Eduardo Vargas Gonzalez – 162500

Luis Julián Álvarez Armenta – 163247

Homework No. 2

Automotive Networks

08/03/22

Spring 2021

UDLAP®

Abstract

In this work, two problems about the automotive networks subject are solved with the objective of testing the acquired knowledge about specific topics. These problems involve CAN Bus Access Logic and CRC-15. In order to solve these problems, software such as Multisim and VHDL EDA Playground are used so that the specific behavior needed to achieve in each problem is reached. This way, problems are successfully simulated and solved.

Introduction

As mentioned earlier, a CAN Bus Access Logic problem is addressed with a Carrier sense multiple access/collision avoidance (CSMA/CA) protocol, where the **CSMA/CA**, as the online stie “Vector” (2021) mentions, despite having a simultaneous bus access, it prevents collisions from occurring and in this case, it uses a non-destructive bitwise arbitration which is a key component in the bus access method, where the arbitration logic helps to decide if a CAN node is able to continue sending data or if it should stop sending . *Figure 1* shows a flow diagram of the CSMA/CA Protocol.

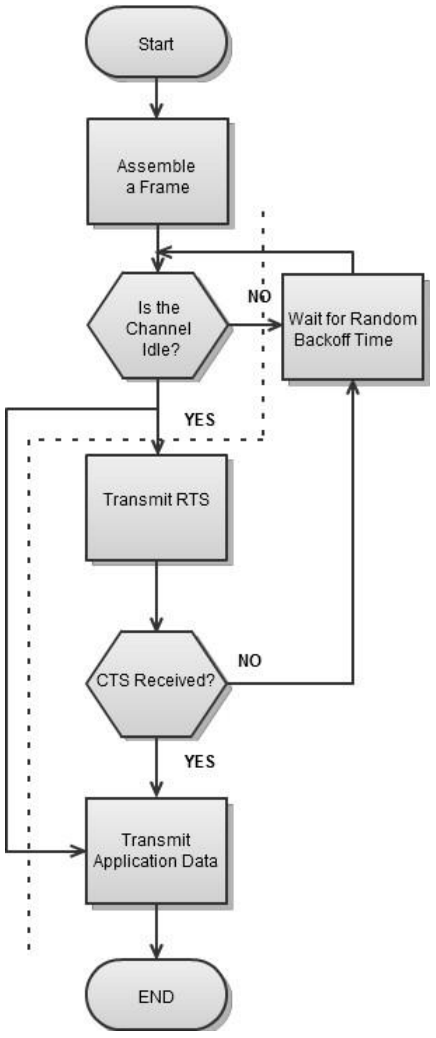


Figure 1. CSMA/CA flow diagram. Retrieved from: https://www.computerhope.com/

Cyclic redundancy codes (CRC) are used as an error-detecting method and this way, it is able to determine if a block of data has been corrupted. The way CRC works is by given a k-bit block of messages or bits, a bit sequence is generated by the transmitter (also known as frame check sequence) and the resulting frame can be divided a predetermined number. With the help of this number, the receiver is able to divide the incoming frame by that number, so, if there is no remainder, it is said that there was no error (Stallings, 2006).

There are different ways in which the CRC process can be viewed in order to be solved, these ways include: Modulo 2 Arithmetic, polynomials and digital logic. Next figures (*2, 2.1 and 2.2*) show an example of the different ways of looking at CRC.

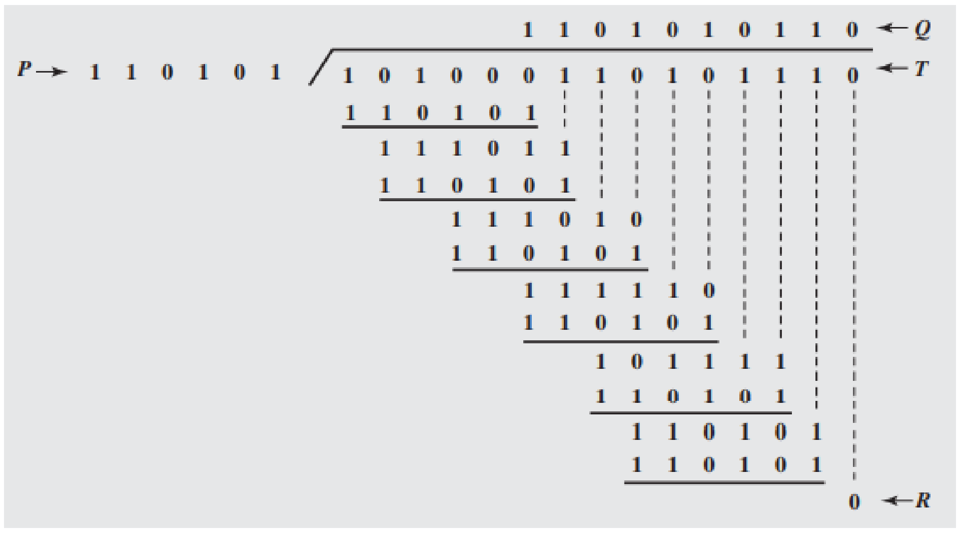


Figure 2. Modulo 2 Arithmetic example, where “P”: pattern of n-k+1 bits, “T”: n-bit frame to be transmitted and “R”: remainder. Retrieved from Data and Computer Communications 8th ed.

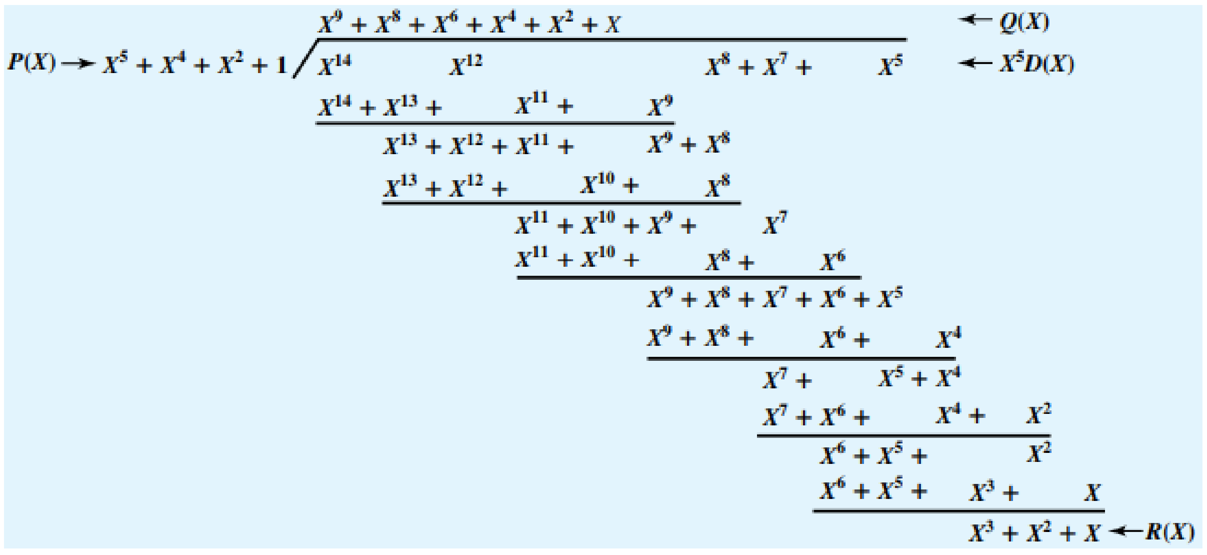


Figure 2.1. Polynomial's method example, where "P(x)": is the polynomial, "D(x)": polynomial representing data blocks and "R(x)": remainder. Retrieved from Data and Computer Communications 8th ed.

