelTxReq	In	When activated, cancels the previous transmission request, if still pending
msgTxStarted	Out	Activated during one CAN bit time at the start of a message transmission
msgTxOk	Out	Activated during one CAN bit time at the end of a successful message transmission
msgTxFailed	Out	Activated during one CAN bit time when a message transmission fails
rxMsgReady	Out	Activated during one CAN bit time at the end of a successful message reception
releaseRxBuffer	In	When activated, releases the Rx buffer, allowing the core to write a new received message on the buffer accessible trough the rxMsg(IdExt, Id, RTR, DLC, Data) ports
dataOverrunError	Out	Activated when a new message was received before an external release of the Rx buffer containing the previous received message. The newly message received is discarded

Table VI
TRANSMISSION AND RECEPTION CONFIGURATION PORTS.

$$f_{CTRL} = 2 \cdot f_{SYNC}$$
$$f_{CLK} = (baudRPScaler + 1) \cdot f_{CTRL}$$

where T_Q is the *Time Quantum* period and f_{CLK} is the *clk* frequency. It means that for a given *Time Quantum* value, an input clock with only twice the frequency is needed.

Control Unit

The *Control Unit* generates all signals that control the other units, mainly enable and reset signals. Figure 4 shows a simplified view of the core internal control sequence within a CAN bit time. The frequency divider that generates the *Control Clock* signal is triggered by the falling edge of the *clk* input. The majority of the units are triggered by the rising edge of the *Control Clock* and during the *Time Segment 2*, i.e. after the *Sample Point*. This imposes some restrictions on the duration of the *Time Segment 2*, namely its minimum duration must be 2 *Time Quanta*. This constraint is required to decrease the number of internally generated clock signals and to limit the frequency of the clock

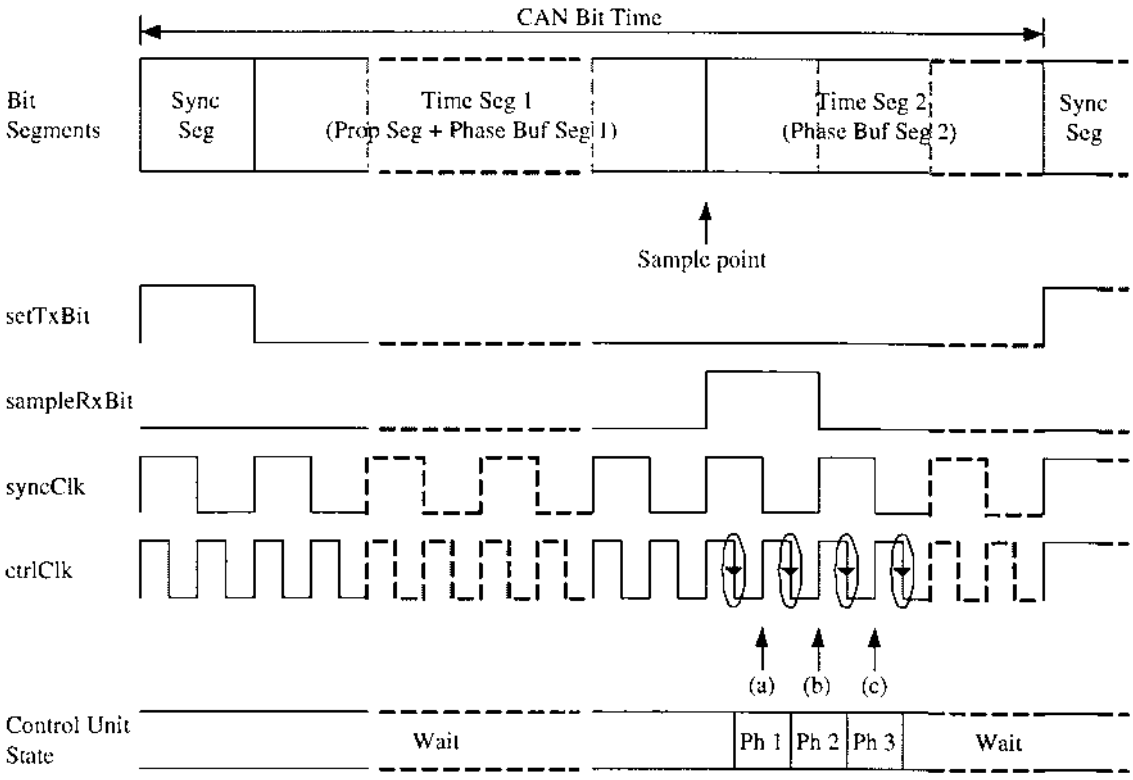
Name	Type	Description
configError	Out	Activated when the timing parameters are invalid
faultConfinState	Out	Current fault confinement state ("Error Active", "Error Passive", "Bus Off")
faultConfStEvent	Out	Activated during one CAN bit time after a change on the fault confinement state
bitErrorFlag	Out	Active during one CAN bit time in case of a bit error
stuffErrorFlag	Out	Active during one CAN bit time in case of a stuff error
crcErrorFlag	Out	Active during one CAN bit time in case of a CRC error
formErrorFlag	Out	Active during one CAN bit time in case of a form error
ackErrorFlag	Out	Active during one CAN bit time in case of a acknowledge error
errorDetected	Out	Active during one CAN bit time in case of a bit, stuff, CRC, form or acknowledge error
txErrorCount	Out	Tx Error Count as defined on the CAN specification
rxErrorCount	Out	Rx Error Count as defined on the CAN specification
errorWarnLimit	In	Threshold value used to flag a disturbed bus
errorWarnFlag	Out	Activated when one of the error counters is greater than the "errorWarnLimit" value
bitErrorCount	Out	Number of bit errors occurred
stuffErrorCount	Out	Number of stuff errors occurred
crcErrorCount	Out	Number of CRC errors occurred
formErrorCount	Out	Number of form errors occurred
ackErrorCount	Out	Number of acknowledge errors occurred

