FPGA BASED CAN PROTOCOL CONTROLLER

Tejaswini Hulawale[1], Neha Koul[2], Shivang Gupta[3]

[1, 2, 3] *E&TC, JSPM's RSCOE, Pune, Maharashtra, (India)*

ABSTRACT

This paper presents the development of a CAN interface to be integrated in custom circuits (like ASICs, FPGAs). The CAN controller has been designed in VHDL. It can be targeted to different implementation technologies. It also helps us in the design process of the FPGAs, planning, simulating, testing, coding and finally programming the FPGA. The method of designing the block is also presented in the paper to easily design digital systems to be reused in different applications, assuring its quality and reliability. After dumping the code into the FPGA kit, the FPGA kit will act as a TRANSCIEVER. All the tools work together to control the CAN controller. They also receive data and forward to the CAN controller which finally forward the data to the CAN bus.

Keyword: CAN, FPGA, Transceiver, VHDL.

1. INTRODUCTION

1.1 HISTOTY OF CAN

The CAN protocol was internationally standardized in 1993 as ISO 11898-1 and it comprises of the data link layer of the seven layer of ISO/OSI reference model.CAN (Controller Area Network) is a serial bus system, which was firstly developed for automotive applications in the early 1980's. CAN provides with two communication services: the sending of a message (data frame transmission) and the requesting of a message (remote transmission request, RTR). A sender sends information which is transmitted to all devices on the bus. All the devices which read the message first and then they decide if it is relevant to them. It detects the error and auto-corrects it.

The protocol is also widely used in industrial automation and other areas of networked embedded control, with applications in various products such as production machinery, medical equipment, building automation, weaving machines, and wheelchairs. Combining networks and mechatronic modules makes it possible to reduce both the cabling and the number of connectors, which increases production and reliability.

1.2 About Can

The CAN protocol standardizes the physical and data link layers, which are the two lowest layers of the open systems interconnect (OSI) communication model. For most systems, higher-layer protocols are needed to enable efficient development and operation. Such protocols are needed for defining how the CAN protocol should be used in applications, for example, how to decide the configuration of identifiers with respect to application messages, how to package application messages into frames, and how to deal with the things like start-up and fault handling. Note that in many cases only a few of the OSI layers are required. Many real-time issues and redundancy management are not covered by the OSI model. The adoption of CAN in a variety of application fields has led to the development of several higher-layer protocols, Device Net, CAN open and CAN

Kingdom. Their characteristics reflect differences in requirements and traditions of application areas. The progress and success of CAN are due to a number of factors. The evolution of microelectronics cemented the way for introducing distributed control in vehicles. In the early 1980s there was, however, a lack of low-cost and standardized protocols suitable for real-time control systems.

II. SYSTEM DESIGN

The main assignment of this project was to produce a working FPGA using the hardware-descriptive language VHDL and Actel Desktop.

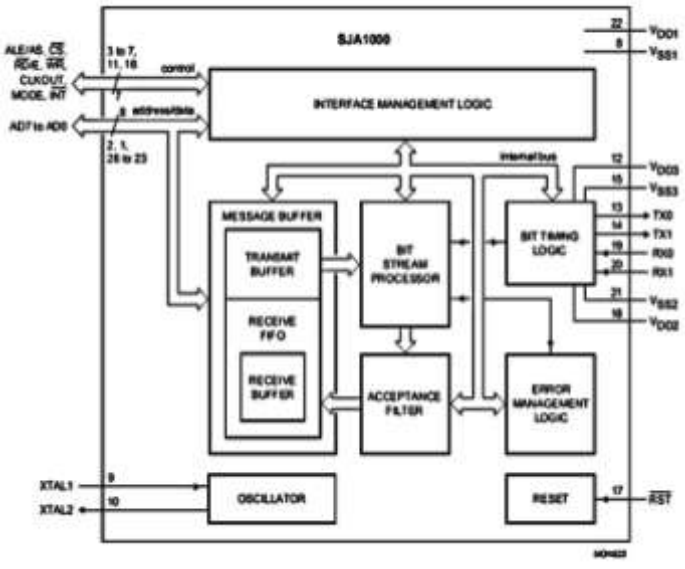
The work consists of:

- Planning of the FPGA to work with the CAN-bus.
- Producing the code.
- Testing the program code in the simulator tool.
- Synthesizing a code which is necessary to program a working FPGA.
- To test the circuit board with the FPGA on so that it works with the CAN-bus.