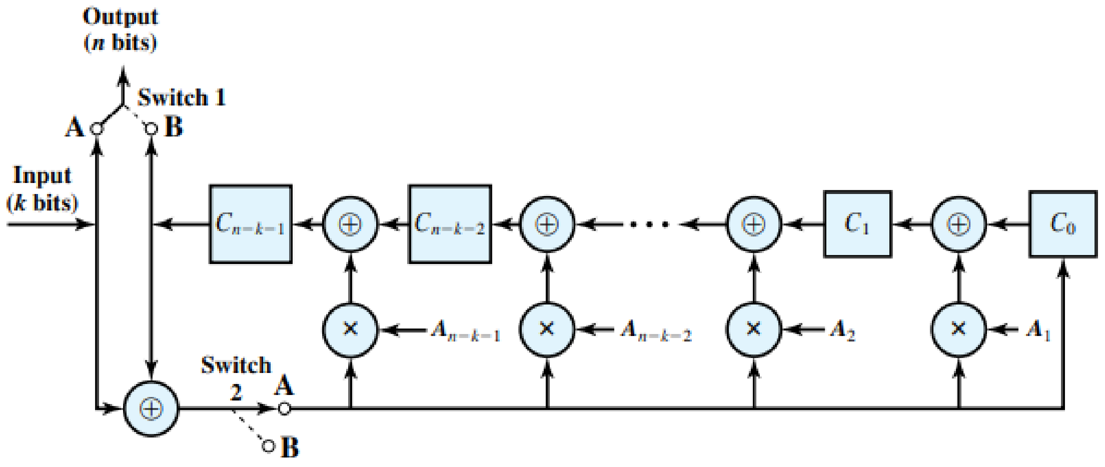


Figure 2.2. Digital logic method example, which shows the general CRC architecture to implement divisor. Retrieved from Data and Computer Communications 8th ed.

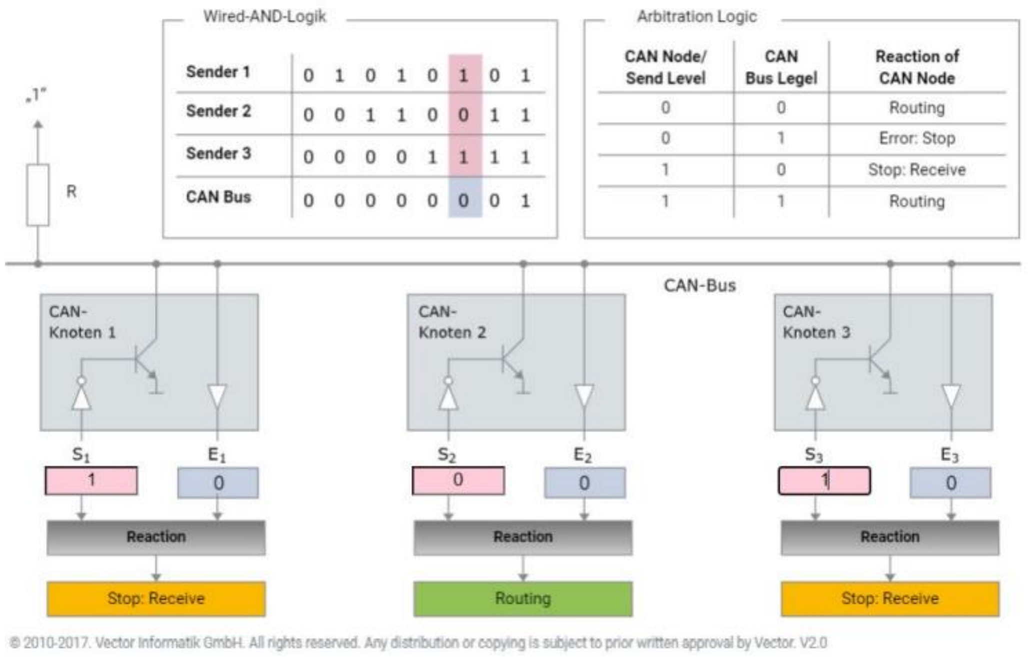
Methodology

In order to solve the problems, the following methodology was followed in each problem:

- **Exercise 1 (CAN Bus Access Logic)**

Using transistors, inverters, buffers, and other devices simulate in Multisim the behavior of the CAN Bus Access Logic (see *Figure 3*). This behavior should emulate the operation of a carrier sense multiple access/collision avoidance (CSMA/CA) protocol with non-destructive bitwise arbitration (BA) used in a CAN Bus network. *Figure 3* shows an example of three nodes in a CAN Bus network, in this case represented by three

transistors in open-collector configuration (“Wired-AND”). The bus level will be at low level (dominant) in case any number of transistors in the network output a dominant level. The bus level will only be at high level (recessive) when all transistors in the network output a recessive level. Wired-AND bus logic and arbitration logic ensure that the priority of the CAN message increases with decreasing identifier value. Finally, the arbitration logic decides whether a CAN node may continue to send, or whether it must stop sending. Note that nodes that lost arbitration will start a new arbitration as soon as the bus is free for access (idle) again. Thus, CAN provides a non-destructive bus arbitration.


„1"
R
CAN-Bus
CAN-Knoten 1
CAN-Knoten 2
CAN-Knoten 3
S1
E1
1
0
S2
E2
0
0
S3
E3
1
0
Reaction
Reaction
Reaction
Stop: Receive
Routing
Stop: Receive
© 2010-2017. Vector Informatik GmbH. All rights reserved. Any distribution or copying is subject to prior written approval by Vector. V2.0

Wired-AND-Logik

Sender 1	0	1	0	1	0	1	0	1
Sender 2	0	0	1	1	0	0	1	1
Sender 3	0	0	0	0	1	1	1	1
CAN Bus	0	0	0	0	0	0	0	1

Arbitration Logic

CAN Node/ Send Level	CAN Bus Legel	Reaction of CAN Node
0	0	Routing
0	1	Error: Stop
1	0	Stop: Receive
1	1	Routing

Figure 3. CAN Bus Access Logic.

First, in order to begin with the simulation of the behavior of the CAN Bus Access Logic, zeros (0) and ones (1) need to be sent through the bus. To do this, 3 DIP switches are added to the circuit which will simulate the 3 senders (connected to a 5V DC source) and thus, those ones (closed) and zeros (opened) sent by each of them. Then, the Wired-AND Logic is added to the circuit. To do this, 3 BJT NPN transistors are added, where all of the transistor’s collectors are connected to a 1 kΩ pull-up, all of the transistor’s bases are connected to a NOT Logic Gate which inputs are each of the sender’s DIP switches and all of the transistor’s collectors are connected to ground. Moving on, all of the sender’s signals (before going into the NOT gate) and all of the transistor’s collector signals (going through a diode) (represent the CAN bus) are directed to the “reaction” block.

The “reaction” block consists of several logic gates to achieve the desired output. These logic gates are classified by 2 stages, where the second stage consists of 4 AND logic gates (all connected to a LED indicator to represent the output (0 or 1)), where the inputs of the first AND gate are 2 NOT gates (first stage) which the input from one of them comes from the sender’s signal and the other one comes from the transistor’s collector signal. The inputs of the second AND gate are a NOT gate (which input is the sender’s signal) and an XOR gate (which inputs are the sender’s signal and the collector’s signal) (first stage). The inputs of the third AND gate are all exactly the same as with the second gate. The last AND gate consists only of the AND gate (second stage) which inputs are the sender’s signal and the collector’s signal. All of these connections are repeated for every sender (so, there are a total of 3 of these connections) in order to achieve all the reactions of the CAN nodes as part of the Arbitration Logic.

This way, the LED indicators of each sender (which represent all reactions: Routing, Error: Stop, Stop: Receive and Routing respectively) will only light up if the first LED receives bot zeros as inputs, the second LED receives a zero and a one as input, the third LED receives a one and a zero and the fourth LED receives both ones as inputs. So, all the logic gates mentioned above are used to obtain such inputs. In the “Results” section, all the results obtained are shown.