Table VII
ERROR AND FAULT CONFINEMENT PORTS.

applied to the core for a given transmission rate. However, it is important to note that this restriction is compliant with the maximum duration of the *Information Processing Time* defined on CAN specification.

Synchronization Unit

The *Synchronization Unit* generates the *sampleRxBit* and the *setTxBit* clocks used to latch the reception and transmission signals at the correct time instants, based on bus transitions and on the timing parameters of the node. The period of the CAN bit is given by the following expression:



Edges responsible for control unit state transitions

(a) - Frame sequencer update phase; Error handler detection phase

(b) - Rx buffer write; CRC unit enable; Bit stuff unit enable; Frame sequencer execution phase; Error handler update phase

(c) - Tx buffer read; Error handler reaction phase

Figure 4 - *CLAN Core* internal control sequence.

Name	Type	Description
rangeFiltering	In	When activated, filtering is performed based on the lower and upper bound of message identifiers specified by the next two ports; when deactivated, filtering is based on identifier patterns
accMskLoBound	In	Specifies the identifier lower bound or don't care bits of the identifier used for message filtering
accIdUpBound	In	Specifies the identifier upper bound or significant bit values of the identifier used for message filtering

Table VIII
MESSAGE FILTERING SETUP PORTS.

Name	Type	Description
busBit	In	Current bus level detected by the input transceiver
rxBit	Out	Bus level at the previous sample point
txBit	Out	Current level applied to the output transceiver

Table IX
BUS INTERFACE PORTS.

where T_{CLK} is the period of the external clock applied to the core. The period T_{CTRL} of the control clock is:

$$T_{CTRL} = T_{CLK} \cdot (baudRPScaler + 1)$$

The values of the timing parameters must respect the following relation:

$$timeSeg1 > timeSeg2 > syncJumpWidth$$

$$T_{BIT} = T_{CLK} \cdot 2 \cdot (baudRPScaler + 1) \cdot (timeSeg1 + timeSeg2 + 3)$$