The CAN controller described above has been designed with VHDL-based design tactic. The main aim of this "virtual component" is to be reused in different applications and to be implemented in different target technologies. Due to the nature of the application, the design has been made at the Register Transfer level, at the architectural design phase. Those tools involved a desktop to manage the files and write the code, a synthesizer tool to create files needed for the programming unit that creates the useable FPGA.

When the FPGA began to take form a small but useful tool was acquired to make it possible to study a completed FPGA signal using a computer and a small unit as an interface between. The company Actel sells these tools. The normal way to design an FPGA is to create the code necessary to program the FPGA. Before programming, the generated code from a synthesizer tool is run in a simulator tool to see if it will work in theory. Theory and practice does not always same but it's still a good indication if it will work in practice. Then the generated files are used to program the FPGA in a programming device.

2.1 Block Diagram



2.1.1 Bit Stream Processor

The data blocks arrive in a serial bit stream and they are coded by the method of "bit stuffing". After five bits with the same polarity a stuff bit with opposite level is inserted to force the necessary edges to be resynchronized. The stuff bits are filtered from the received bit stream and the coded data is transmitted to the interface module.

2.1.2 Message Buffer

This block is in charge of the identification of the different messages fields (start of frame, data field, control field, arbitration field, etc.). The message processor cuts up the received message and inserts the bits in the corresponding registers. It is also responsible for building the messages to be transmitted: it creates the message from the different fields. which are taken out of the application. The message processor takes care of the acceptance filtering, offering different possibilities. One or several identifiers can be indicated explicitly (Full-CAN) or it is operated with a mask register (Basic-CAN) . The implementation we are developing will use the Basic-CAN strategy. It takes the received identifier to be compared with a register of acceptance filter. If they are not identical, the received message will be rejected.

2.1.3 Error Management Logic Unit

The Error Management Logic Unit is responsible for the fault confinement of the CAN protocol, which contains a mechanism for the automatic location and switching-off of defectives nodes. This mechanism is performed with two error counters: Receive Error Count and Transmit Error Count. The Error Management unit is also in charge of changing the error counts, setting the suitable error flag bits and interrupts and changing the error status (active, passive and bus off).

2.1.4 Cyclic Redundancy Code Generator And Comparator

Each message is provided with a 15-bit-long CRC code. This code is generated from the preceding telegram sections (start of frame, arbitration field, control field, data field). When receiving a message frame, the code is created analogously from the received data and compared with the CRC field in the message. This implies an error protection for the messages through the network.

2.1.5 Transmission Controller:

The Transmission Controller block controls the output drivers and the synchronization of the bus with the CAN clock, which is generated by the Bit Timing block. It is the final responsible of the transmission: it controls the transmitted message, which comes from the message processor, the error counters and the bit timing logic.

2.2VHDL

VHDL is an abbreviation of Very high-speed integrated circuit Hardware Description Language. It is a standardized language used to verify, specify and design electronics. VHDL was invented by the US department of defense at the beginning of the eighties and was made a standard for modeling and simulating. The translation of VHDL-code to a net- list for e.g. an FPGA is called synthesis. The synthesizing process is not made into a standard thus it is the synthesizer tool that decides what VHDL-code constructs that are supported for synthesis. VHDL is an object-based language. It is a general hardware-descriptive language, which gives several opportunities to describe the same behavior with different language designs. The design is made up of components consisting of two parts.

