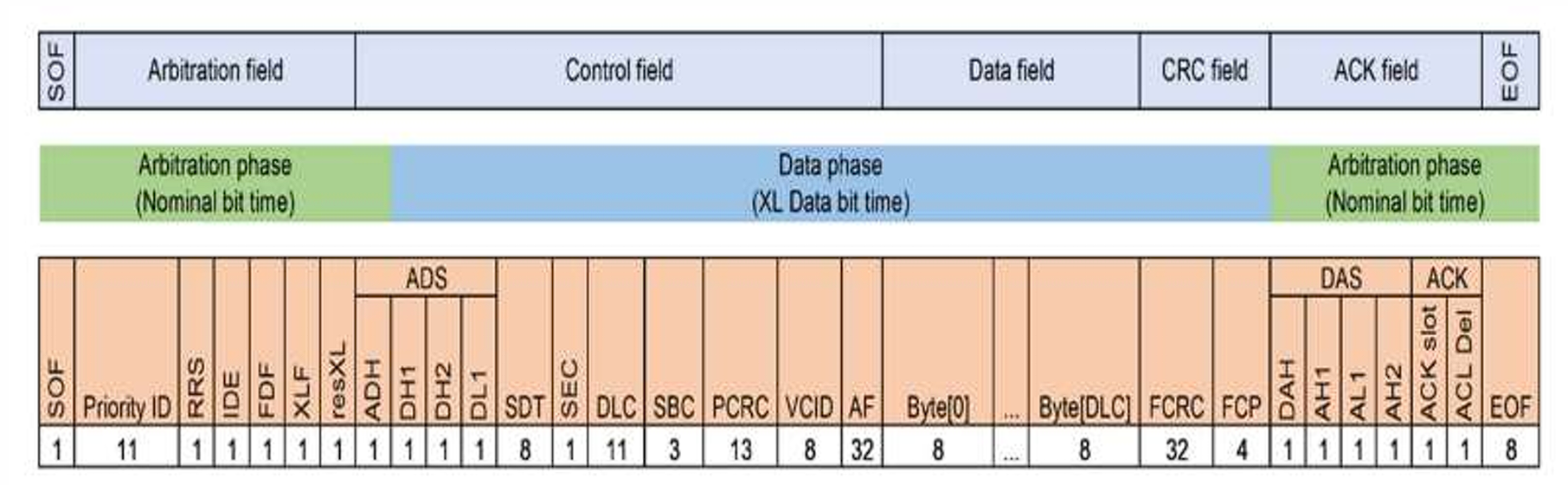
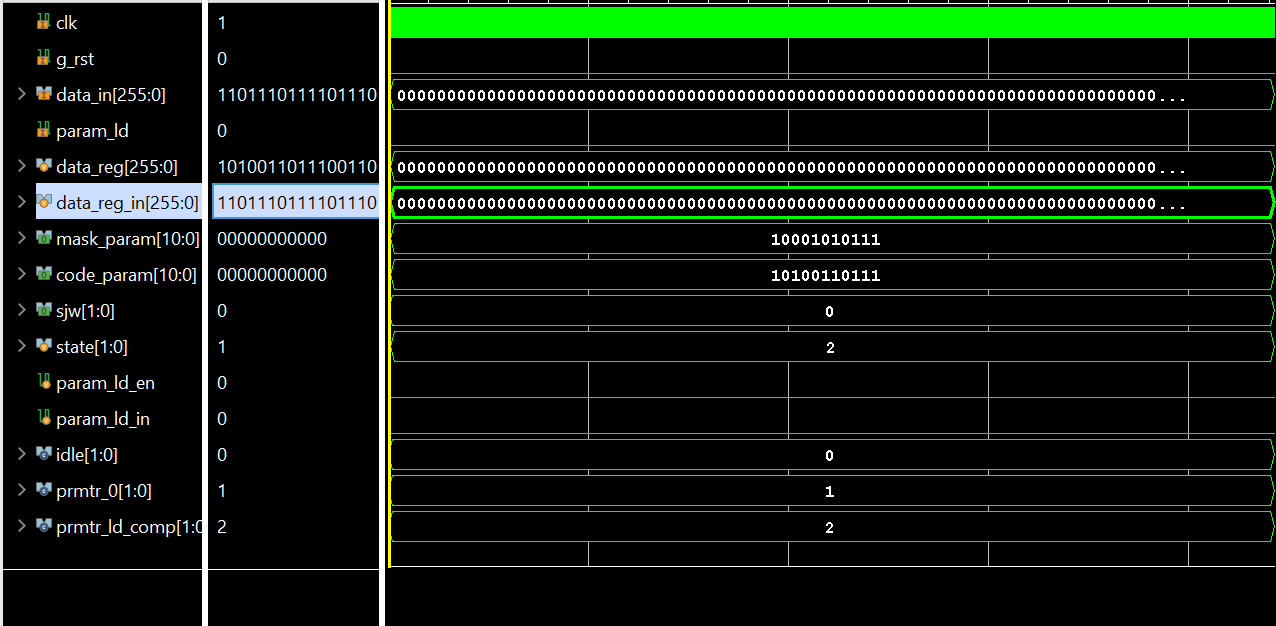
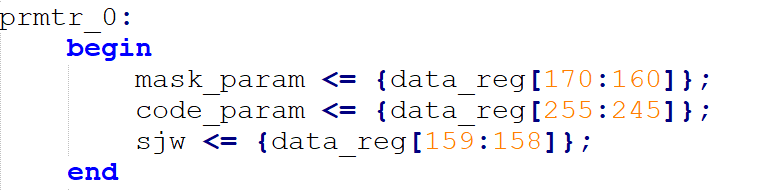
**CAN – XL with CANSec**

****

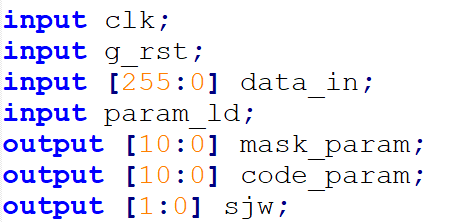
1. Registry :

The registry in the controller stores essential parameters such as the code parameter, mask parameter, and re-synchronization jump width (SJW), which are specified by the host CPU. These parameters play a critical role in message acceptance and synchronization. The code and mask parameters are used to filter and determine the acceptance of incoming messages, ensuring that only relevant data is processed.





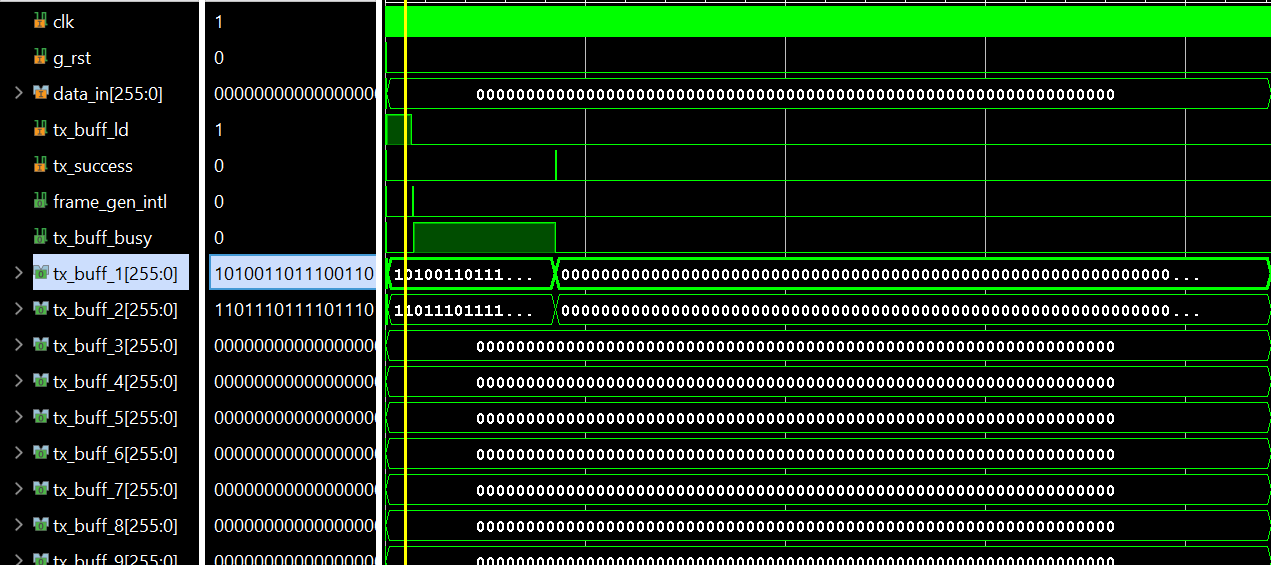
In the above image of the code it shows that the code parameter gets the value of “priority id” and the mask parameter gets the first 11 bits of “AF” (Arbitration Field).

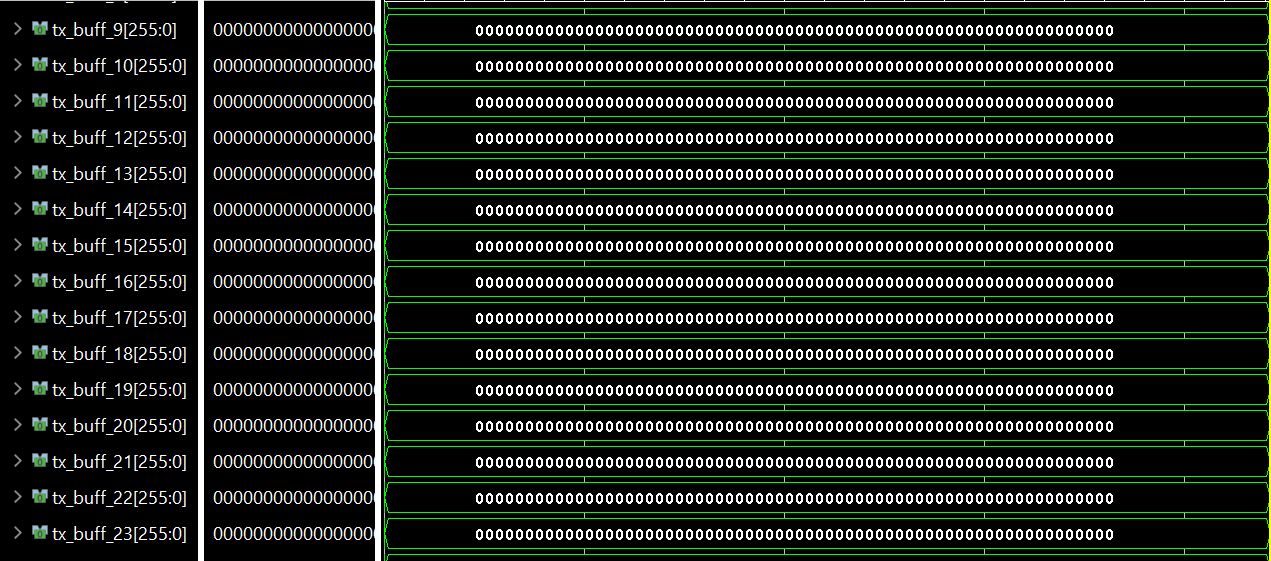


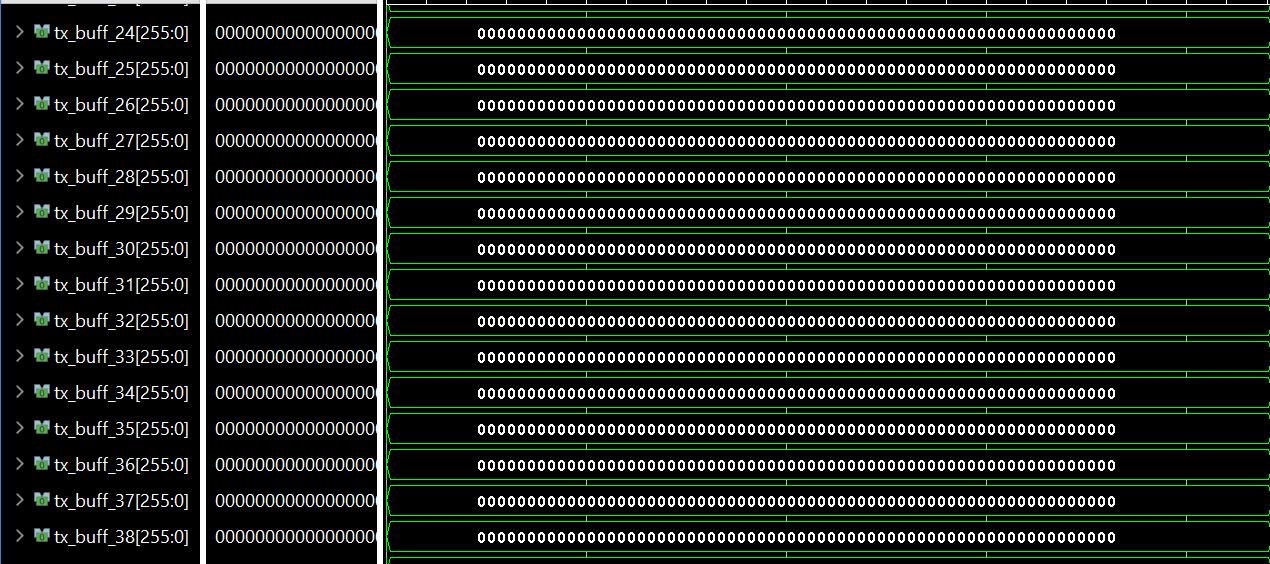
The “param\_ld” signal initiates the process of transferring these parameters from the host CPU to the parameter registers within the controller and this signal is given in testbench to simulate CPU.

