CAN Protocol Implementation

Using the Xilinx Nexys 4 FPGA board

List of Authors (David Guoin, William Couturiaux, Garrett Willobee)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: wlcouturiaux@oakland.edu, dbguoin@oakland.edu, gfwillob@oakland.edu

***Abstract*—Our goal is to implement a bit sequence identical to that of the CAN protocol using VHDL logic and a Nexys 4 FPGA board.**

I. INTRODUCTION

The idea behind our project was to program the logic for the Base Frame Format and CAN protocol onto an FPGA board Nexys 4 board. We wanted to then be able to verify the output from our data on the 7 segment displays and be able to pass 4 different hard-coded messages from our top level design onto the board depending on the selector position of switches. If we would had been successful in completing this we would have tried to add more functionality to our design in the form of serial data transmission and more nodes to communicate within our board. Unfortunately due to time constraints and project deadlines, we were unable to meet some of the core functionality portions of our project within the time frame we wanted.

II. METHODOLOGY

A. *Researching CAN protocol*

A large aspect of our project was to research and understand the CAN format and functionality of a CAN controller. Controller Area Network (CAN) is a type of serial communication that includes many features such as, bit stuffing, arbitration, cyclic redundancy checking, that allow it to transmit information to multiple devices while constantly checking the validity of the information sent. The message format includes a start bit, arbitration field, control field, data field, cyclic redundancy check field, acknowledgement field and end of frame field. Understanding what each of these fields are used for was the first step in creating a design that would allow us to properly coordinate events, store information and process information. Starting our research for this project was immensely helped by our professor, who provided us with a guide of the CAN protocol for a controller at the bit by bit sequence level. Any other research was done outside of this independently through internet articles and several different power-point presentations.

B. *Define functionality*

In hindsight our goals were larger than we perceived them to be at the start of the project. We attempted to create a CAN controller that would possess the following functionality: detection of start bit within a 0 bit stuffed stream of 1's, detect stuffed bits within the message, properly record each field, check CRC, send a reply message based of received data field, create a reply message with a generated CRC value and stuffed bits and display the data recorded. We started to understand how large those goals were about mid-way through the project when we started to really get a grasp on the logic and the methods we would need to write this project in the Hardware Description Language. A lot of time that we spent writing code and logic for this project was in a way, backtracked because we would understand previous state machines or portions only when we had moved onto the next. This made communication and timing very critical and difficult to understand.

C. *Design approach*

The design process started with the idea of a top level component that could read an incoming message. Then looking inside this component we defined all the lower level components that

would be necessary to get the job done. These components include a state machine that detects the start bit, a state machine that "removes" stuff bits and stores the separate fields into separate registers, a state machine that handles the acknowledgement bits, CRC state machine, and finally a state machine that detects the end of frame bits. A diagram of this can be seen in figure 1.

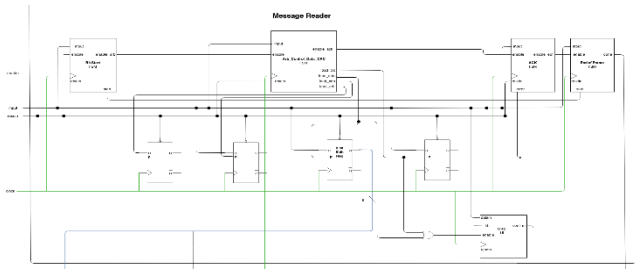

Figure 1, above: Diagram of components used to read a CAN message.

Once the basic format of the read component was laid out we began on the state machines. Creating the state machine that could detect stuffed bits and load registers was the most challenging. We began the process with four separate state machines that would detect stuffed bits in the individual fields, but realized that this did not take into consideration the occurrence of five similar bits between two adjacent fields. Therefore all the functionality was incorporated into one state machine that checked the arbitration through the CRC field for stuffed bits. After completing this state machine it made the implementation of new state machines easier given that they would follow a very similar format. A diagram of this state machine can be seen in Figure 2.

After working on the reader portion of the project we began on a message sending component. Our project was assumed to only include one main node that would be communicating with one other hypothetical node. Therefore depending on the data field received we would send a message with a corresponding data field value. To implement this functionality we created a decoder that would supply a data field based on the incoming data field. This data field would then be sent to both a CRC generator and a bit stuffing state machine. A counter was utilized to give the CRC generator enough time to generate the CRC given that it was done serially. Because we assumed the single node the arbitration field would always be the same, "00000000". Although the actual CAN protocol the data field can range from 8-64 bits, we simplified the data field to remain a static 8 bits. Therefore the corresponding control field would always be "001000". Finally all these fields were combined with the acknowledgement field and the end of frame field, which again would always remain all 1's. The bit stuffing state machine would then utilize a vector that would be large enough to hold the worst case scenario of stuffed bits. A diagram of the sender portion of the project can be seen in figure 3.