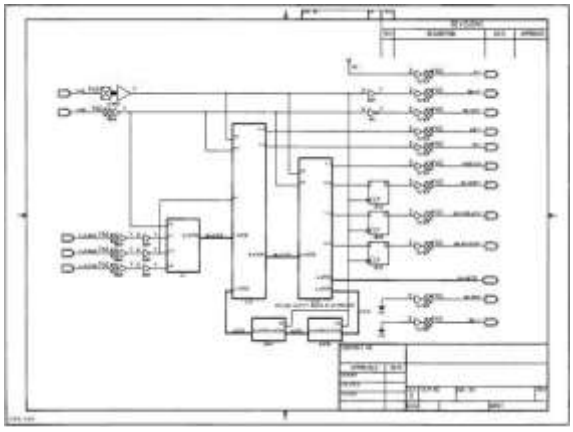
Entity in and output signals of the component Architecture Behavior of the entity, described by various abstraction models. The component can be described by different abstraction- levels and structural descriptions. The abstraction- levels can describe the same function but with different levels of detail. When using VHDL it is possible to mix different abstraction-levels simply by connecting the different components that are using different levels. The different levels used in practical electronic design are:

- The Behavior-model
- RTL-Model (Register Transfer Level)
- Gate- level

The behavior model is used at an early stage as a specification on how the circuit is supposed to work, thus it is very easy to read and can be used for documentation. The RTL- model describes the behavior in asynchrone and synchrone state machines, bus-structures, operators, registers, multiplexers, ALU and many more structures. These exist in different language-designs that can be synthesized if the synthesis-tool supports it .Gate level is the lowest abstraction level used for synthesis. At this level a gate-net is written or the design is described using Boolean algebra. This level gives the most control over synthesis and optimizing of circuit-area. Structural descriptions or design- hierarchies are used in VHDL to hide details. The information is hid in a “black box”, thus only the signals to the “box” can be seen. It is the designer that decides how many hierarchies to use in the design. Using both language-abstractions and structural descriptions the result will give a decreasing level of detail toward the top of the hierarchy. When testing the code written in VHDL to verify the correct behavior of a component a test-bench is used. This test-bench is a VHDL-code that generates stimuli to the component. By means of the test-bench on the component’s different abstraction- levels it is possible to verify the correct function at the different levels.

2.3 FPGA

This is the central piece of this project; it handles and controls everything in it. The diagram shown below is a picture of its internal structure.



2.4 Actel FPGA

The FPGA used for this project is an FPGA from Actel, belonging to the 42MXfamily. This family has FPGAs with the number of gates ranging from 2000 to 36000. The largest model of Actel FPGA has SRAM. They come in various packages depending on model, some well-matched with other models of the same family, some not. It is designed with different logic modules.

.

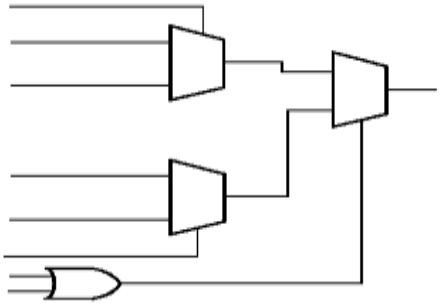


Figure: Logic module

2.5 TOOLS

2.5.1 Smart Design

It is a tool to enable faster creation of simple to complex System on chip(Soc)designs including processor/bus based and fusion mixed signal design. Complete FPGA systems and subsystems can be designed in minutes by selecting from Actel's core libraries, making rapid and error-free connections and automatically creating a synthesis ready HDL file.

2.5.2 Core Console

CoreConsole is a system-level development tool and IP development platform that simplifying the task of assembling and connecting IP for implementation in Actel FPGAs. It enables us to select IP components from a database supplied by Actel and third-party IP vendors, and graphically switch them together to build a processor-based system-level integration(SLI) design.

2.5.3 Synplify Synthesis

It accepts high-level input written in industry-standard hardware description languages(Verilog and VHDL), and using the Synplicity Behavior Extracting Synthesis Technology algorithms, they convert the design into small, high performanbe design netlisis for admired technology vendors. They can also write VHDL and Verilog netlists after synthesis, which you can then simulate in order to verify functionality.

2.5.4 Modelsim

It is a powerful simulator that can be used to simulate the behavior and performance of logic circuits. The simulator allows the user to give inputs to the designed circuit and to observe the corresponding outputs generated in response.