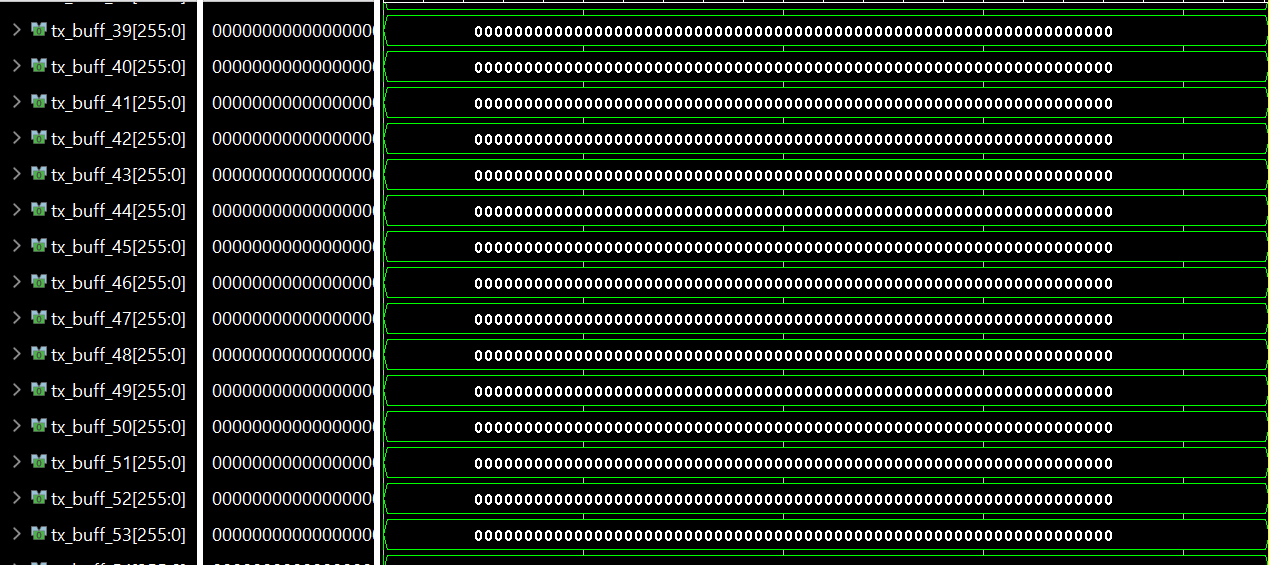
2. Transmission Buffer :

The transmission buffer in the controller temporarily stores data to be sent across the CAN network. It acts as an intermediary between the host CPU and the CAN bus, holding the message until it is processed for transmission. The buffer is capable of accepting data up to 256 bits and the size of the buffer is limited to 65 including buffer “0”.