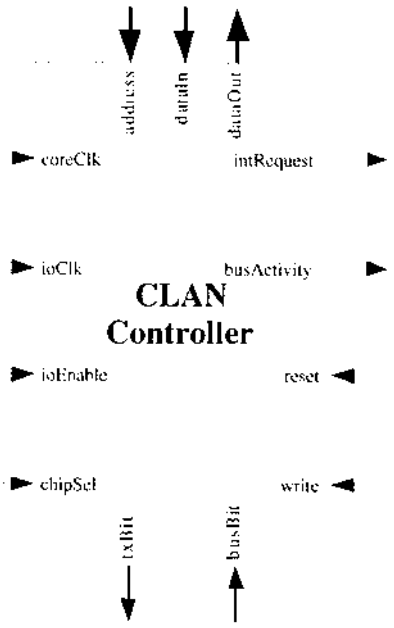


Figure 5 - *CLAN Controller* module interface.

otherwise the *configError* output will be active and the core will remain in the reset state.

B. CLAN Microprocessor Interface Module

Based on the *CLAN Core* module, different interfaces can be created. The first interface built was a synchronous parallel interface with a data bus of 8, 16 or 32 bits.

B.1 Interface Ports

The external interface of the *CLAN Controller* module is shown on Figure 5. A short description of each port is given on Table X. The *coreClk* is the *clk* synchronization signal of the *CLAN Core* module (Table I). Figure 6 shows examples of read and write cycles. The microprocessor changes the signals at the falling edge of the *ioClk* signal. The *CLAN Controller* validates the signals at the rising edge of the same clock. Thus, if the *ioClk* frequency is adequate for internal core synchronization, the *coreClk* and *ioClk* clock signals can be connected together to the same clock source.

B.2 Configuration Registers

The configuration registers map into a space of 128 addresses all the input and output ports of the *CLAN Core* module. All registers are at a fixed offset location independently on the bus width (Table XI). The complete description of the configuration registers can be found at the *CLAN Project Web Page* [8].

IV. MODELLING AND SIMULATION

The CLAN IP block was modelled with the VHDL hardware description language because VHDL provides the adequate abstractions to model the CAN controller building blocks, such as multiplexors, registers, state machines, etc. The model created contains about 3700 lines of code and it is completely independent of the implementation technology. It was successfully validated with the ModelSim VHDL simulator.

Name	Type	Description
reset	In	Asynchronous reset input
coreClk	In	Core internal synchronization signal
ioClk	In	Interface synchronization signal
chipSel	In	Global interface enable signal
ioEnable	In	Enable signals for individual bytes of a multi byte data bus interface
write	In	Write enable signal
address	In	Address bus
dataIn	In	Data input bus
dataOut	Out	Data output bus
intRequest	Out	Interrupt request output for microcontroller
busActivity	Out	Flag that indicates activity on the bus
busBit	In	Port for connection to the reception transceiver (line-driver)
txBit	Out	Port for connection to the transmission transceiver (line-driver)

Table X
CLAN Controller PORTS.

V. SYNTHESIS, IMPLEMENTATION AND TEST

The CLAN IP block was synthesized and implemented on a Xilinx XC2S300 Spartan-IIE low cost FPGA. The synthesis report is shown on Figure 8. The complete circuit occupies about 30% of the available slices (logic cells) corresponding to 100,000 logic gates. The core internal logic can operate up to 42MHz. Figure 7 shows the complete project hierarchy. To use the CLAN IP block as a black box in a project three components must be included:

- The file containing the synthesized netlist;
- The file *CAN.VHD* containing a package with generic CAN definitions;
- The file *CLANPublic.vhd* containing a package with CLAN specific definitions.

The *CLAN Core* was tested within a bus with other commercial CAN controllers operating at 1Mbit/seg. The test setup is depicted on Figure 9. The main purpose of this setup is to perform a simple functional validation of the controller that must retransmit all received messages.