- **Exercise 2 (Calculate R (residual or frame check sequence) for use in CRC)**

In our lectures, we have discussed Cyclic Redundancy Codes (CRC) in CAN, the predominant method of checking whether data has been corrupted while being processed by the data link layer of the CAN network protocol. For this task, you must simulate in VHDL EDA Playground (or other VHDL simulation tools: Xilinx Vivado, Modelsim, Active-HDL, among others) an algorithm that calculates R, given D and G. Refer to the slides or reference, to review CRC. Your simulation should perform the following:

1. Receives as input the bit string G and the bit string D, were

• |D| = 20

• |G| = 16 (CRC-15 CAN)

2. Performs the XOR calculations of dividing D (padded with fifteen 0s) by G.

3. Outputs the value of the 15-bit string R. Note that your simulation should be tested on several inputs

By using VHDL EDA Playground website, the algorithm described above is simulated. In the "testbench" section of the program, the functionality of the program is described, while in the "design" section, the logic of the program is described, where in both sections, libraries to be used are defined (in this case IEEE library).

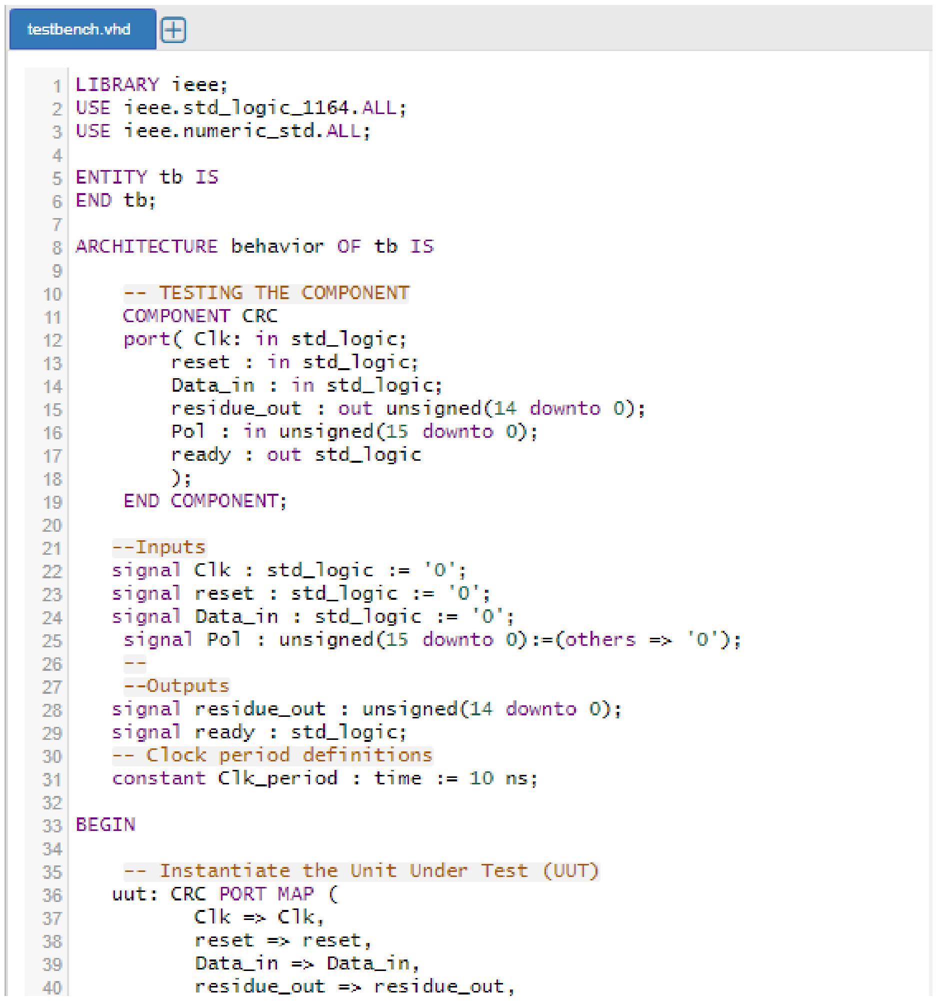
In the design section, entities, ports (inputs & outputs), signals, architectures, and processes are defined. The size of the polynomial (16 bits) to be used to calculate the remainder (R) (the polynomial is represented with by "Pol" in the program) is defined (it can actually have any combination) and in this case, the following polynomial is obtained:

(1) $X^{15} + X^{14} + X^{10} + X^{8} + X^{7} + X^{4} + X^{3} + 1$

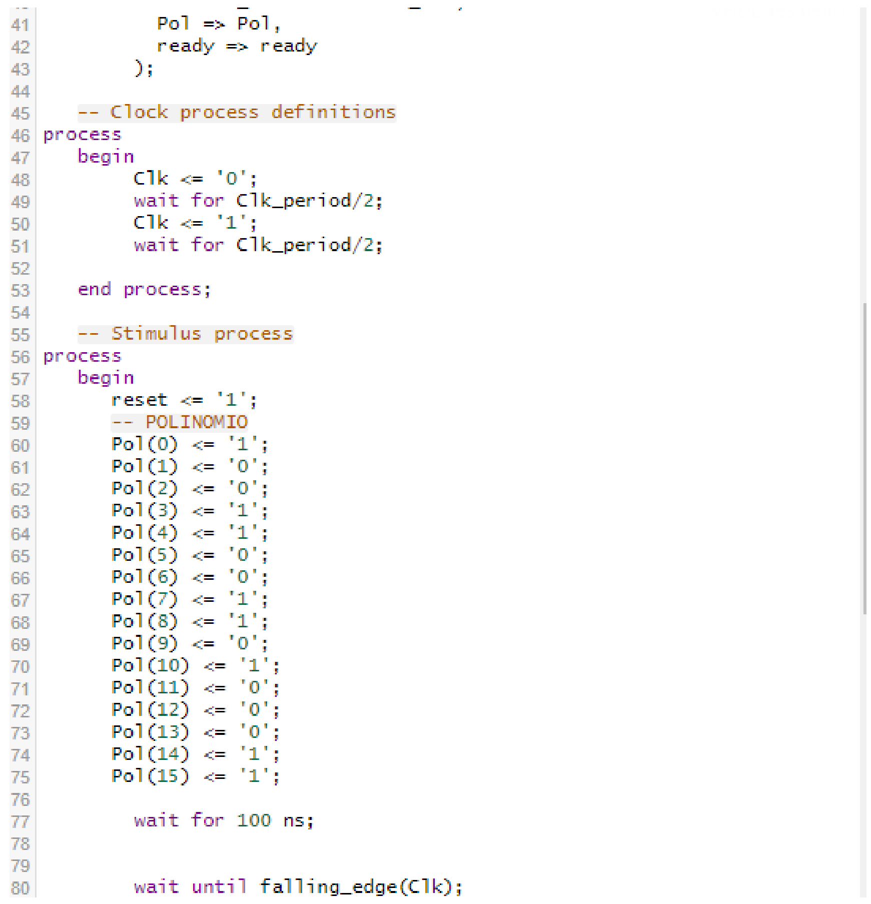
An architecture is also defined here, where it is stablished that the CRC calculation will be done every time there is a rising edge detected from a clock previously defined and the operations to obtain the remainder are defined. By using XOR operations with every term of the polynomial (defined in the ”design” section) in order to perform the division of D by G and this way, the remainder is obtained. Finally, for this section it is established for the program to check if the operations defined inside this architecture have already been performed for a maximum of 20 times (due to the 20-bit length). After all this, the “residue_out” output stores the resultant remainder which has a length of 15 bits.

On the other hand, in the “testbench” section, a clock period of 10 ns is defined and this way, this will work as a clock signal for different inputs to come in. Finally, all the values from the polynomial in (1) are defined.