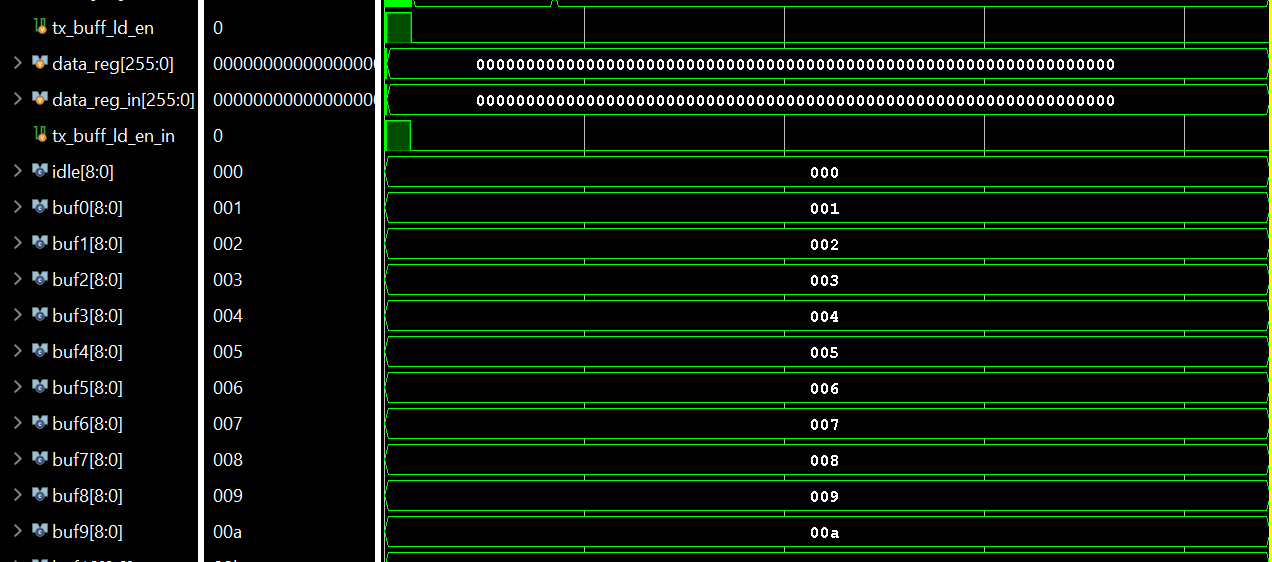


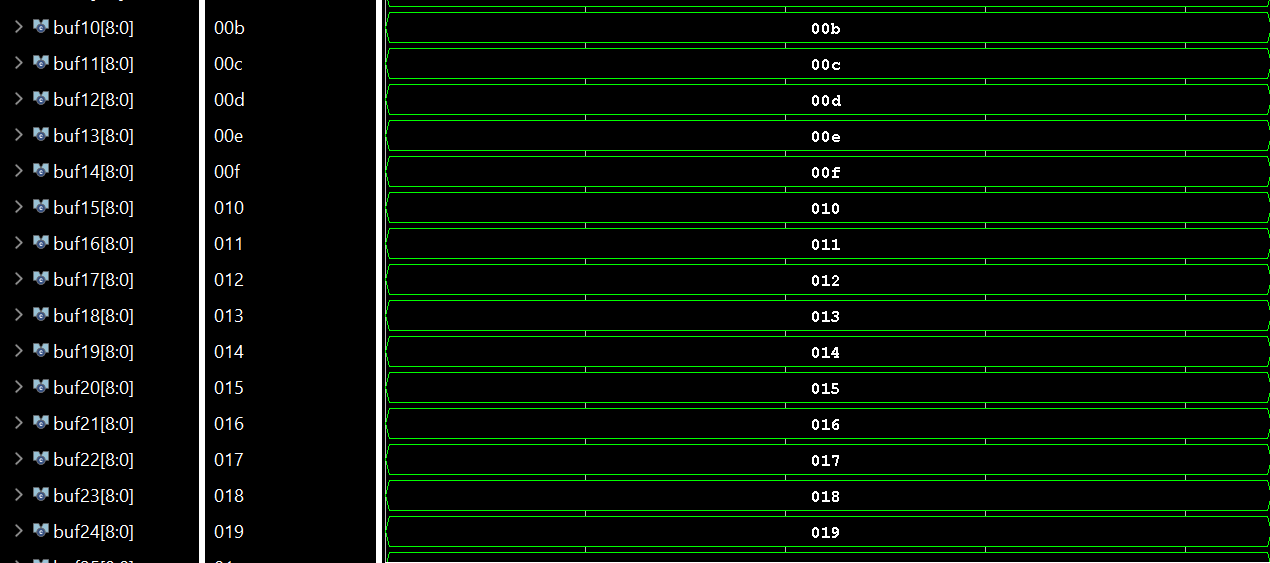


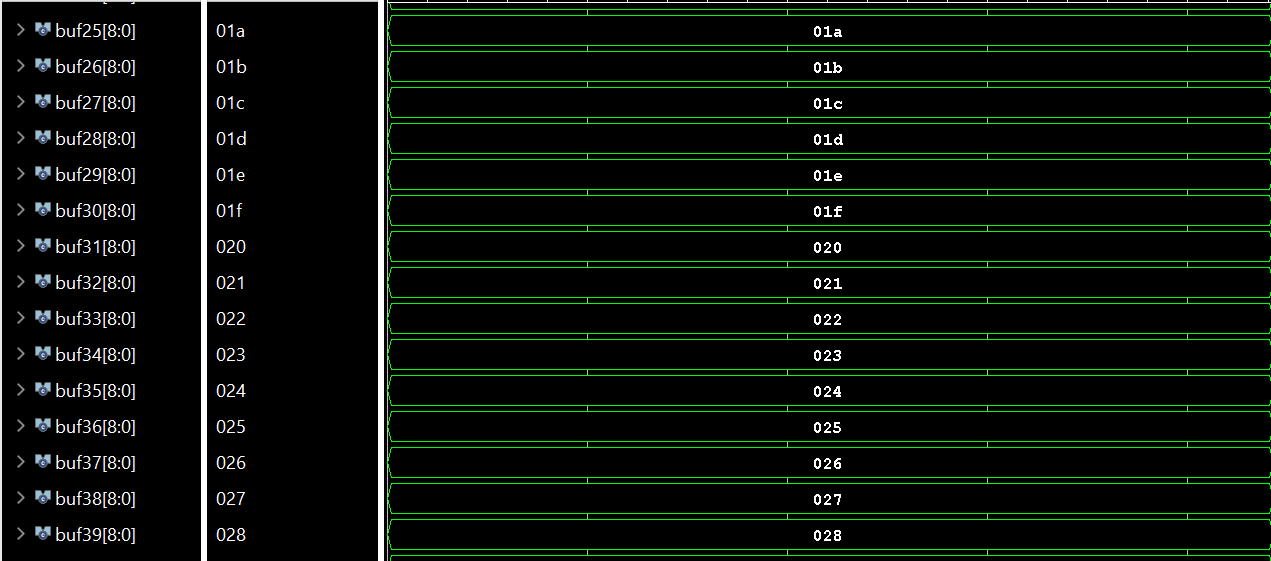


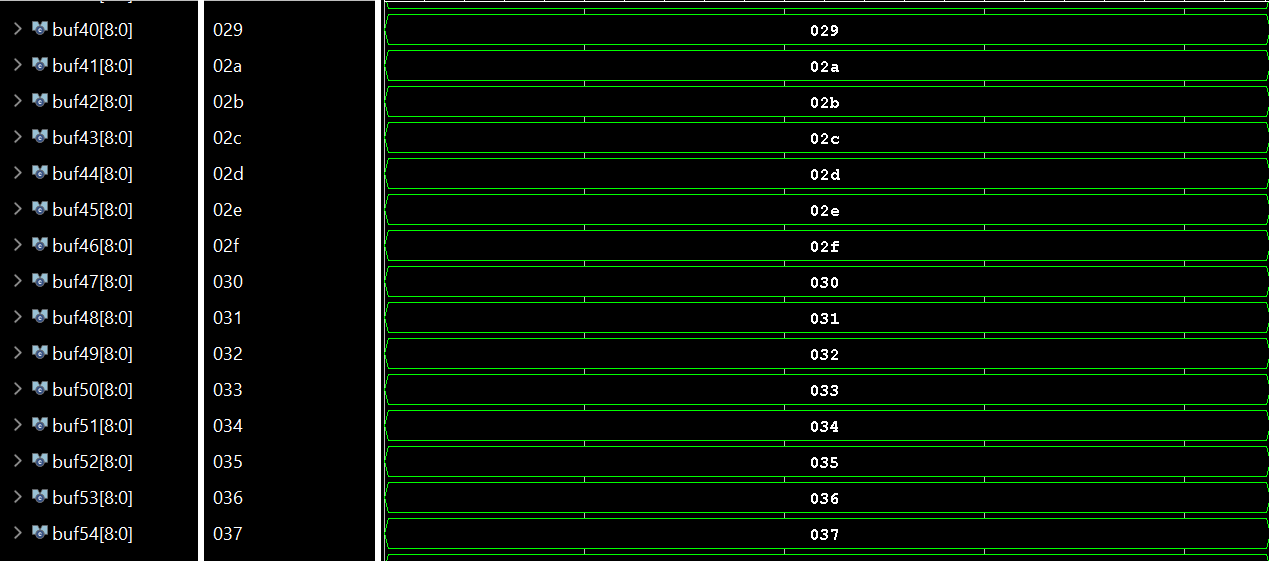


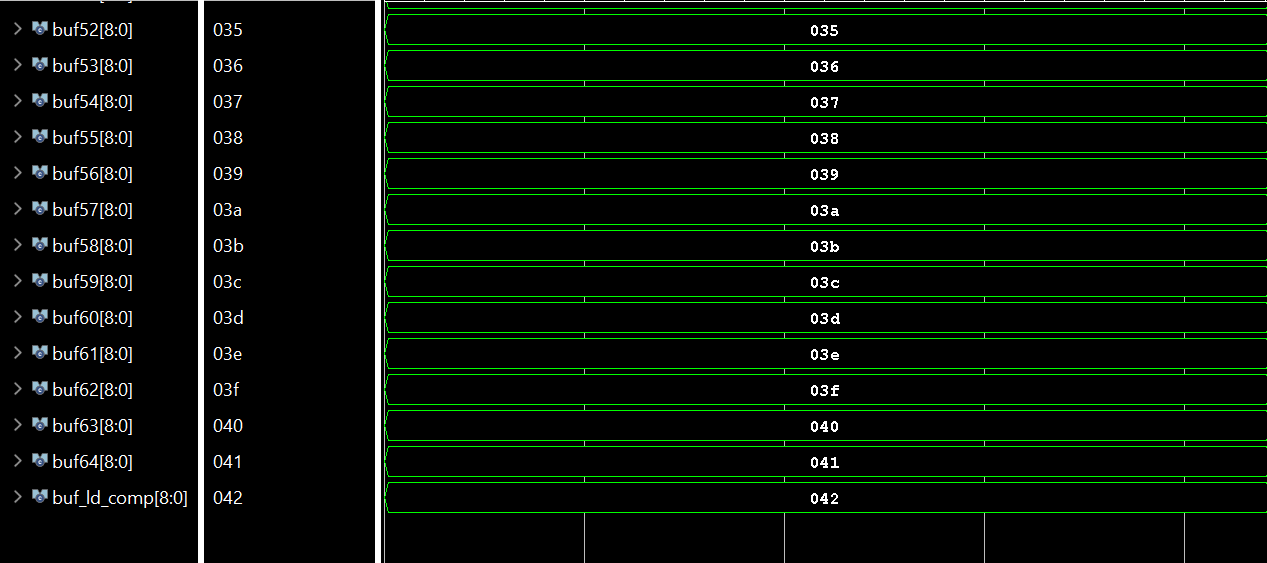




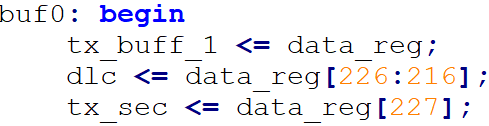






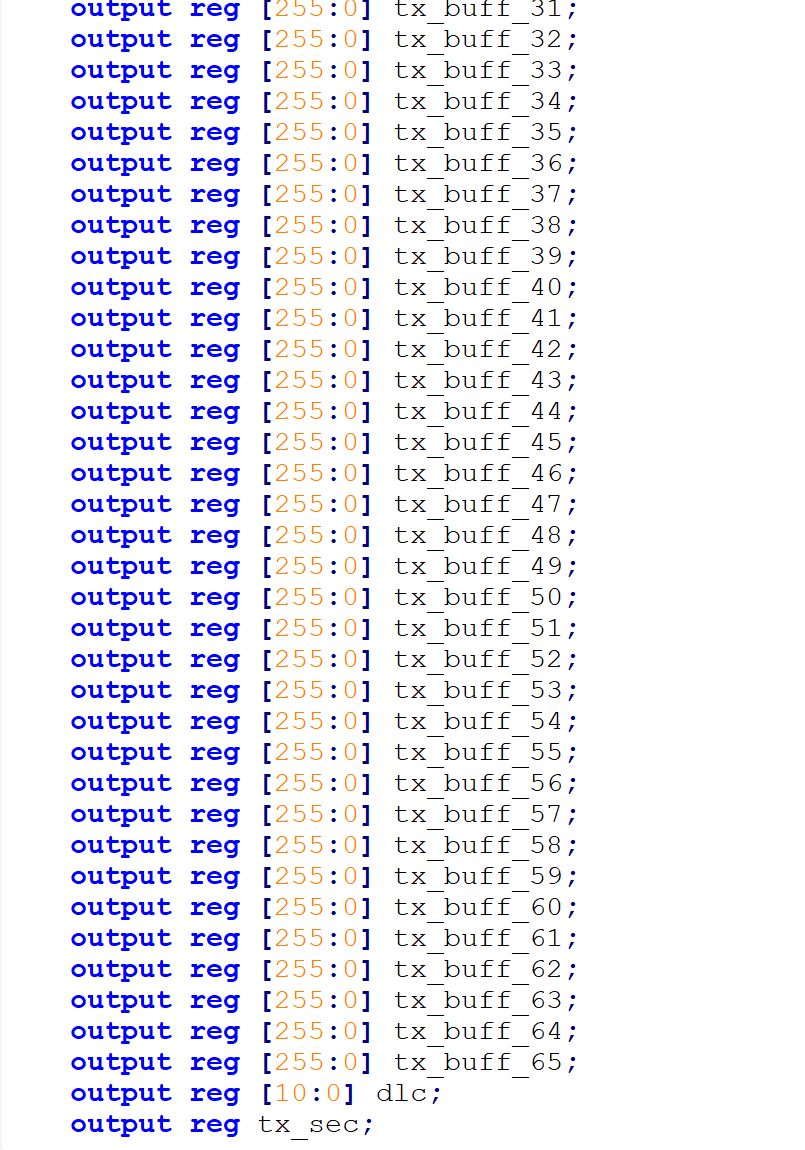


The above images show the various buffers and the data that will be stored in them. In the images only two buffers are full as in this node only 333 bits are being sent.



The above code shows the allocation of data form testbench to buffers and also the value of “DLC” to determine the size of payload and “tx\_sec” to indicate if the CAN-XL Frame is a secure frame or not.





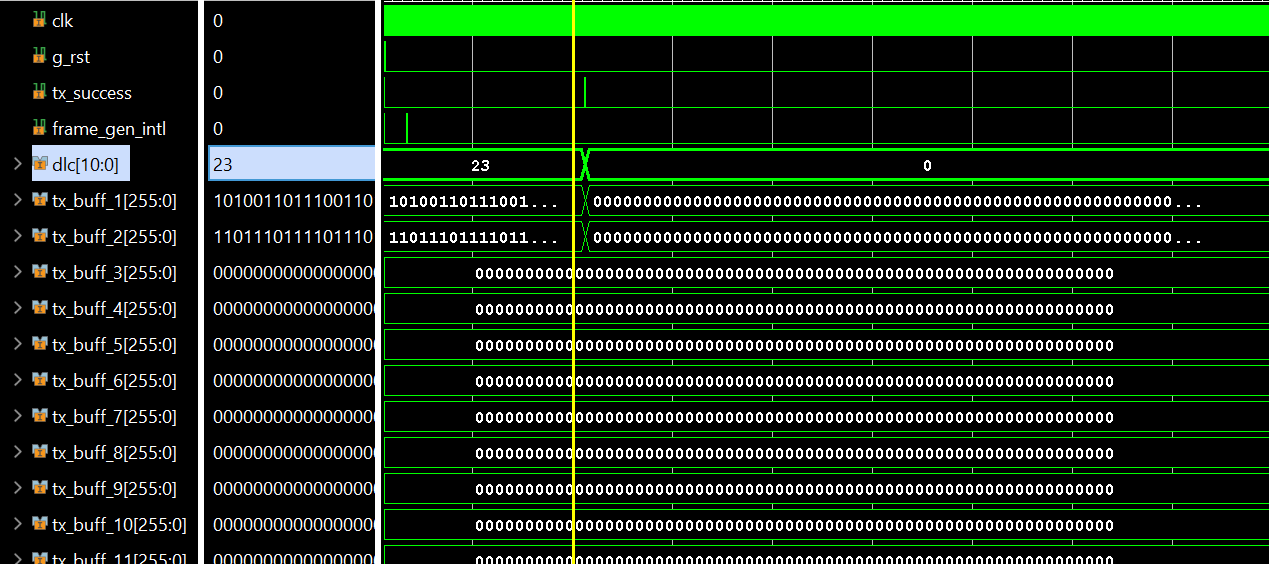
The signal “tx\_buff\_ld” is used to load data from the host CPU into the transmission buffer. When asserted, it triggers the process of transferring data bytes from the host CPU to the controller's transmission buffer.

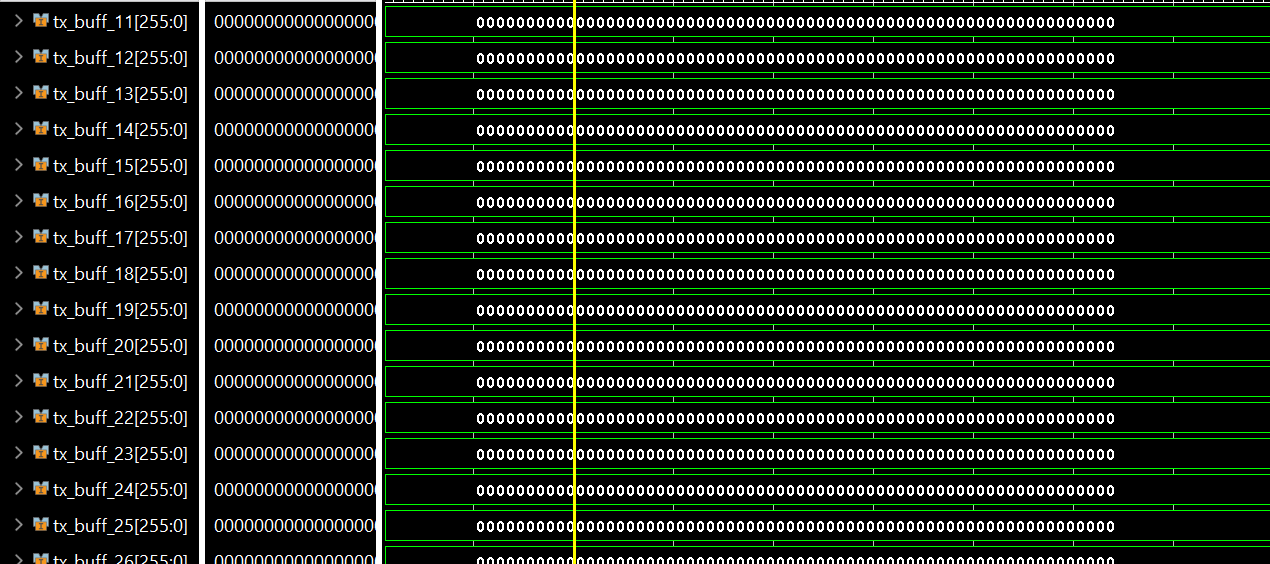
The signal “frame\_gen\_intl” starts the process of generating a frame from the data stored in the transmission buffer and this signal is only supposed to be “asserted only once in a frame”.

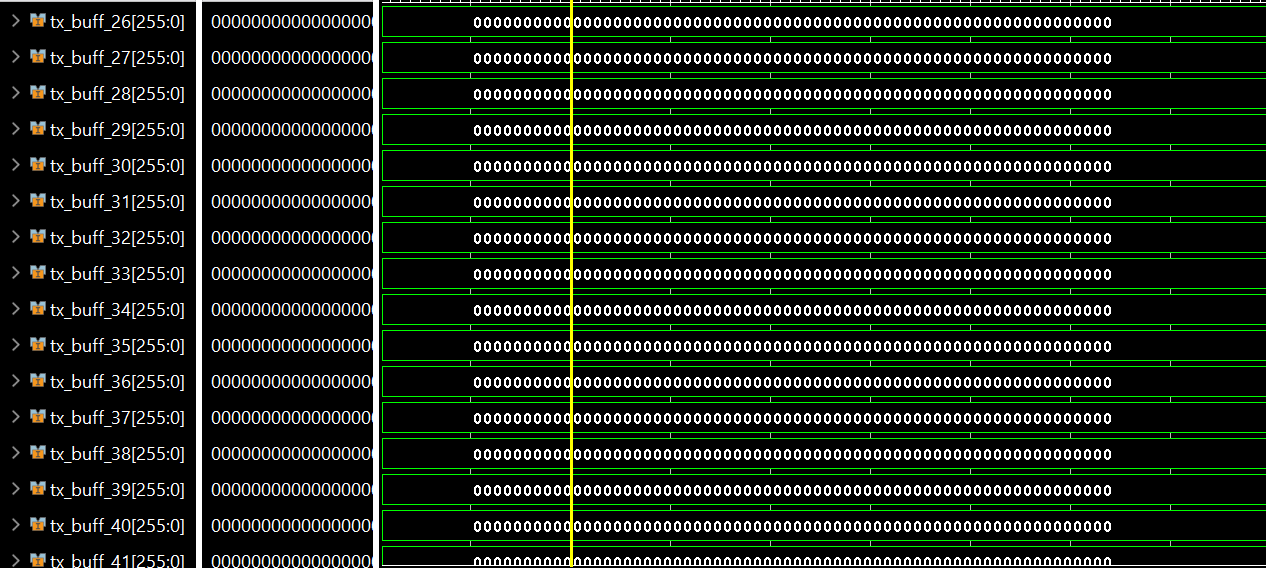
The signal “tx\_buff\_busy” indicates whether the transmission buffer is currently in use. When asserted, it signifies that the buffer is actively involved in the transmission process or is processing a loaded message.