The *CLAN Controller* module was also integrated on the ARPA System-on-Chip with a MIPS32 processor optimized for real-time systems [9]. An API library was developed that allows the configuration and communication with the *CLAN Controller* from a program in C language.

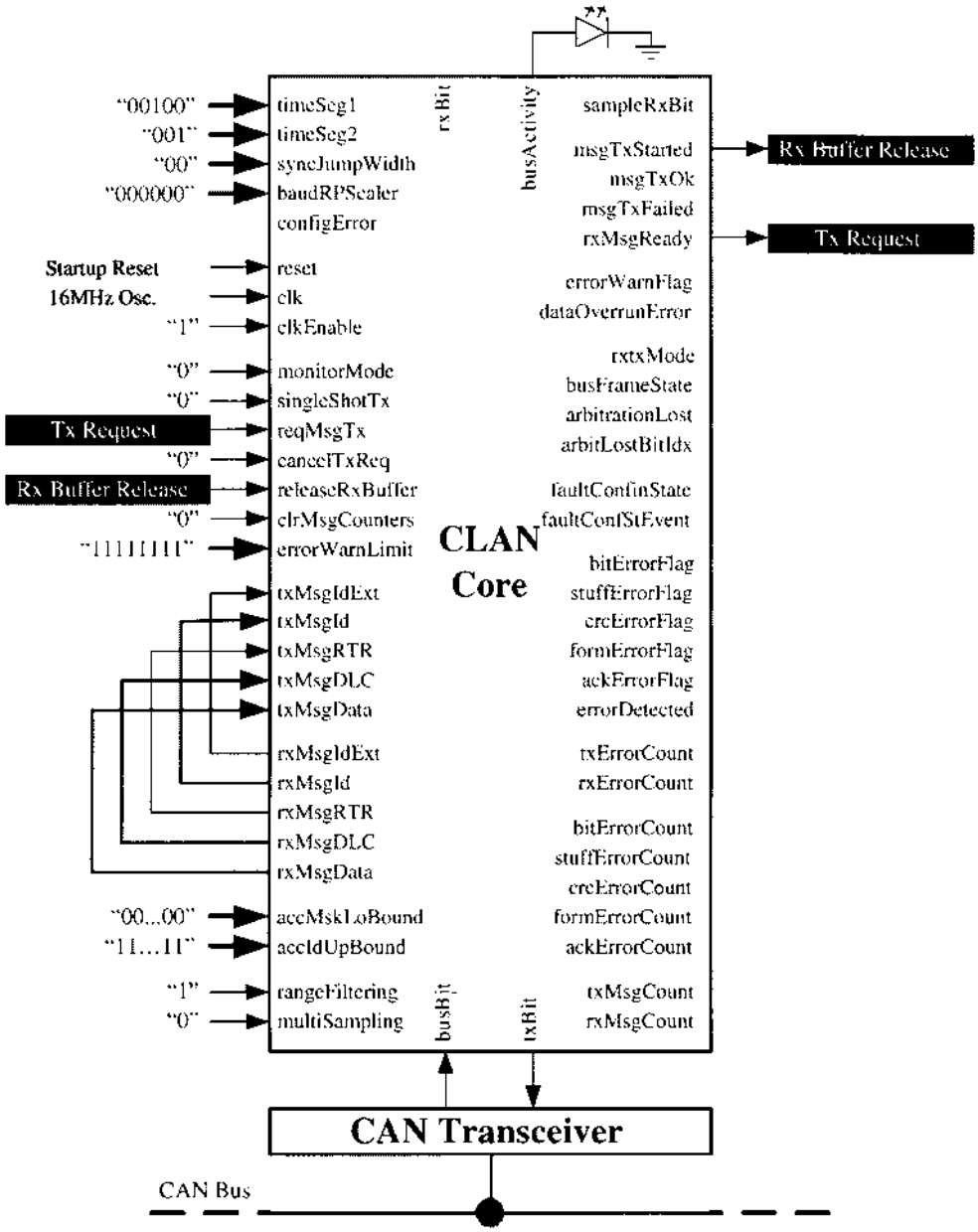


Figure 9 - *CLAN Core* loop-back test setup.