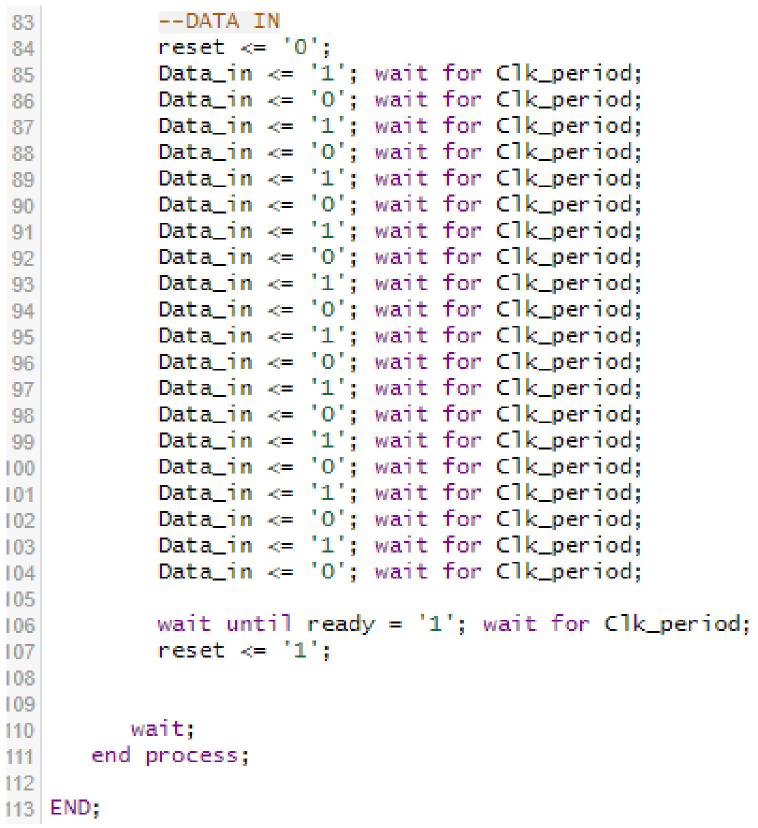
With this in mind, the following VHDL code is developed for each section (*figure 4* shows the testbench code, while *figure 4.1* shows the design code):



```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY tb IS
END tb;

ARCHITECTURE behavior OF tb IS

    -- TESTING THE COMPONENT
    COMPONENT CRC
    port( Clk: in std_logic;
        reset : in std_logic;
        Data_in : in std_logic;
        residue_out : out unsigned(14 downto 0);
        Pol : in unsigned(15 downto 0);
        ready : out std_logic
        );
    END COMPONENT;

   --Inputs
   signal Clk : std_logic := '0';
   signal reset : std_logic := '0';
   signal Data_in : std_logic := '0';
    signal Pol : unsigned(15 downto 0):=(others => '0');
    --
    --Outputs
   signal residue_out : unsigned(14 downto 0);
   signal ready : std_logic;
   -- Clock period definitions
   constant Clk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: CRC PORT MAP (
          Clk => Clk,
          reset => reset,
          Data_in => Data_in,
          residue_out => residue_out,
```

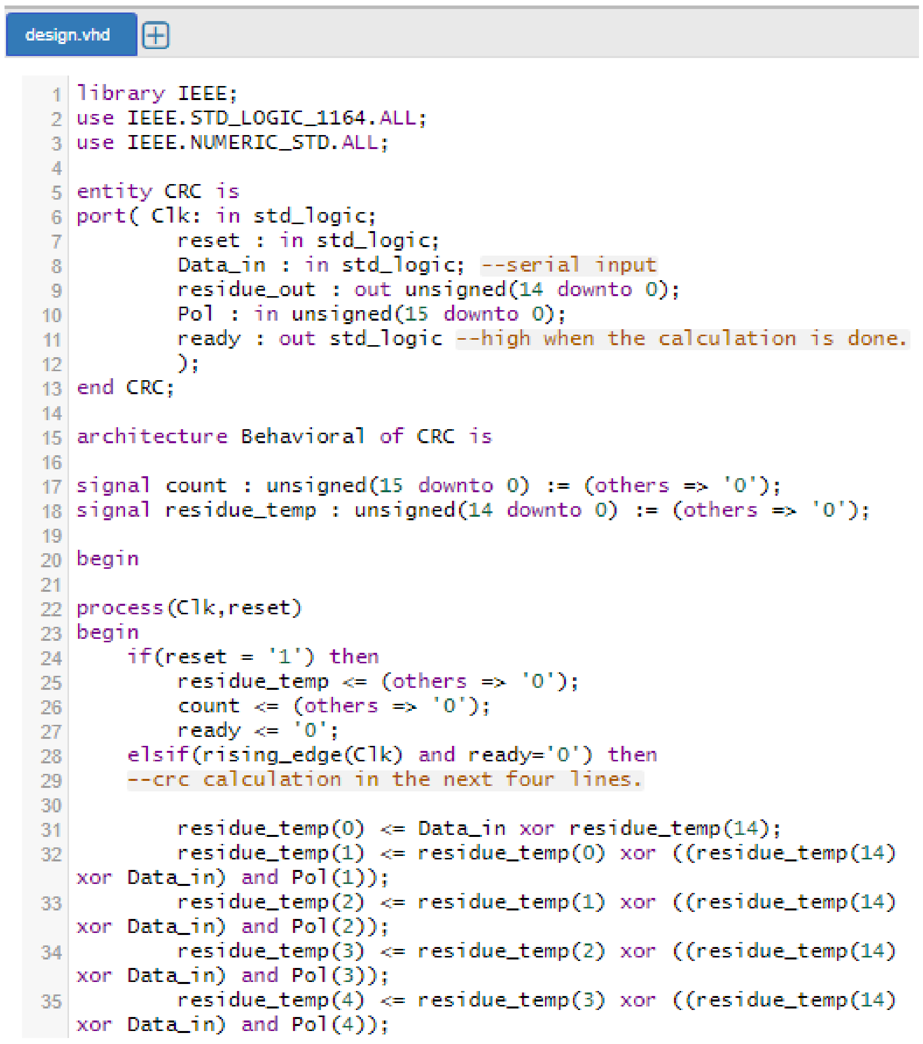


```vhdl
          Pol => Pol,
          ready => ready
        );

   -- Clock process definitions
process
   begin
        Clk <= '0';
        wait for Clk_period/2;
        Clk <= '1';
        wait for Clk_period/2;

   end process;

   -- Stimulus process
process
   begin
      reset <= '1';
      -- POLINOMIO
      Pol(0)  <= '1';
      Pol(1)  <= '0';
      Pol(2)  <= '0';
      Pol(3)  <= '1';
      Pol(4)  <= '1';
      Pol(5)  <= '0';
      Pol(6)  <= '0';
      Pol(7)  <= '1';
      Pol(8)  <= '1';
      Pol(9)  <= '0';
      Pol(10)  <= '1';
      Pol(11)  <= '0';
      Pol(12)  <= '0';
      Pol(13)  <= '0';
      Pol(14)  <= '1';
      Pol(15)  <= '1';

        wait for 100 ns;


        wait until falling_edge(Clk);
```

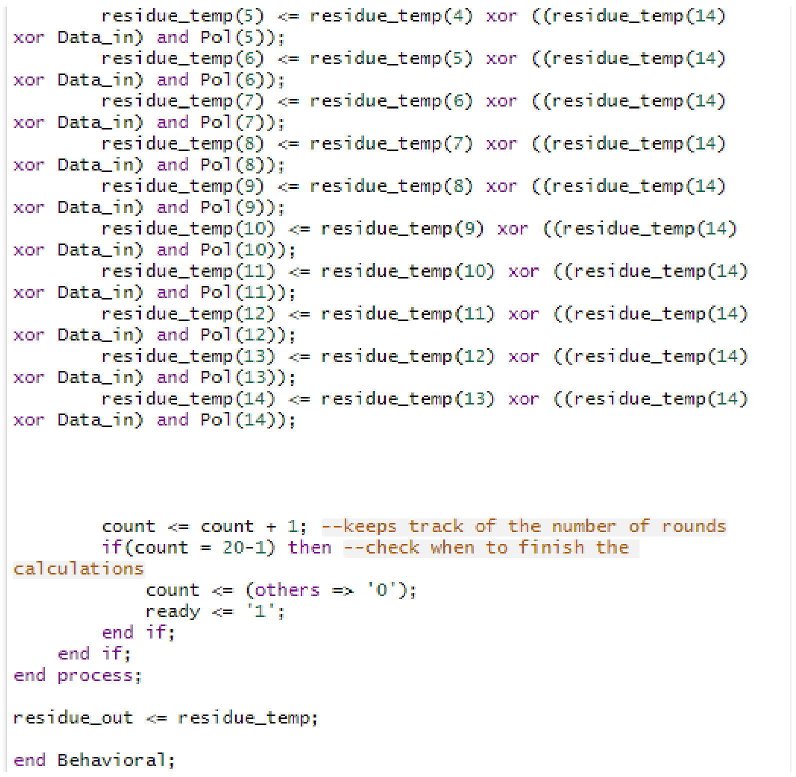


```
        --DATA IN
        reset <= '0';
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;
        Data_in <= '1'; wait for Clk_period;
        Data_in <= '0'; wait for Clk_period;

        wait until ready = '1'; wait for Clk_period;
        reset <= '1';


      wait;
   end process;

END;
```