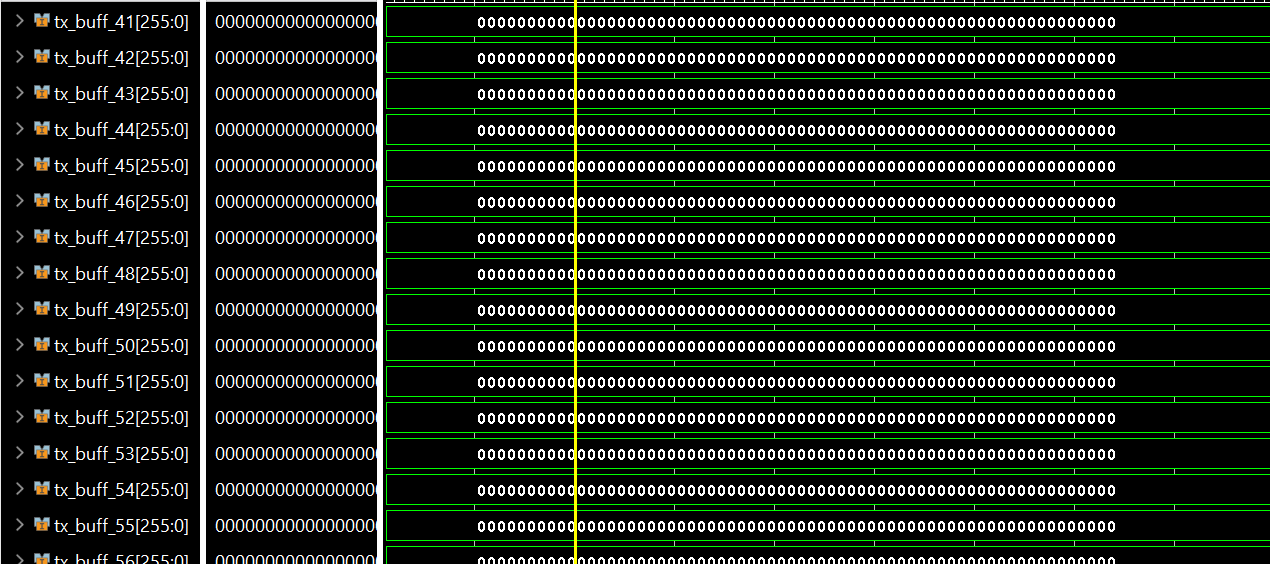
3. Data Frame Generator :

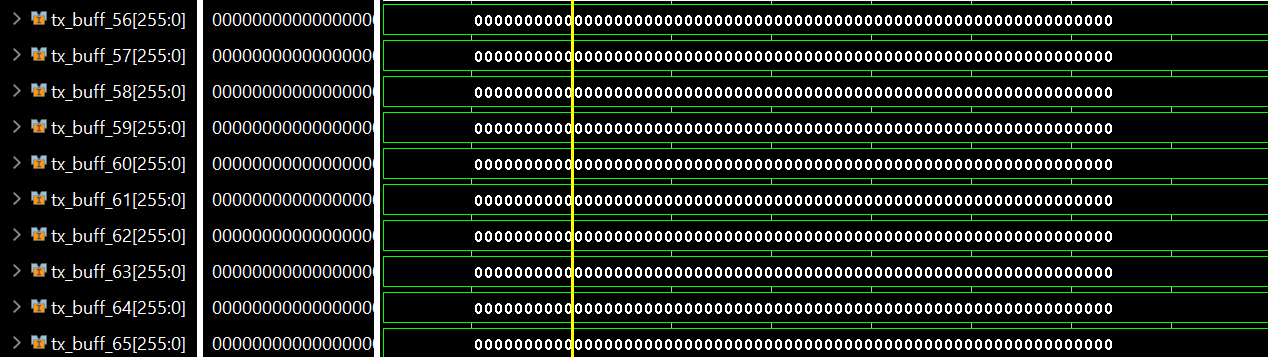
The Data Frame Generator in the controller is responsible for creating a complete CAN-XL data frame from the data stored in the transmission buffer. It assembles the frame by adding SOF (Start Of Frame),PCRC (Preface CRC) ,FCRC (Frame CRC) fields. It also enables the AES security code to work on the data provided by it.

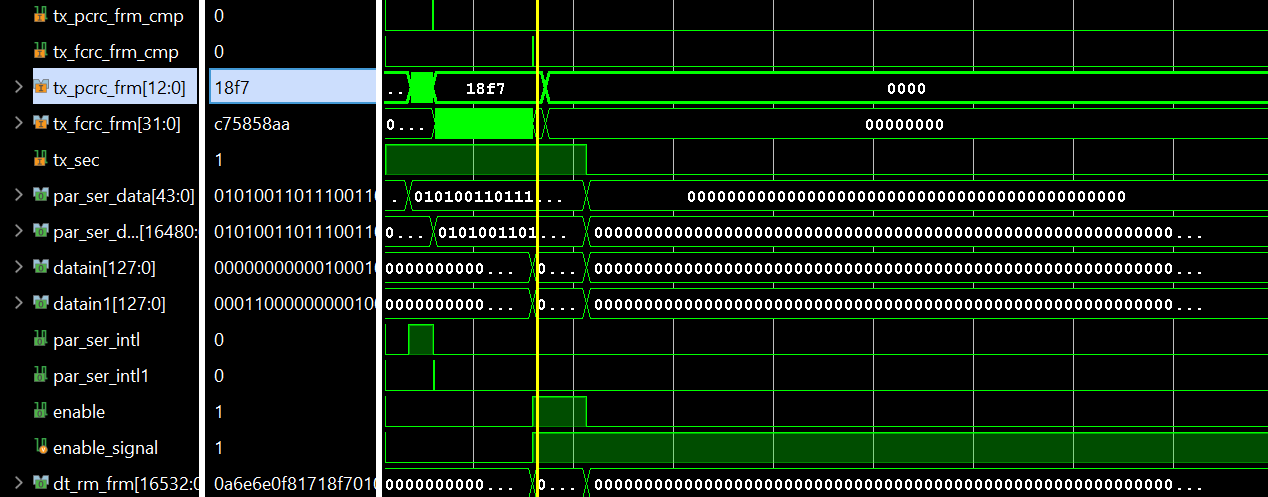


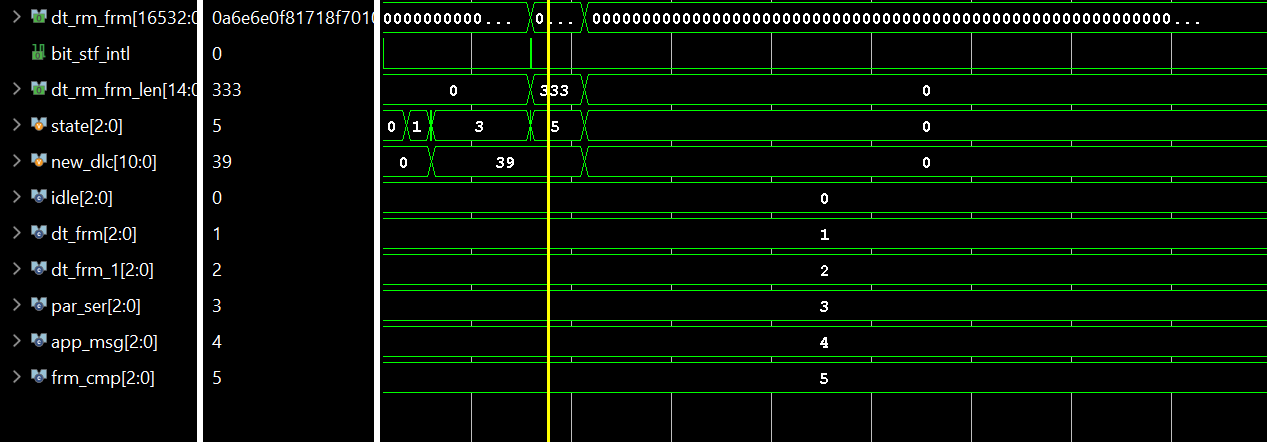






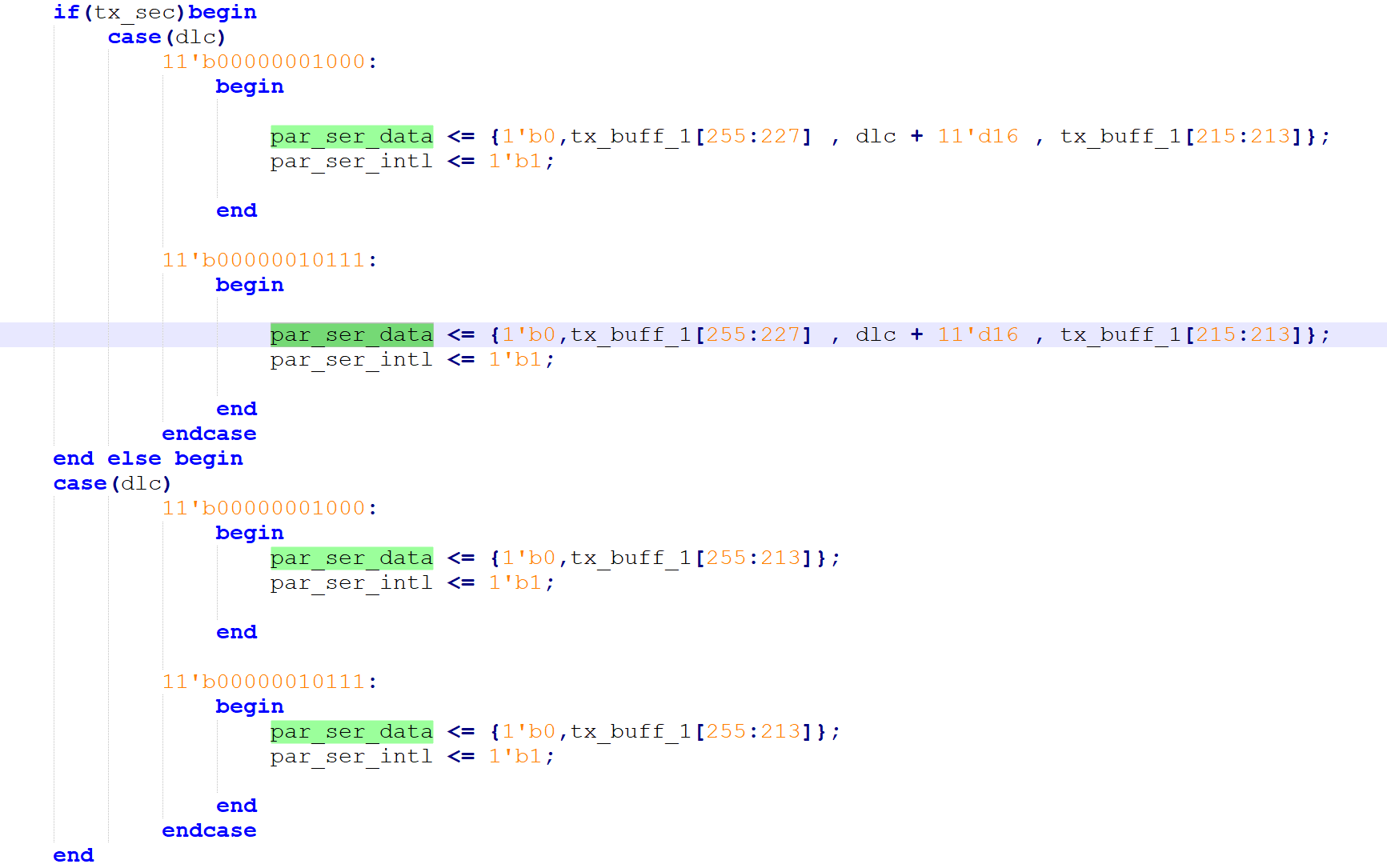




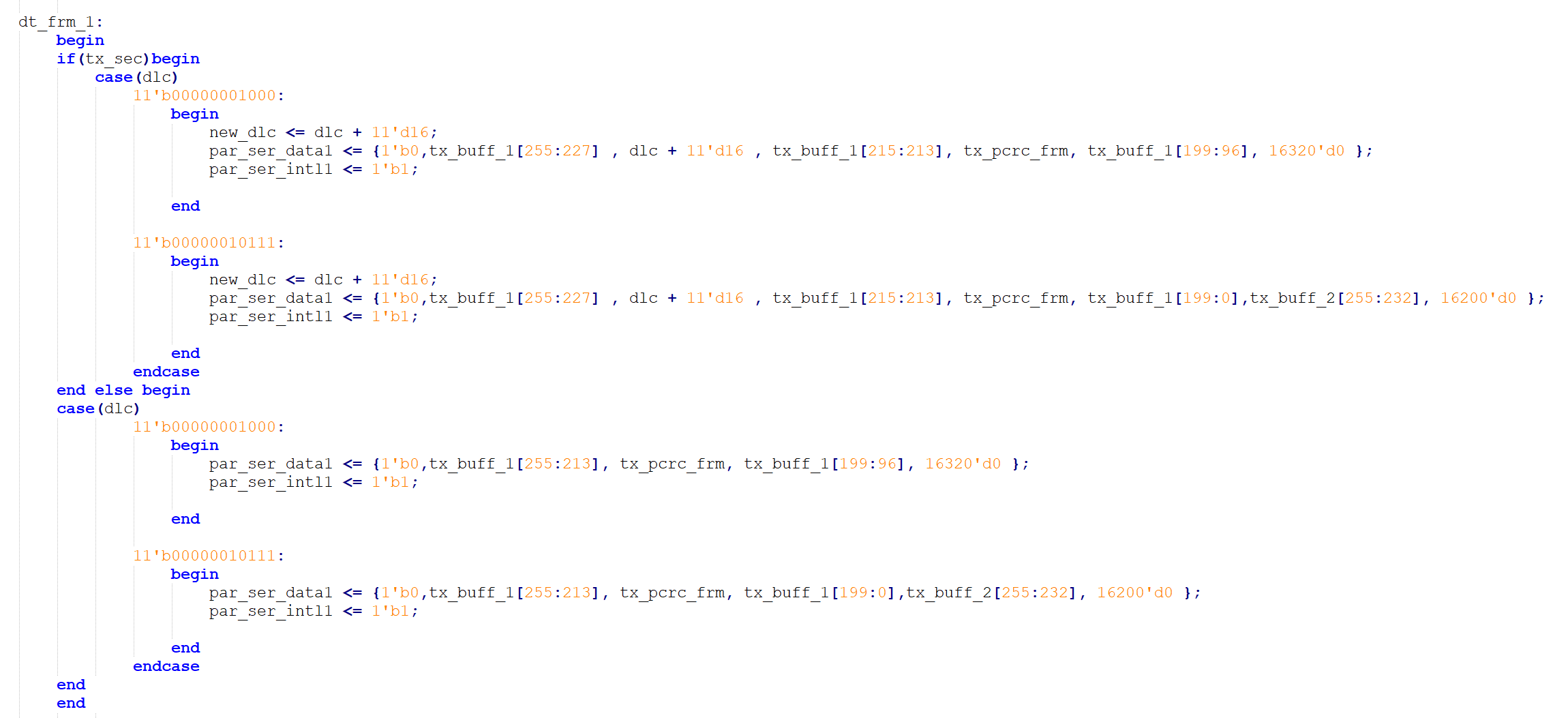


In the above image pcrc , fcrc values are shown , and the “par\_ser\_data” , “par\_ser\_data1” that are used to create them respectively and their initialisation signal “par\_ser\_intl” and “par\_ser\_intl1”.