VI. CONCLUSION

A full CAN 2.0B controller with synchronous parallel microprocessor interface was presented in this paper. The IP core was developed for educational and research purposes. It communicates correctly with other commercial controllers operating up to 1Mbit/seg. However, it is important to refer that it was not validated with the CAN conformance tests. The web page of the CLAN project with detailed and updated information can be found at [8].

REFERENCES

[1] Robert Bosch GmbH, "CAN Specification Version 2.0 (http://www.can.bosch.com)", 1991.

[2] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1. edition, 1997.

[3] Thomas Fuhrer, Bernd Muller, Werner Dieterle, Florian Hartwich, Robert Hugel, and Michael Walther, "Time triggered communication on CAN (Time Triggered CAN - TTCAN)", in *ICC 2000 - 7th International CAN Conference*. October 2000, CiA - CAN in Automation.

[4] Luís Almeida, Paulo Pedreiras, and José Alberto Fonseca, "The FTT-CAN protocol: Why and how", *IEEE Transactions on Industrial Electronics*, vol. 49, no. 6, pp. 1189–1201, December 2002.

[5] ISO/TC 22/SC 3/WG 1, "Road Vehicles — Controller Area Network (CAN) Part 4: Time Triggered Communication", Tech. Rep. ISO/WD 11898-4, ISO, December 2000.

[6] CAN in Automation, "CAN products web page (http://www.can-cia.org)", 2005.

[7] Arnaldo Oliveira, Pedro Fonseca, Valery Sklyarov, and Antônio Ferrari, "An object-oriented framework for can protocol modeling and simulation", in *FET 2003 - The 5th IFAC International Conference on Fieldbus Systems and their Applications*, July 2003, pp. 243–248.

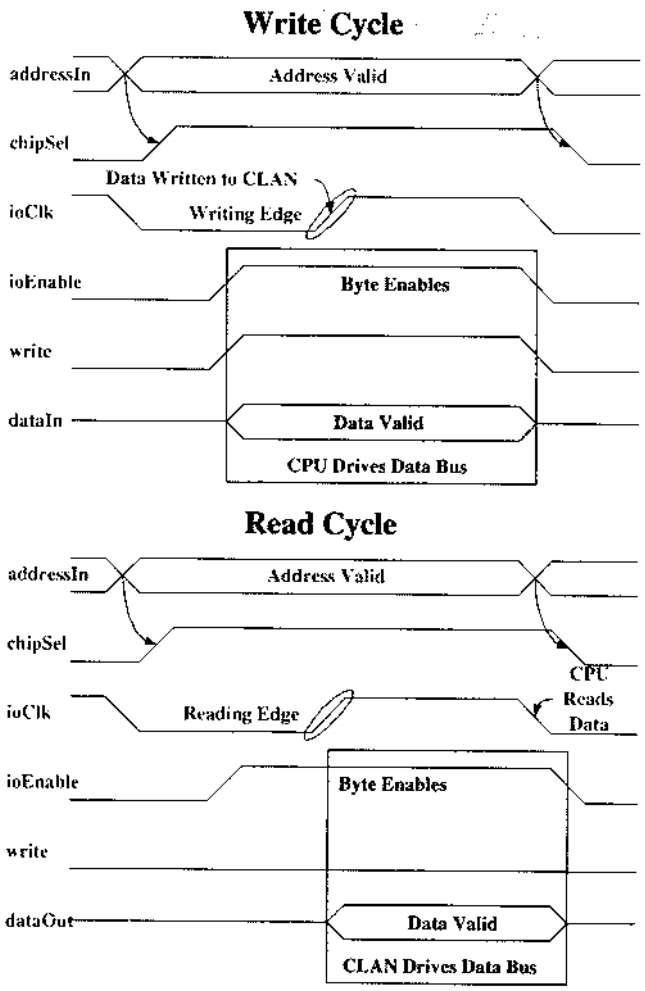


Figure 6 - *CLAN Controller* write and read bus cycles.