Figure 4. Testbench VHDL code.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CRC is
port( Clk: in std_logic;
        reset : in std_logic;
        Data_in : in std_logic; --serial input
        residue_out : out unsigned(14 downto 0);
        Pol : in unsigned(15 downto 0);
        ready : out std_logic --high when the calculation is done.
        );
end CRC;

architecture Behavioral of CRC is

signal count : unsigned(15 downto 0) := (others => '0');
signal residue_temp : unsigned(14 downto 0) := (others => '0');

begin

process(Clk,reset)
begin
    if(reset = '1') then
        residue_temp <= (others => '0');
        count <= (others => '0');
        ready <= '0';
    elsif(rising_edge(Clk) and ready='0') then
    --crc calculation in the next four lines.

        residue_temp(0) <= Data_in xor residue_temp(14);
        residue_temp(1) <= residue_temp(0) xor ((residue_temp(14)
xor Data_in) and Pol(1));
        residue_temp(2) <= residue_temp(1) xor ((residue_temp(14)
xor Data_in) and Pol(2));
        residue_temp(3) <= residue_temp(2) xor ((residue_temp(14)
xor Data_in) and Pol(3));
        residue_temp(4) <= residue_temp(3) xor ((residue_temp(14)
xor Data_in) and Pol(4));
```



```
        residue_temp(5) <= residue_temp(4) xor ((residue_temp(14)
xor Data_in) and Pol(5));
        residue_temp(6) <= residue_temp(5) xor ((residue_temp(14)
xor Data_in) and Pol(6));
        residue_temp(7) <= residue_temp(6) xor ((residue_temp(14)
xor Data_in) and Pol(7));
        residue_temp(8) <= residue_temp(7) xor ((residue_temp(14)
xor Data_in) and Pol(8));
        residue_temp(9) <= residue_temp(8) xor ((residue_temp(14)
xor Data_in) and Pol(9));
        residue_temp(10) <= residue_temp(9) xor ((residue_temp(14)
xor Data_in) and Pol(10));
        residue_temp(11) <= residue_temp(10) xor ((residue_temp(14)
xor Data_in) and Pol(11));
        residue_temp(12) <= residue_temp(11) xor ((residue_temp(14)
xor Data_in) and Pol(12));
        residue_temp(13) <= residue_temp(12) xor ((residue_temp(14)
xor Data_in) and Pol(13));
        residue_temp(14) <= residue_temp(13) xor ((residue_temp(14)
xor Data_in) and Pol(14));



        count <= count + 1; --keeps track of the number of rounds
        if(count = 20-1) then --check when to finish the
calculations
            count <= (others => '0');
            ready <= '1';
        end if;
    end if;
end process;

residue_out <= residue_temp;