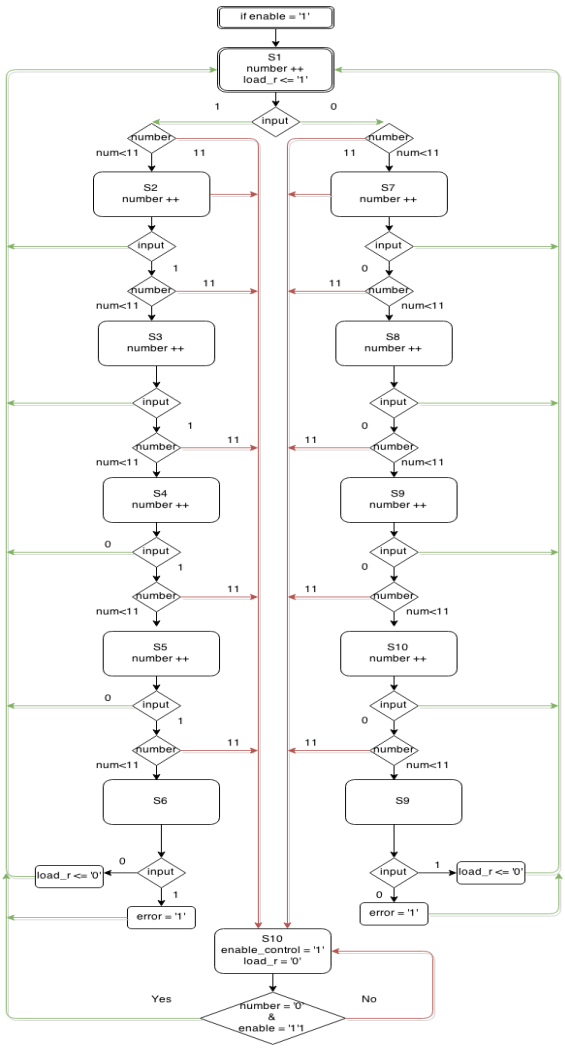

Figure 2, above: Diagram of Arb_control_data_crc state machine.

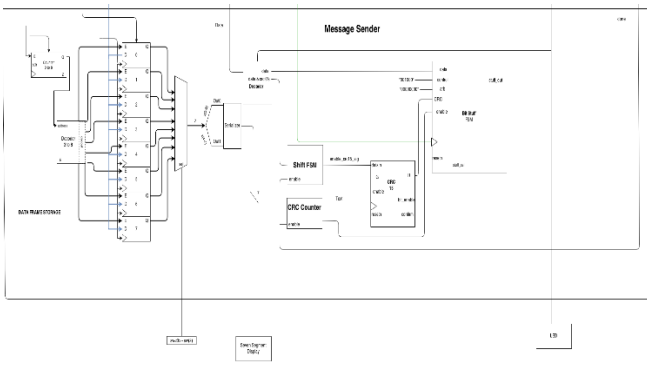

Figure 3, above: Diagram of message sending portion of project.

Finally after creating the reader and sender portions we created a final top level design that would be able to load a message serially into the reader which would communicate with the sender when it was done and then the sender would generate an output message. The data saved from the incoming message would be stored and displayed on two seven segment displays. A diagram of this can be seen in figure 4.

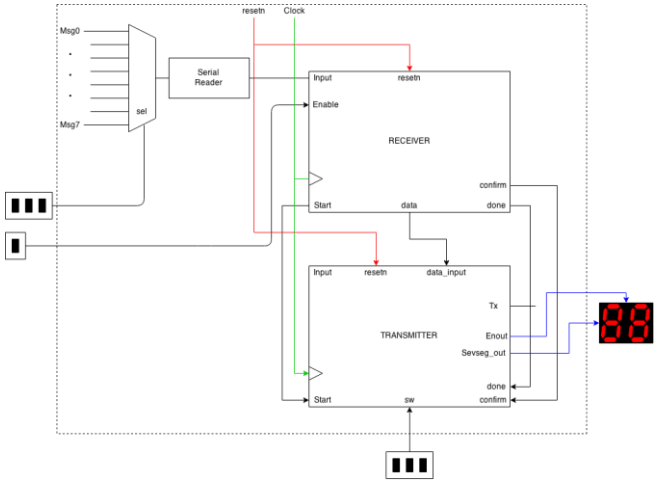

Figure 4, above: Diagram of top level design.

III. Experimental Setup

To verify our results, the bulk of our testing was done through test bench simulations. For every low level component we created a corresponding test bench that would simulate several possible scenarios. This was a very beneficial approach, but we could see as the complexity of our system increased our ability to use the test benches to debug was increasingly difficult. The final test would then be to download our program on to the Nexys 4 and observe if the proper data would be displayed on the seven segment displays.

IV. Results

We were able to simulate both the message reader portion and the message sending portion with good results. We were able to get the proper data saved into our RAM component; however we were not able to get the final top level to produce the proper outgoing message. In hindsight, it seems that we probably didn't simulate enough scenarios for our lower level simulations. Our focus was on the largest possible bit stuffing scenarios, and although this was important to test we should have created more scenarios before moving on to the next step. We also tried implementing the program on to the Nexsys 4 board, and although in the simulation we saved the correct data the board would not display it correctly.

V. Conclusion

Although we were not able to successfully produce the results that we set out to achieve, our success will have to be measured in what we learned and gained out of the experience. First and foremost would be an increased understanding of the CAN format. CAN is such a widespread and useful format, especially in the automotive industry. The relevance of the topic to electrical and computer engineering is obvious, and having a low level understanding of what is happening can only be a benefit for our future careers. Secondly would be an increased understanding of VHDL. Coding in a group atmosphere can help to speed up the learning process by observing and talking about techniques used by others. Third would be the importance of diligent testing. Not only should a test bench be created for each component, but a comprehensive list of scenarios should be generated so that problems can be debugged sooner rather than later.

References

[1] "can-cia.org," 2001-2015. [Online]. Available: http://www.can-cia.org/index.php?id=systemdesign-can-protocol.

[2] "CAN bus," 12 April 2015. [Online]. Available: http://en.wikipedia.org/wiki/CAN_bus. [Accessed 17 04 2015].

[3] P. D. Llamocca, "Introduction to Controller Area Network (CAN)," Oakland University, Rochester Hills, 2014.

[4] "Bit stuffing," 18 December 2014. [Online]. Available: http://en.wikipedia.org/wiki/Bit_stuffing.

[5] "Cyclic redundancy check," 15 April 2015. [Online]. Available: http://en.wikipedia.org/wiki/Cyclic_redundancy_check.