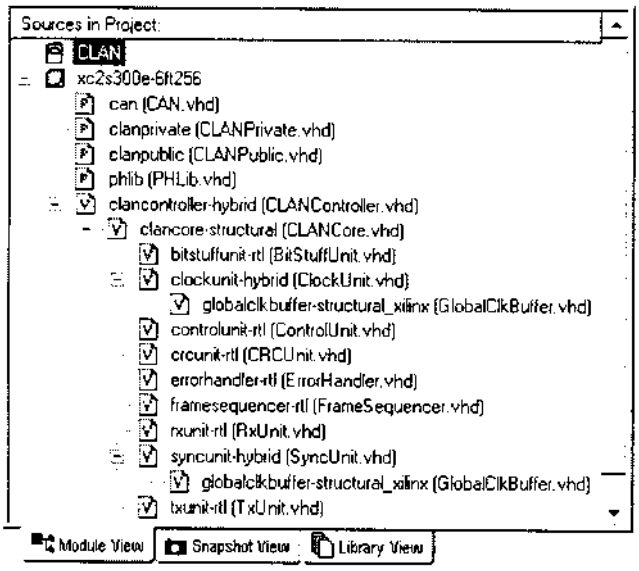


Figure 7 - CLAN Project hierarchy.

Offset (Hex)	Register Name	Access Type
00h	Command	W
00h	Status 0	R
04h	Status 1	R
08h	Control	RW
0Ch	Rx/Tx Status	R
10h	Arbitration Lost Capture	R
14h	Error Status	R
18h	Bus Timing	RW
1Ch	Interrupt Enable	RW
20h	Interrupt Identification	R
24h	Rx Error Count	R
28h	Tx Error Count	R
2Ch	Error Warning Limit	RW
30h	Bit Error Count	R
34h	Stuff Error Count	R
38h	CRC Error Count	R
3Ch	Form Error Count	R
40h	Acknowledge Error Count	R
50h	Acceptance Mask/Lower Bound	RW
54h	Acceptance Identifier/Upper Bound	RW
58h	Rx Message Count	R
5Ch	Tx Message Count	R
60h	Rx Message Control	R
64h	Rx Message Identifier	R
68h	Rx Message Data 03	R
6Ch	Rx Message Data 47	R
70h	Tx Message Control	RW
74h	Tx Message Identifier	RW
78h	Tx Message Data 03	RW
7Ch	Tx Message Data 47	RW

Table XI

CLAN Controller REGISTER NAMES AND OFFSETS.

```
Final Synthesis Report
=========================================================
Device utilization summary:
Selected Device : 2s300eft256-6
  Nº of Slices:              931 out of 3072  ( 30%)
  Nº of Slice Flip-Flops:    863 out of 6144  ( 14%)
  Nº of 4 input LUTs:       1513 out of 6144  ( 24%)
  Nº of TBUFs:                32 out of 3072  (  1%)
  Nº of GCLKs:                 2 out of    4  ( 50%)
Timing Summary:
  Speed Grade: -6
  Min. period: 23.3ns (Max. Frequency: 42.8MHz)
  Min. input arrival time before clock:  11.4ns
  Max. output required time after clock: 10.2ns
  Maximum combinational path delay:       3.8ns
```

Figure 8 - Summary of *CLAN Controller* synthesis report.

[8] Arnaldo S. R. Oliveira, "CLAN project web page (http://www.ieeta.pt/~ arnaldo/projects/clan/)", 2005.

[9] Arnaldo S. R. Oliveira, Valery A. Sklyarov, and António Ferrari, "The ARPA project - creating an open source real-time system-on-chip",

Electrónica e Telecomunicações, vol. 4, no. 3, pp. 389-392, September 2004.