end Behavioral;
```

Figure 4.1. Design VHDL code.

Results

- **Exercise 1 (CAN Bus Access Logic)**

Figure 5, shows the resultant circuit, while *figures 5.1* and *5.2* show the CAN Bus level when transistors output a recessive and dominant level, and figures *5.3*, *5.4, 5.5 and 5.6* show the different cases able to be obtained by the reaction of the CAN node for each sender.

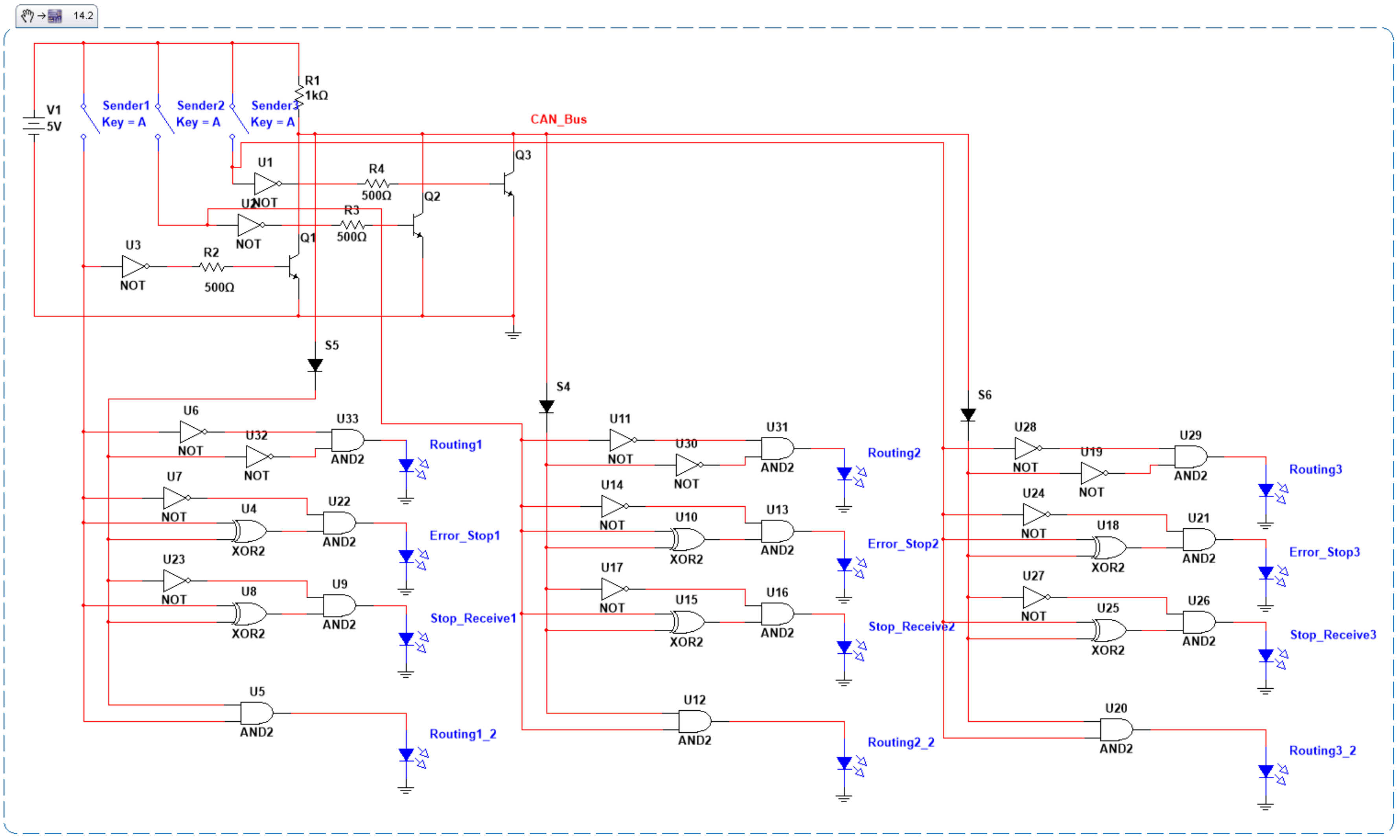


Figure 5. Resultant circuit to simulate the CAN Bus Access Logic.

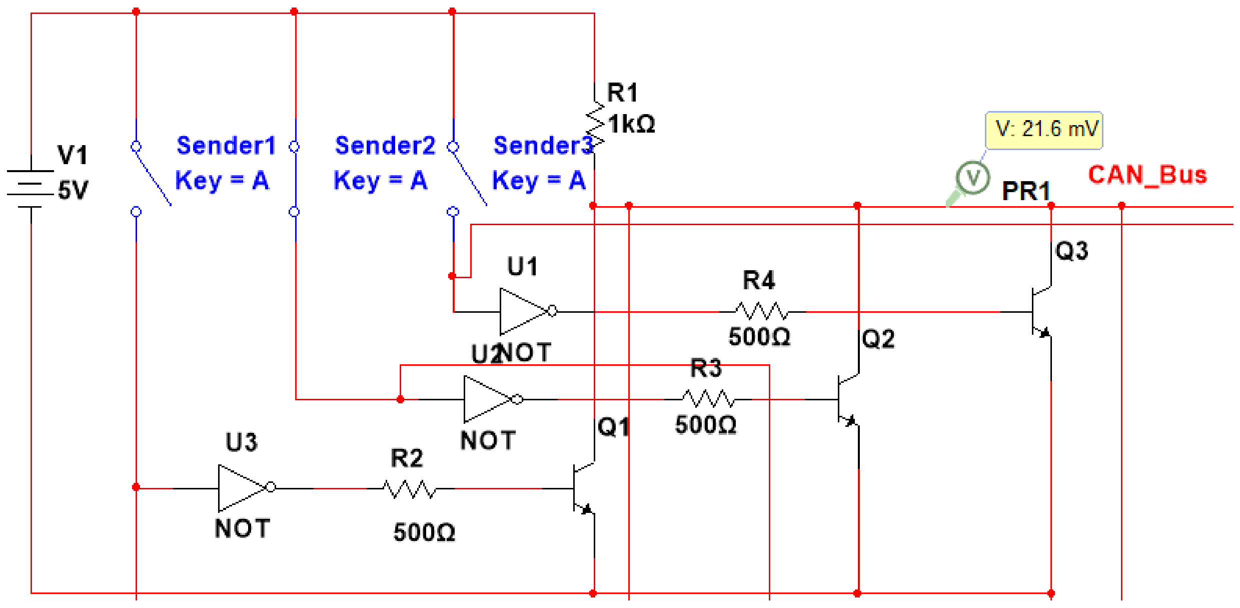


Figure 5.1. CAN Bus with dominant level (21.6 mV) when any number of transistors output a dominant level.

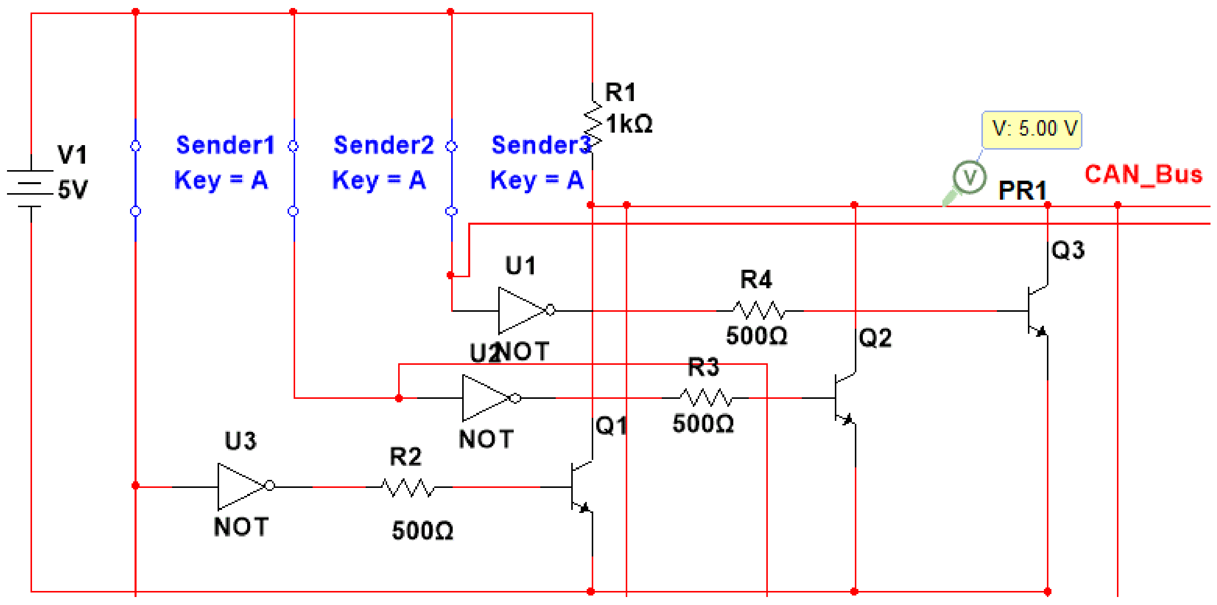


Figure 5.2. CAN Bus with recessive level (5 V) when all transistors output a recessive level.

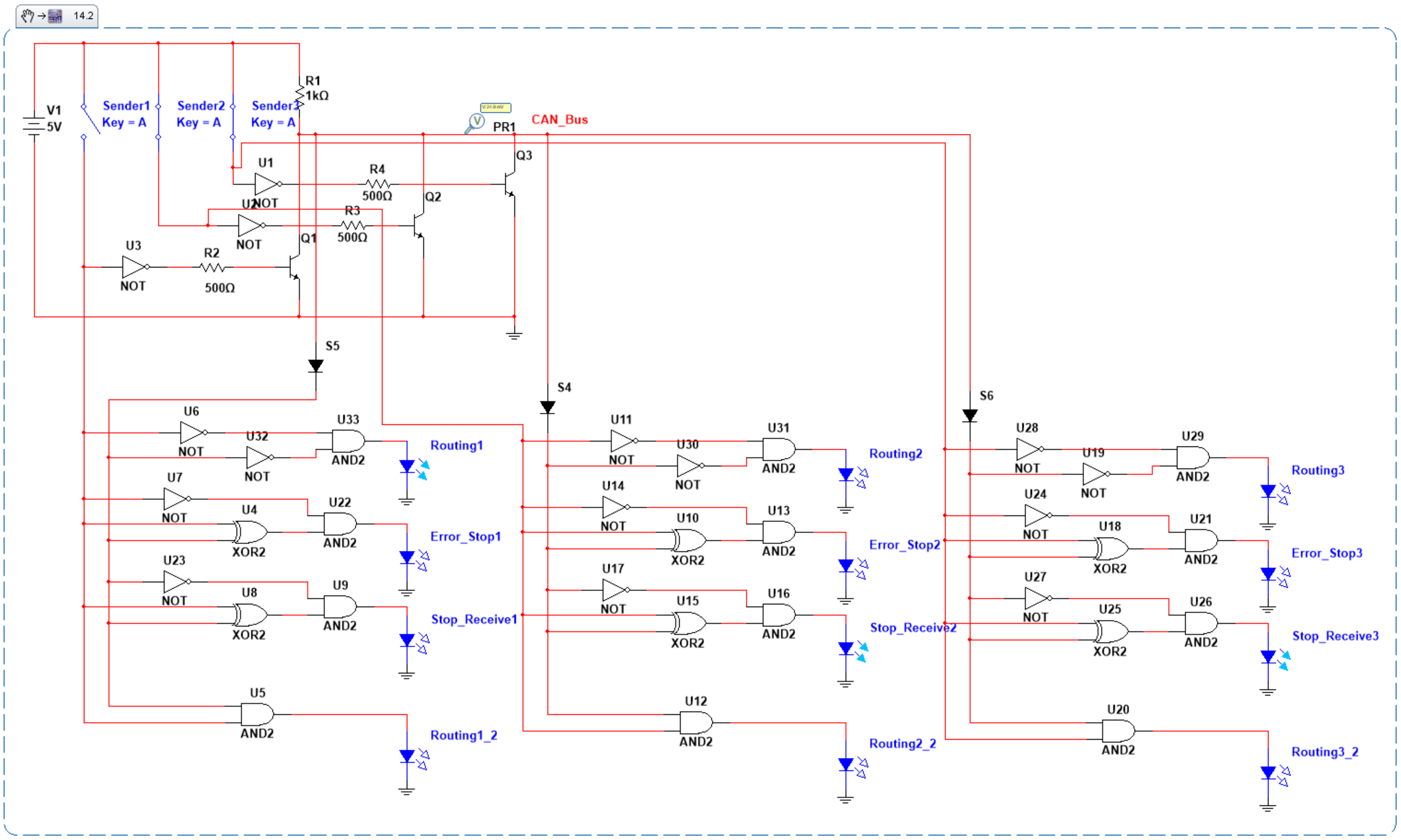


Figure 5.3. Reaction of CAN nodes with Bus level: 0, where Node 1: Routing (Send level:0), Node 2: Stop: Receive (Send level:1) and Node 3: Stop: Receive (Send level:1).

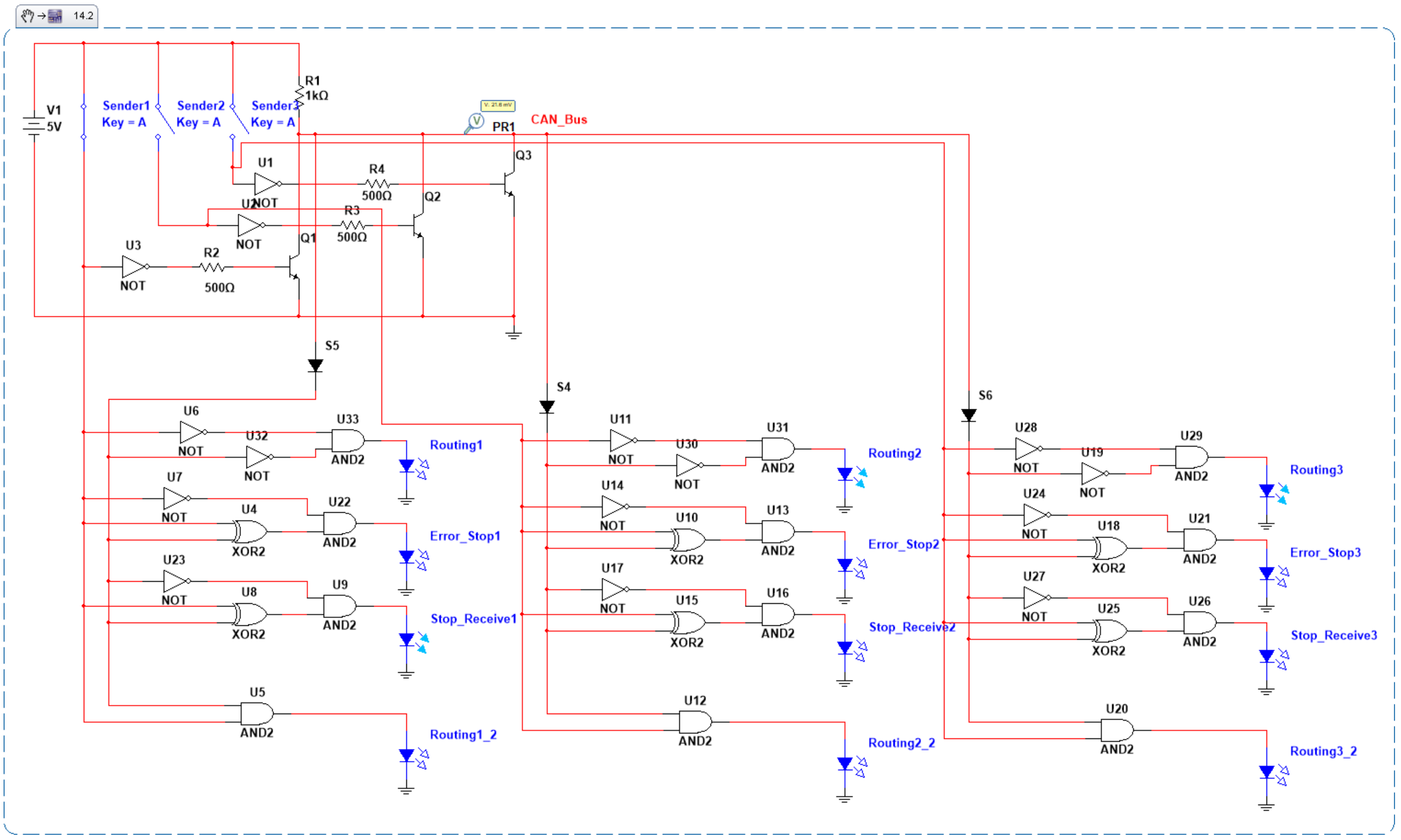


Figure 5.4. Reaction of CAN nodes with Bus level: 0, where Node 1: Stop: Receive (Send level:1), Node 2: Routing (Send level:0) and Node 3: Routing (Send level:0).

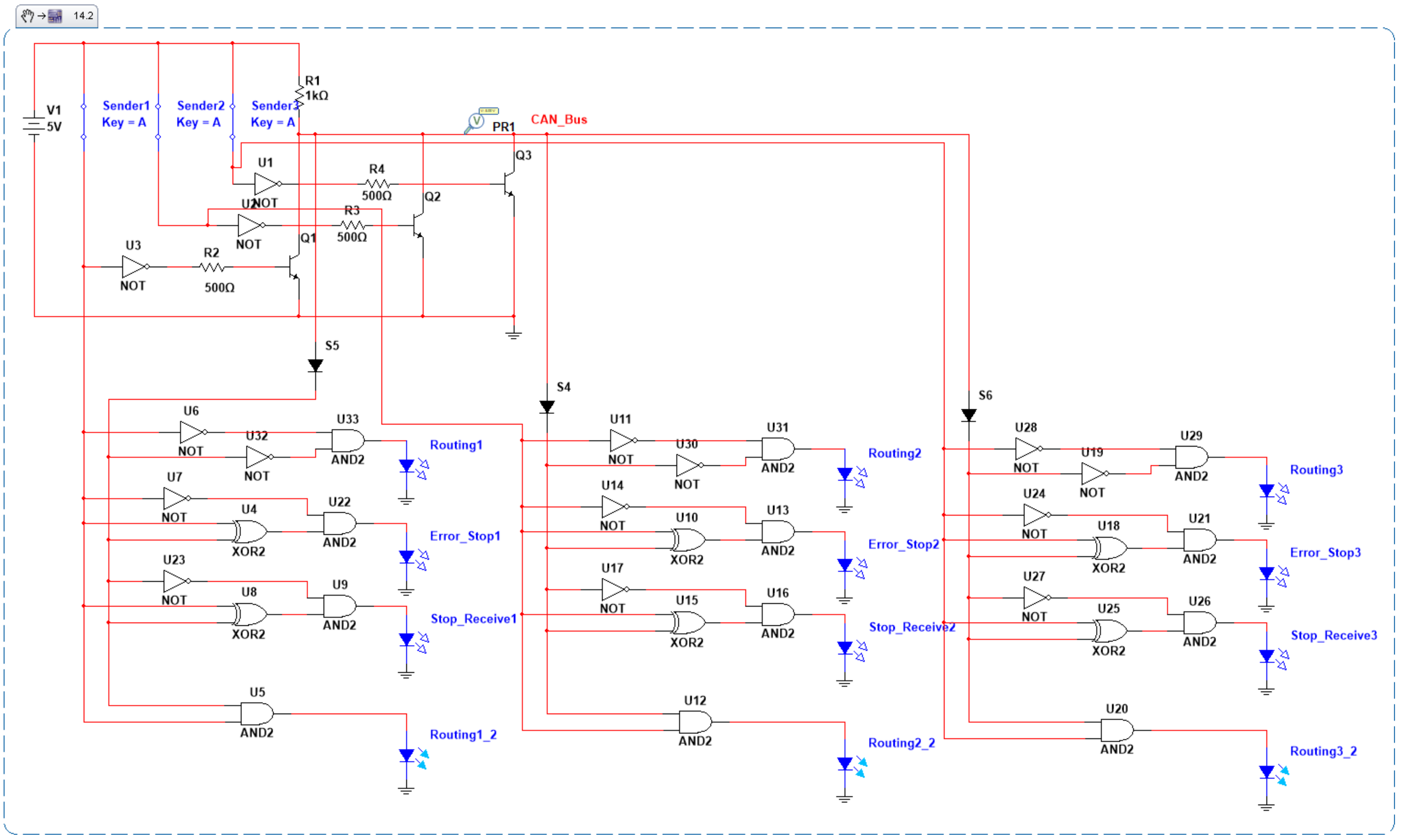


Figure 5.5. Reaction of CAN nodes with Bus level: 1, where Node 1: Routing (Send level:1), Node 2: Routing (Send level:1) and Node 3: Routing (Send level:1).

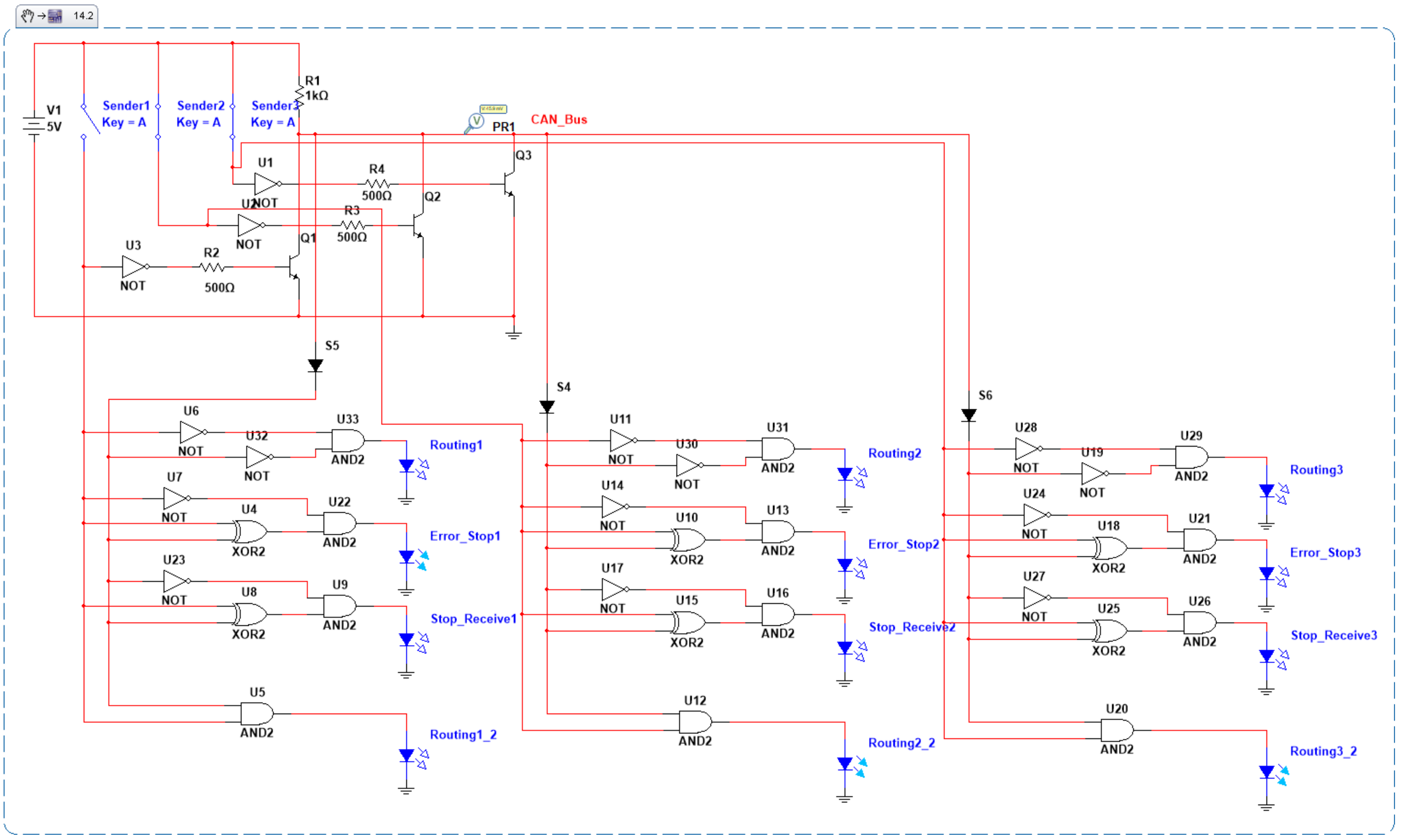