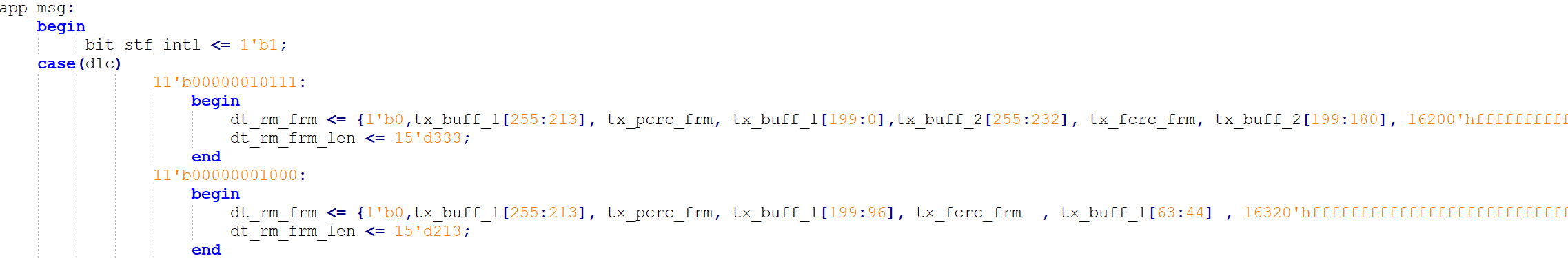
The signal “dt\_rm\_frm” will get its value after all CRC is calculated and this value will be the CAN\_XL frame. The signal “bit\_stf\_intl” signal initialises the bit stuffing module.



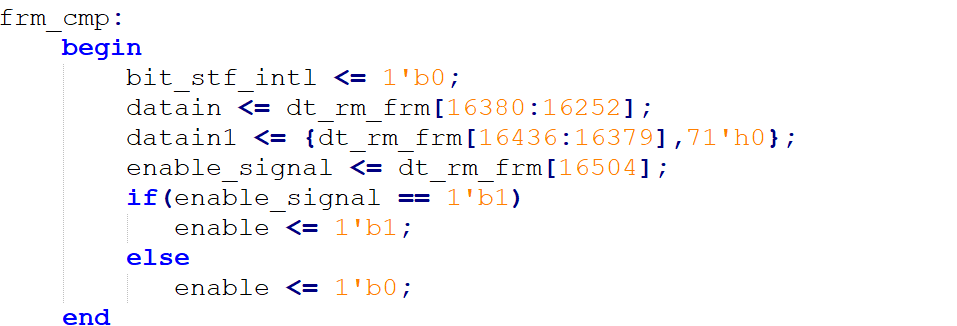
In the above code depending on if security signal is high then different ways “par\_ser\_data” which is used to calculate “PCRC” is created as in secure frames the payload will be appended with security header and ICV (Integrity Check value).



In the above code depending on if security signal is high then different ways “par\_ser\_data1” which is used to calculate “FCRC” is created as in secure frames the payload will be appended with security header and ICV (Integrity Check value).

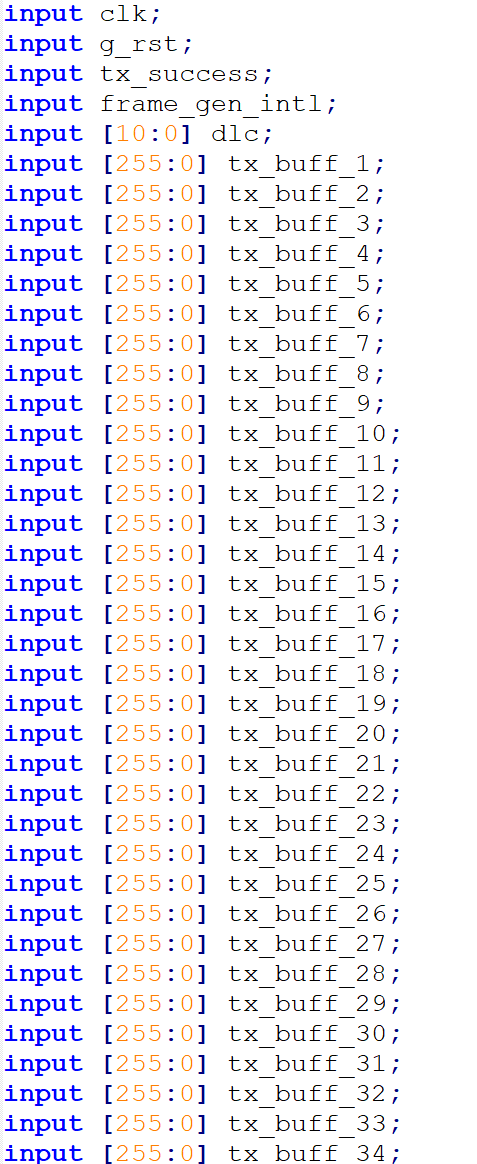


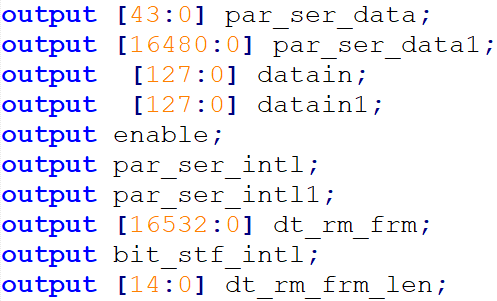
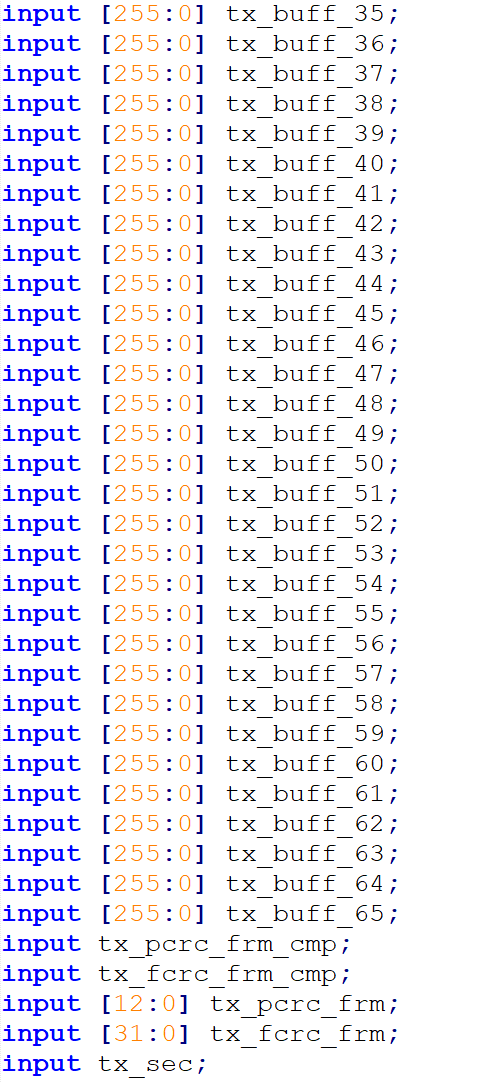
In the above code the “dt\_rm\_frm” is created as soon as the CRC value calculation is done and the “dt\_rm\_frm\_len” which will indicate the length of the total CAN-XL frame being sent and “bit\_stf\_intl” signal initialises the bit stuffing module.



In the above code the signal “datain” and “datain1” are used to send the parts of data that will undergo encryption and ICV calculation for security , the signals “enable” and “enable1” are used to trigger the AES encryption code to start working.

**“Currently the code only supports handling of 213 bits and 333 bits of data ”**





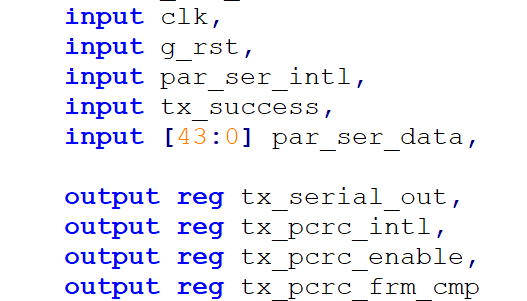
3. Parallel-To-Serial Convertor for PCRC :

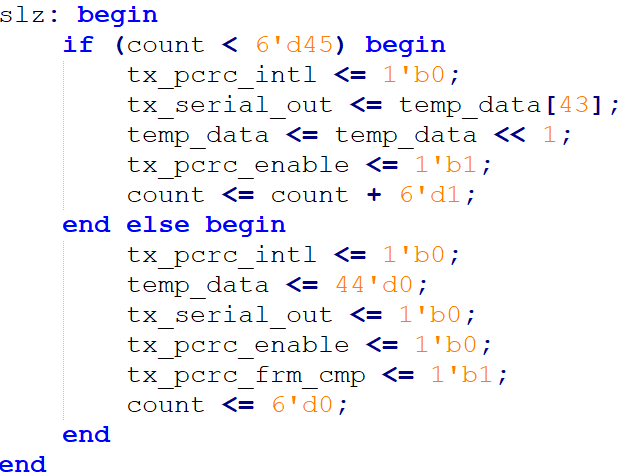
The Parallel-to-Serial Converter in the controller converts the data provided by Data Frame Generator to a serial format meaning 1 bit at a time to make it easier to calculate the PCRC value.



In the above image as seen the “par\_ser\_data” will be converted into a single bit signal called “tx\_serial\_out” and it will be sent to PCRC for calculation and “tx\_pcrc\_intl” to initialise the PCRC module and “tx\_pcrc\_enable” to enable the PCRC module and “tx\_pcrc\_frm\_cmp” to indicate the PCRC module that calculation is done and to stop calculating.

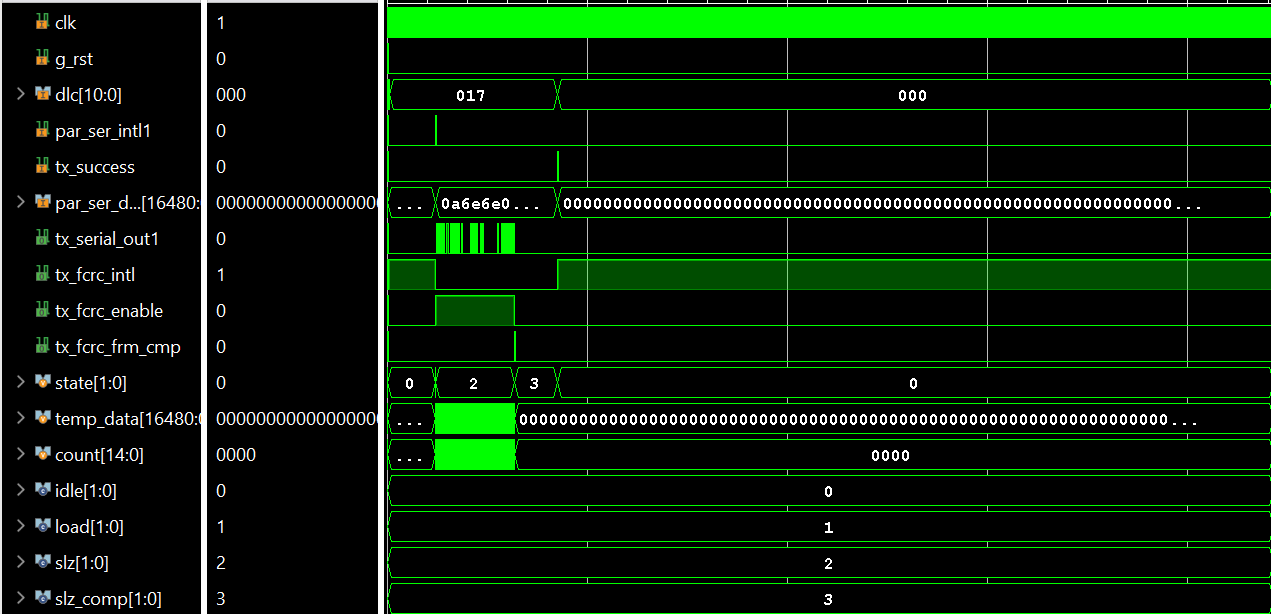
PCRC is calculated from the SOF to last bit of SBC.