2.6. Error Correction and Detection

This mechanism is used for detecting errors in messages appearing on the CAN bus, so that the transmitter can transmit the message again. The CAN protocol specifies five different ways of detecting errors. Two of these works at the bit level, and the remaining three at the message level.

1. Bit Monitoring.
2. Bit Stuffing.
3. Frame Check.
4. Acknowledgement Check.
5. Cyclic Redundancy Check

1. Each transmitter on the CAN bus monitors (i.e. reads back) the transmitted signal level. If the signal level is different from the transmitted signal level, a Bit Error is signaled. No bit error is raised during the arbitration process.

2. When five consecutive bits of the same level have been transmitted by a node, it will add a sixth bit of the opposite level to the outgoing bit stream. The receivers will eradicate this extra bit. This is done in order to avoid excessive DC components on the bus, but it also gives the receivers one more opportunity to detect errors: if more than five consecutive bits of the same level arrives on the bus, a Stuff Error is signaled.

3. Some parts of the CAN message have a specific format, i.e. the standard defines precisely what levels must arrives and when. (Those parts are the CRC Delimiter, End of Frame, ACK Delimiter, and also the Intermission). If a CAN controller detects an invalid value in one of these fields, a Frame Error is signaled.

4. All nodes on the bus that correctly receives a message (regardless of their being "interested" of its contents or not) are expected to send a dominant level in the so-called Acknowledgement Slot in the message. The transmitter will transmit a recessive level here. If the transmitter is not able to detect a dominant level in the ACK slot, it will signaled an Acknowledgement Error.

5. Each message features a 15-bit Cyclic Redundancy Checksum and any node that detects a different CRC in the message than what it has calculated itself will produce a CRC Error.

2.6.1Synchronize

Suppose a node receives a data frame. Then it is essential for the receiver to synchronize with the transmitter to have proper communication. But we are not having any explicit clock signal that a CAN system can use as a timing reference. As an alternative, we use two mechanisms to maintain synchronization, which is shown below.

2.6.2Hard synchronization

It occurs at the Start-of-Frame or at the transition of the start bit. The bit time is again started from that edge.

2.6.2Resynchronization

To compensate for phase differences and the oscillator drift between transmitter and receiver oscillators, further synchronization is needed. The resynchronization for the successive bits in any received frame occurs when a bit edge doesn't occur within the Synchronization Segment in a message. The resynchronization is automatically appealed and one of the Phase Segments are shortened or lengthened with an amount that depends on the phase error in the signal. The highest amount that can be used is determined by a user-programmable number of time quanta known as the Synchronization Jump Width parameter (SJW).

Higher Layer Protocols:

Higher layer protocol (HLP) is required to manage the communication within a system. The term HLP is derived from the OSI model and its seven layers. But the CAN protocol just explains how small packets of data may be transported from one point to another safely using a shared communications medium. It contains nothing on the topics such as transportation of data larger than CAN fit in an 8-byte message, flow control, node addresses, establishment of communication, etc. The HLP gives solution for these topics.

Higher layer protocols are used for:

1. Standardize start up procedures including bit rate setting.
2. Distribute addresses among participating kind of
3.Determine the layout of the messages
4. Provide routines for error handling on system level

III. CONCLUSION

The design of a virtual component for a CAN controller has been reported in this paper. The importance of CAN networks in distributed control applications has driven the possibility of its integration in different types of devices (ASICs, FPGAs). The importance of the design process to ensure the quality of the design to be practically reused in different applications.

We are now making the final tests to the CAN to integrate it in a Actel FPGA that will be used in a CAN network controlled.

REFERENCES

[1] J. de Lucas, M. Quintana, T. Riesgo, Y. torroja, J. Uceda, "Design of CAN interface for cusom circuits".

[2] Michiel van Osch, Scott A. Smolka, "Finite-State Analysis of the CAN bus Protocol".

[3] Blagomir Donochev, Marin Hristov, "Implementation of CAN controller with FPGA Structures".

[4] Robby Andersson, "FPGA Design of a Controller for a CAN Controller".