Figure 5.6. Reaction of CAN nodes with Bus level: 1, where Node 1: Error: Stop (Send level:0), Node 2: Routing (Send level:1) and Node 3: Routing (Send level:1).

- **Exercise 2 (Calculate R (residual or frame check sequence) for use in CRC)**

The following *figure 6* shows the results obtained from calculating the remainder “R” given “D” and “G”.

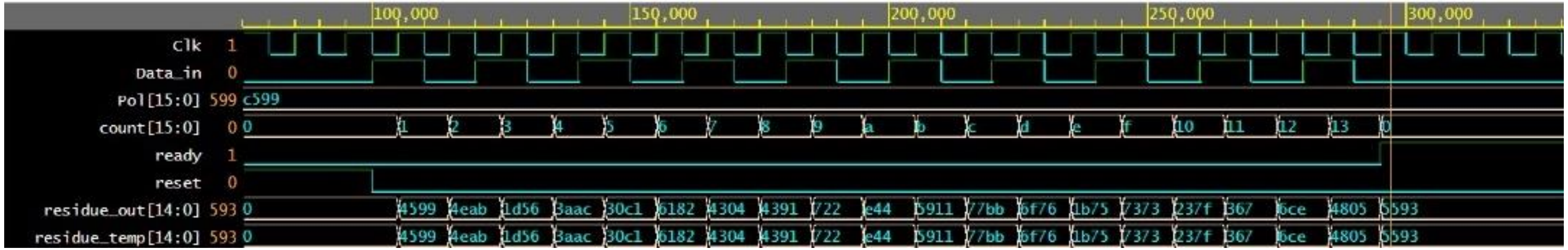

100,000
150,000
200,000
250,000
300,000
Clk 1
Data_in 0
Pol[15:0] 599 c599
count[15:0] 0 0 1 2 3 4 5 6 7 8 9 a b c d e f 10 11 12 13 0
ready 1
reset 0
residue_out[14:0] 593 0 4599 4eab 1d56 3aac 30c1 6182 4304 4391 722 e44 5911 77bb 6f76 1b75 7373 237f 367 6ce 4805 5593
residue_temp[14:0] 593 0 4599 4eab 1d56 3aac 30c1 6182 4304 4391 722 e44 5911 77bb 6f76 1b75 7373 237f 367 6ce 4805 5593

Figure 6. Calculation of “R” (5593) in VHDL.

Conclusion

After completing the exercises, one thing was observed. In the case of the exercise 1, out of the 4 reaction CAN node cases, the last “Error: Stop” reaction was only able to be reached when the send level was a 0 and the CAN Bus level a 1 and as the problem itself stated, the only way the CAN Bus level could be at a high level (with a value of 1) was when all the nodes output a high level as well, so, this way, it was not possible to have a CAN Bus level 1 when one of the node’s output is 0 (which as a consequence would generate the CAN Bus level to be 0 as