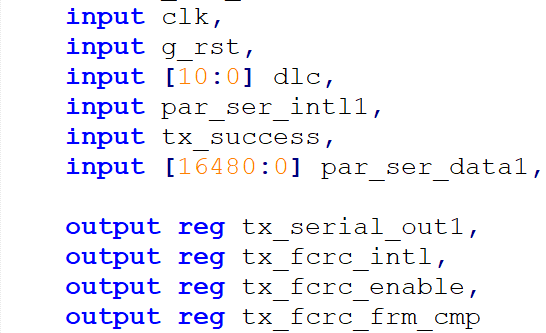
4. Parallel-To-Serial Convertor for FCRC :

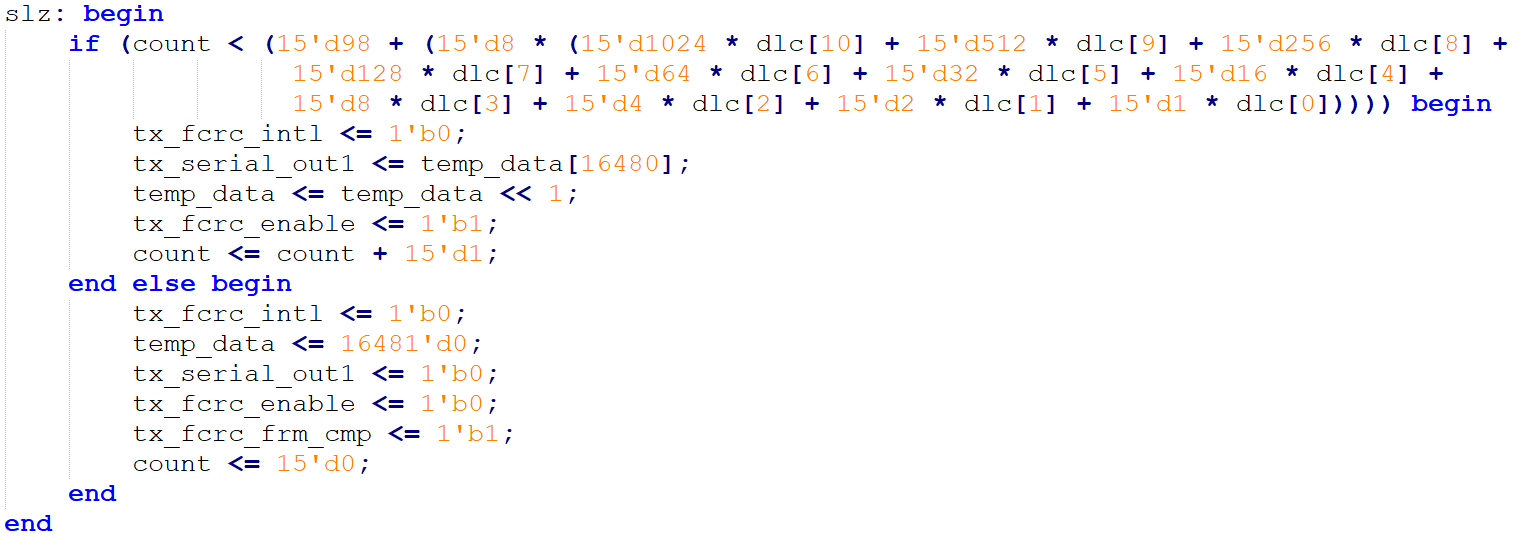
The Parallel-to-Serial Converter in the controller converts the data provided by Data Frame Generator to a serial format meaning 1 bit at a time to make it easier to calculate the FCRC value.



In the above image as seen the “par\_ser\_data1” will be converted into a single bit signal called “tx\_serial\_out” and it will be sent to FCRC for calculation and “tx\_fcrc\_intl” to initialise the FCRC module and “tx\_fcrc\_enable” to enable the FCRC module and “tx\_fcrc\_frm\_cmp” to indicate the FCRC module that calculation is done and to stop calculating.

FCRC is calculated from the SOF to last bit of data payload.

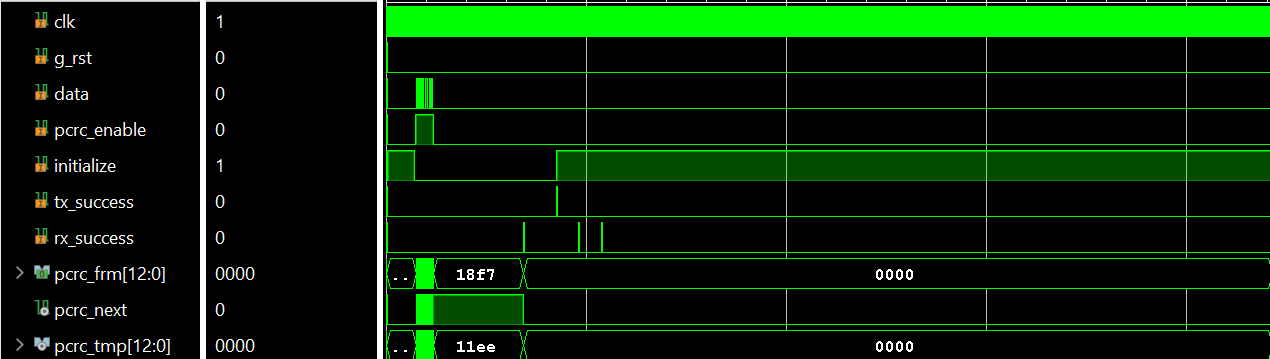


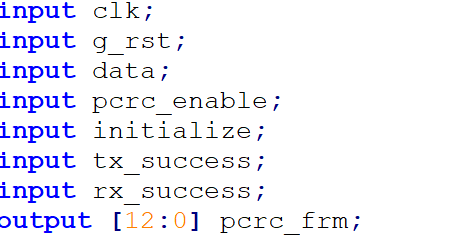


In the above image DLC value is taken as input as FCRC must be calculated till last bit of data payload and it is required to estimate how large is the data payload present.

5. Transmission PCRC :

This module calculates the PCRC value by a polynomial division with polynomial value of hexadecimal 19E7 and the remainder from this will be the calculated PCRC value that must be appended to the CAN-XL frame.



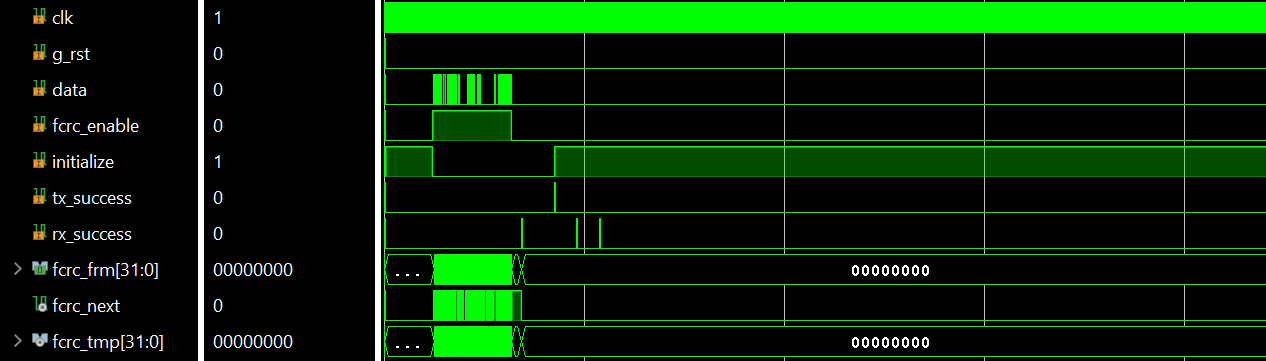


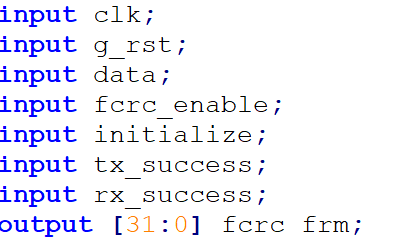


The above image is the polynomial division and “pcrc\_frm” will be the appended PCRC to the CAN-Xl frame.

6. Transmission FCRC :

This module calculates the FCRC value by a polynomial division with polynomial value of hexadecimal FA567D89 and the remainder from this will be the calculated FCRC value that must be appended to the CAN-XL frame.

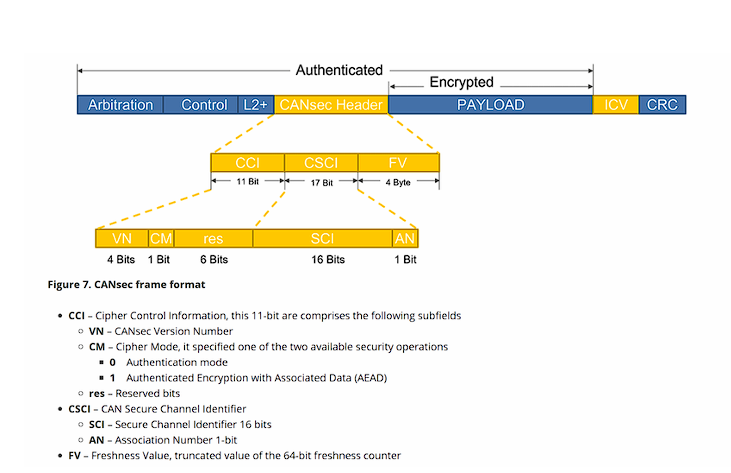


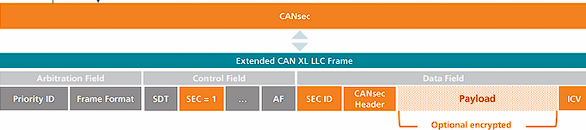




The above image is the polynomial division and “fcrc\_frm” will be the appended FCRC to the CAN-Xl frame.

7. Security by AES-128 at transmitter side :





Security will only be present if SEC value in CAN-XL frame is high. If SEC is low then this process can be ignored.

For CANsec we need at least 28bits (CANsec header)+32bits (FV Freshness Value)+data (at least 128 bits since AES encryption)+128bits (ICV since AES encryption) of data to be present in data field but since the code depends on dlc value and has to match the 8 byte payload gap so then CANsec header is reduced to 24 bits by reducing the number of reserved bits.

Totally data in CAN-XL frame will be 312 bits and full CAN-XL frame will be 461 bits.

Steps for AES 128 encryption :

**1. Key Expansion**

* The 128-bit key is expanded into **11 round keys** (one for each round plus an initial key).
* Each round key is 128 bits.

**2. Initial AddRoundKey**

* The **initial round** begins by XORing the 128-bit plaintext block with the first round key.
* This initial AddRoundKey serves as a "pre-whitening" step before the main rounds.

**3. Main Rounds (Rounds 1 to 9)**

* Each of these rounds consists of four steps:
  + **SubBytes**: Each byte in the state matrix is replaced with a new byte from a lookup table called the S-box. This step introduces non-linearity.
  + **ShiftRows**: The rows of the 4x4 matrix are shifted cyclically to the left by different offsets:
    - Row 0: No shift.
    - Row 1: Shift left by 1 byte.
    - Row 2: Shift left by 2 bytes.
    - Row 3: Shift left by 3 bytes.
  + **MixColumns**: Each column of the matrix is transformed using a linear mixing operation based on Galois field arithmetic. This step diffuses the bytes within each column.
  + **AddRoundKey**: The state is XORed with the current round key, mixing the key into the data.

**4. Final Round (Round 10)**

* The final round is slightly different from the main rounds:
  + **SubBytes** and **ShiftRows** are performed as usual.
  + **MixColumns** is **omitted** in the final round.
  + **AddRoundKey** is applied with the final round key.

**5. Output**

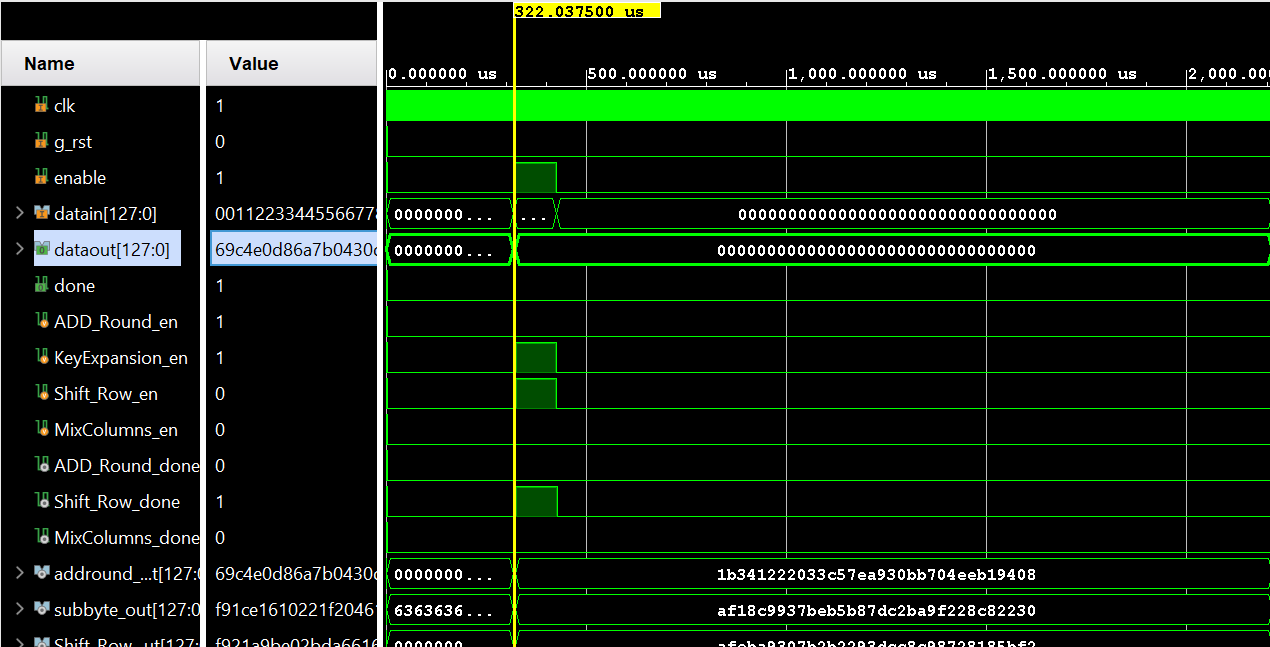
* The result after the final AddRoundKey is the **ciphertext**.

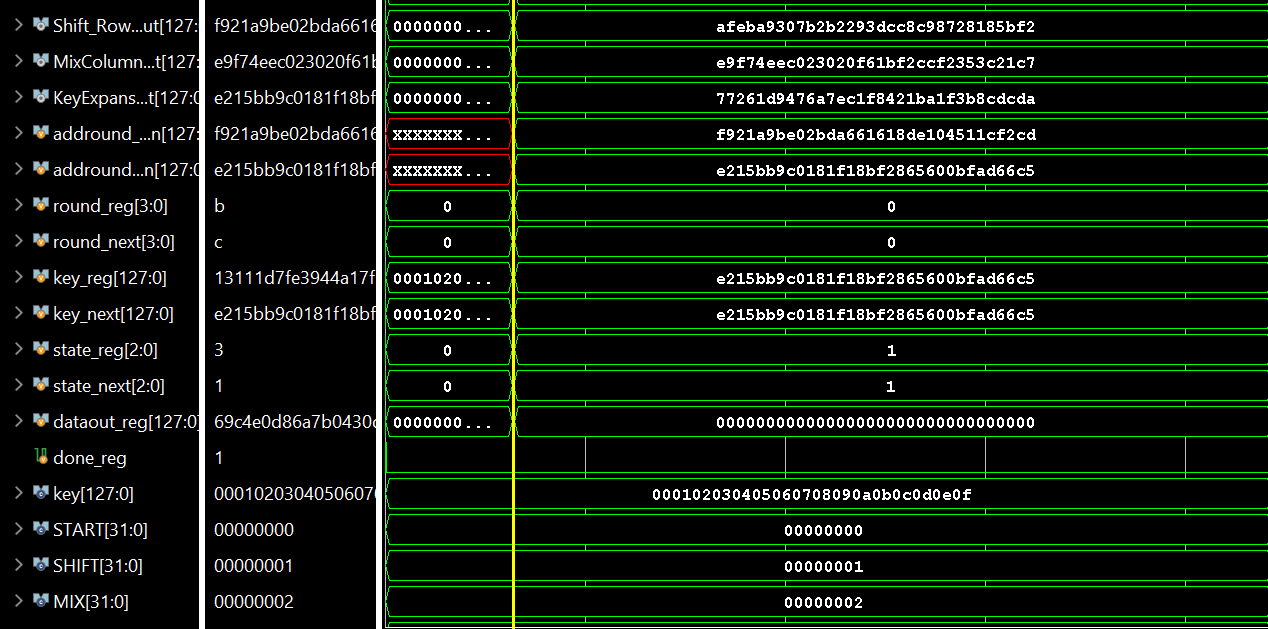
First Encryption is done on payload :

Input Data: 00112233445566778899aabbccddeeff

Key: 000102030405060708090a0b0c0d0e0f

Cipher Result: 69c4e0d86a7b0430d8cdb78070b4c55a



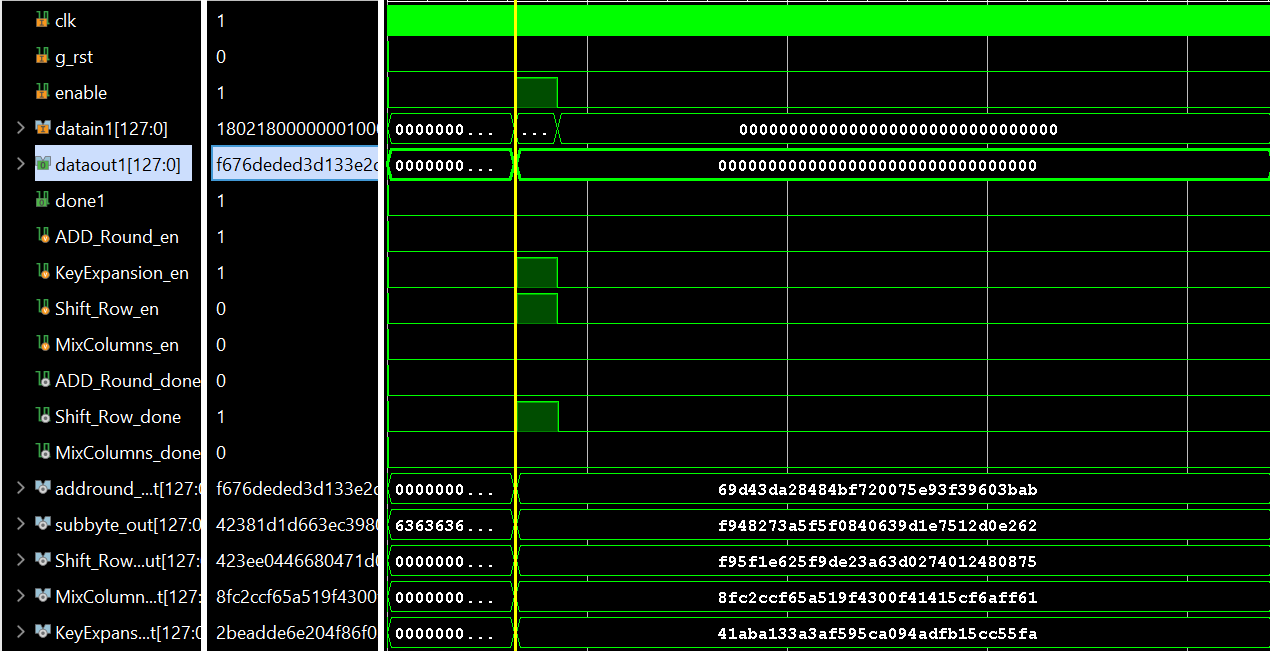


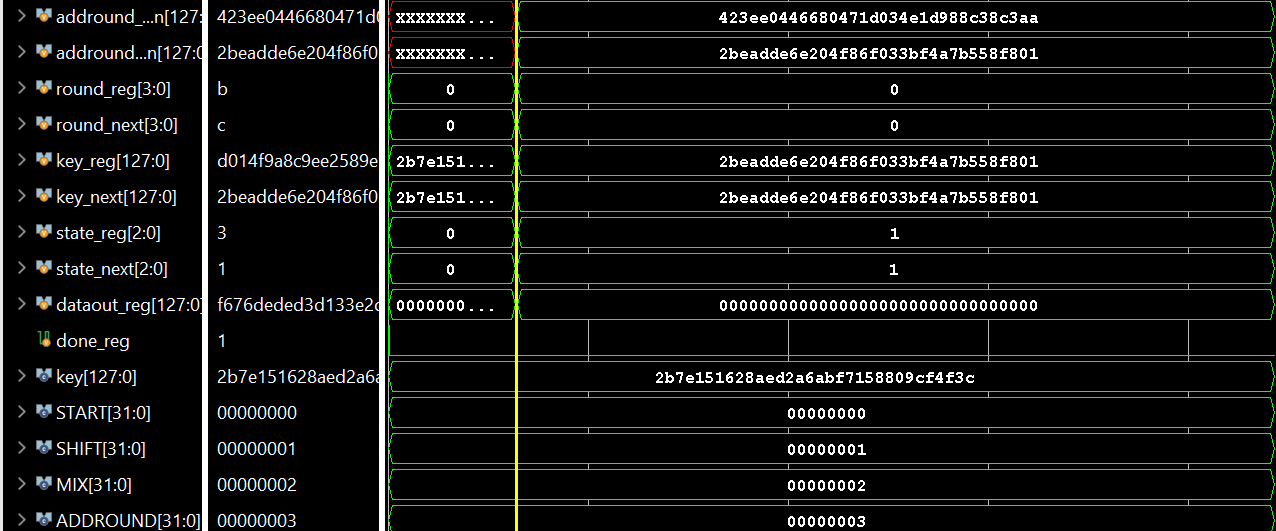
Second Encryption (header + FV) :

Input Data: 18021800000001000000000000000000

Key: 2b7e151628aed2a6abf7158809cf4f3c

Cipher Result: f676deded3d133e2d53aa062f0b61686





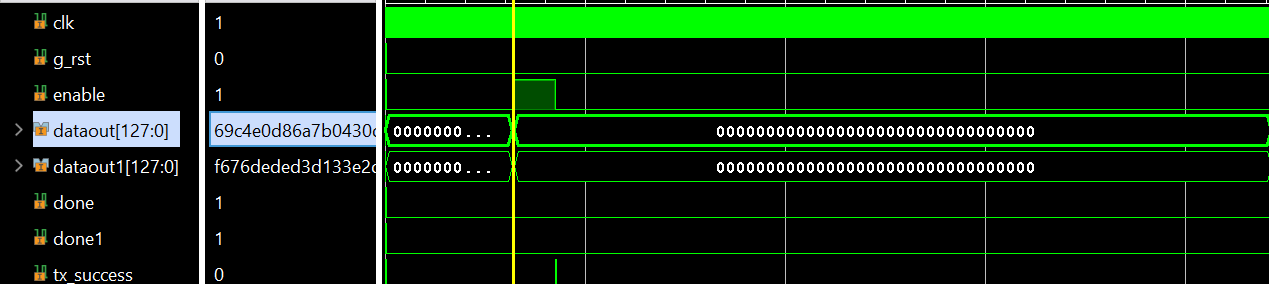
To generate ICV both the encrypted header + FV must be XORed with encrypted payload

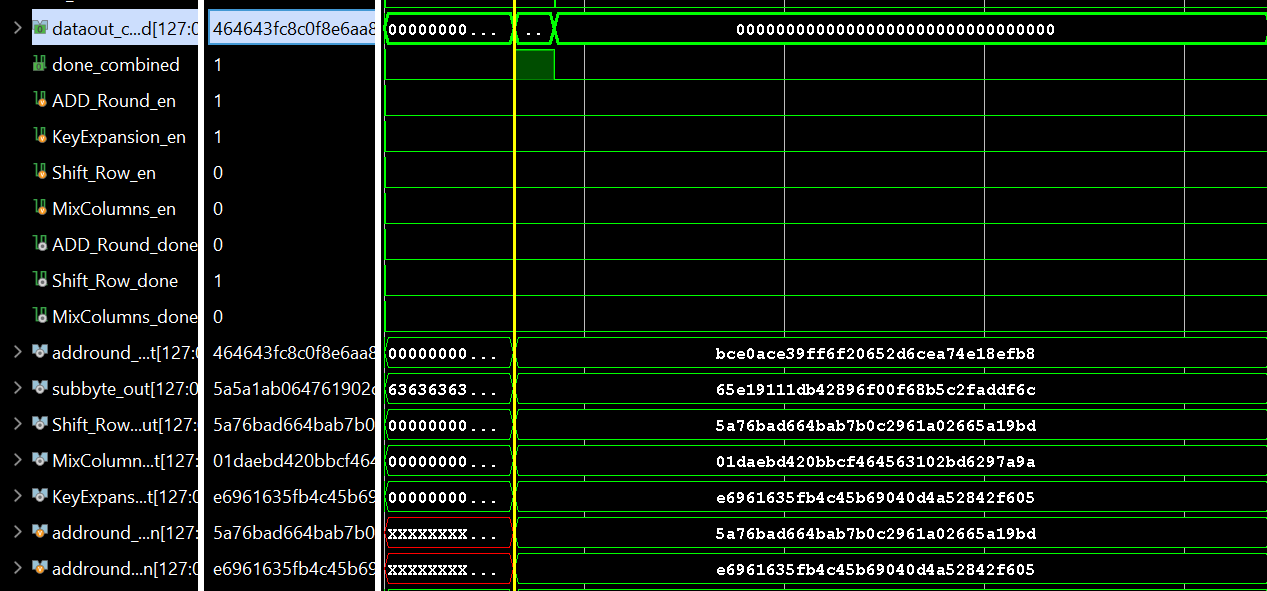
Third Encryption (ICV) :

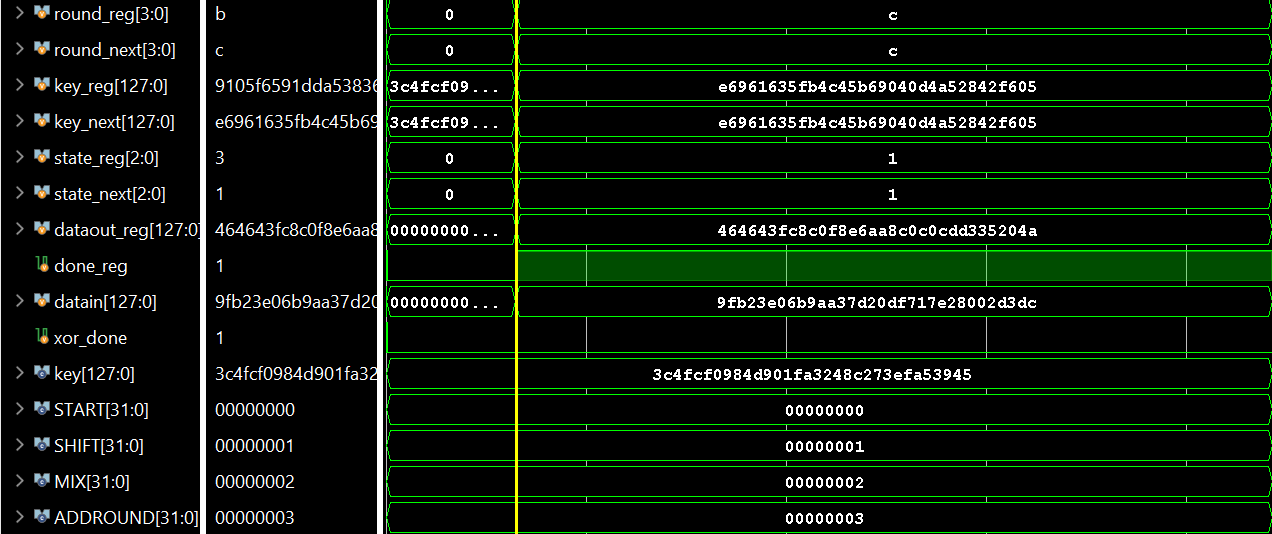
Input Data: 9fb23e06b9aa37d20df717e28002d3dc