well), However, after playing around with the simulator a bit, it was noticed that for a quite small amount of time, the “Error: Stop” reaction was reached in one sender, while the other senders remained in the “Routing” reaction, so, this way, only for that small period of time, the “Error: Stop” reaction was able to be reached, meaning that the CAN Bus was at a high level for a small period of time before the other node’s dominant level would turn the CAN Bus level into a dominant level as well.

Also, we came across some doubts and confusions which in the beginning made it impossible to finish the exercises (such as the one just mentioned above for exercise 1) and, getting the right logic to obtain the desired remainder for exercise 2, however, all of these confusions were overcome and the solution to both exercises was achieved.

This way, our knowledge about the CAN Bus and how CRC works, was reinforced and we were able to see from another perspective (Multisim circuit and VHDL implementation) how both of these concepts work.

Bibliography

Stallings, W. (2006). Memberfiles (8th ed.). Pearson Education Inc. Retrieved March 9, 2022, from https://memberfiles.freewebs.com/00/88/103568800/documents/Data.And.Computer.Communications.8e.WilliamStallings.pdf.

VECTOR. (2021). CAN_E: Bitwise Bus Arbitration. Retrieved March 8, 2022, from VECTOR E-Learning website: https://elearning.vector.com/mod/page/view.php?id=351