Key: 3c4fcf0984d901fa3248c273efa53945

Cipher Result: 464643fc8c0f8e6aa8c0c0cdd335204a

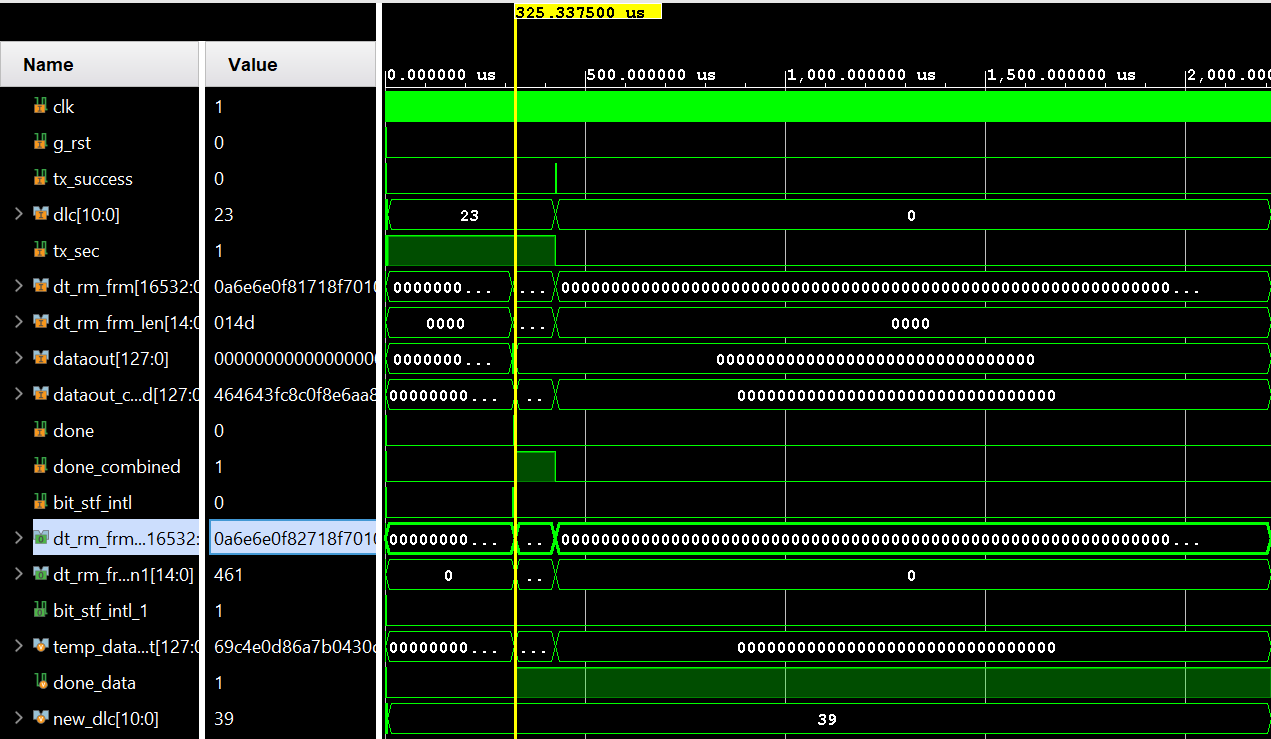


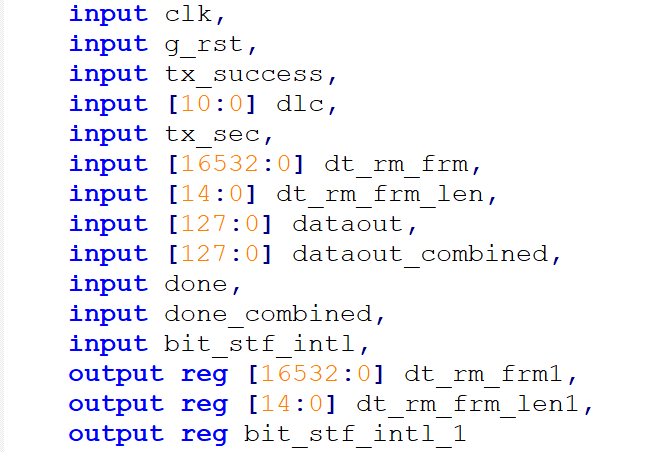


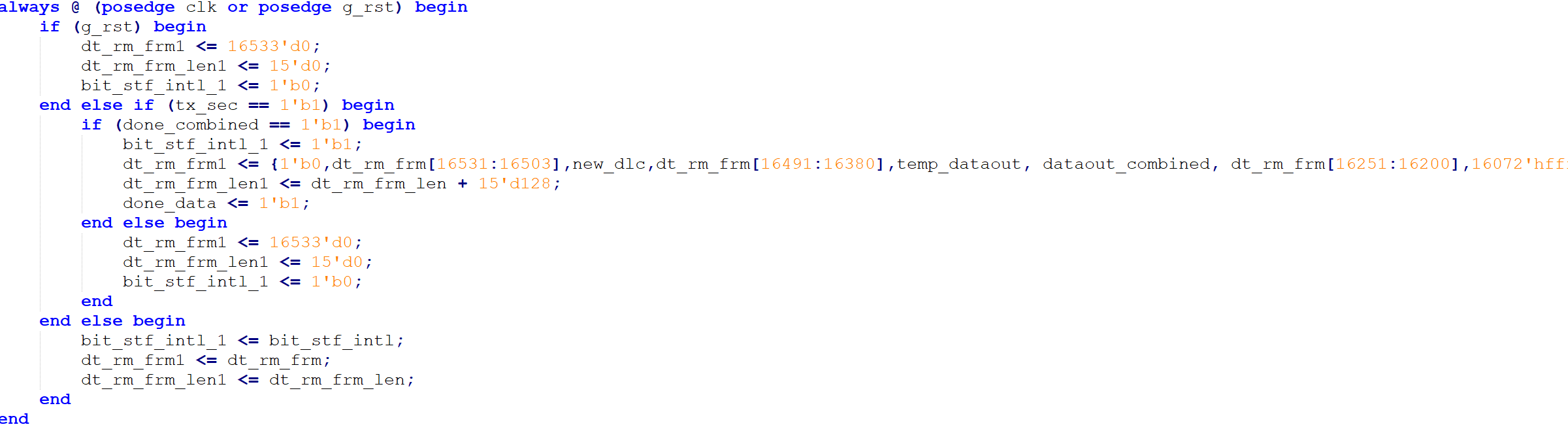


8. New Data Frame Generator after security :

This module replaces the old CAN-XL frame with a new CAN-XL frame which has the encryption on data payload and the ICV value appended to make the final CAN-XL frame with CANsec . This only applies if SEC is high otherwise the data is passed as usual form the data frame generator to the bit stuffing module.



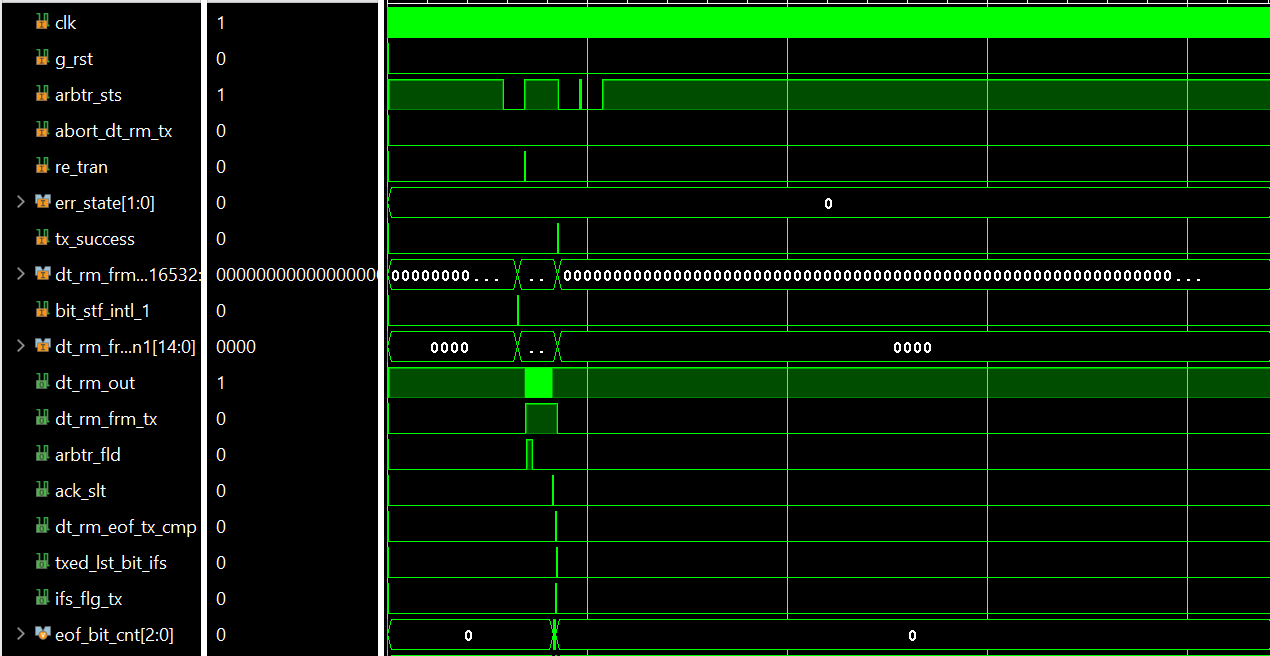


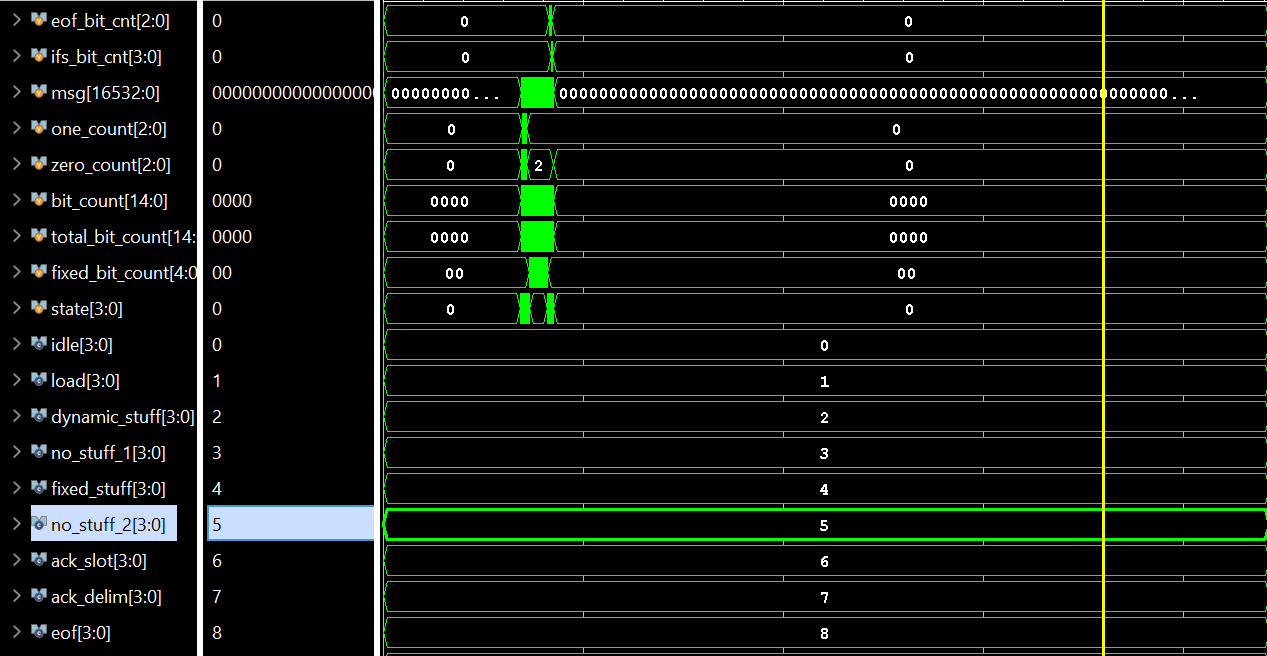


In the above image if “tx\_sec” is high then only we add the extra encrypted part to the previous generated “dt\_rm\_frm” else we use the same data and it is passed into a new signal called “dt\_rm\_frm1” and also a new “dt\_rm\_frm\_len1” is need as 128 bit of ICV is added to the data payload.

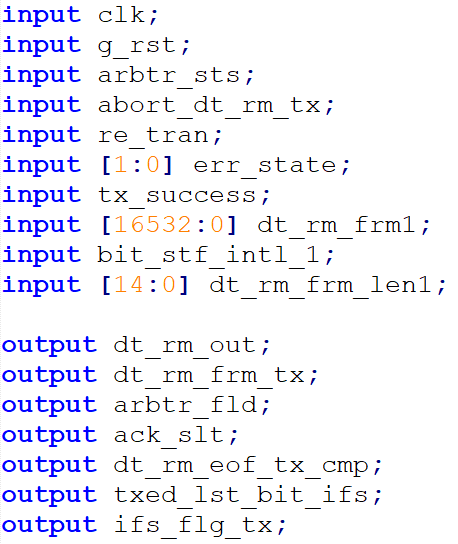
9. Bit Stuffing :

In this module due to NRZ transmission of CAN-XL we need to add stuff bits to ensure data integrity and synchronization. Before the FDF bit, a dynamic stuffing rule is applied. The transmitter inserts, after each sequence of five consecutive equal bits, one bit of inverse value, called a dynamic stuff bit. Starting at DL1 bit up to the last bit of FCRC, a fixed stuffing rule is applied. The transmitter inserts, after 15 consecutive bits a fixed stuff bit. The fixed stuff bit has the inverse value of its preceding bit.





In the above picture “dt\_rm\_out” will be the final output signal with the stuff bit which will send the data to serialized frame transmitter to be sent out to can bus out.



10. Serialized Frame Transmitter :

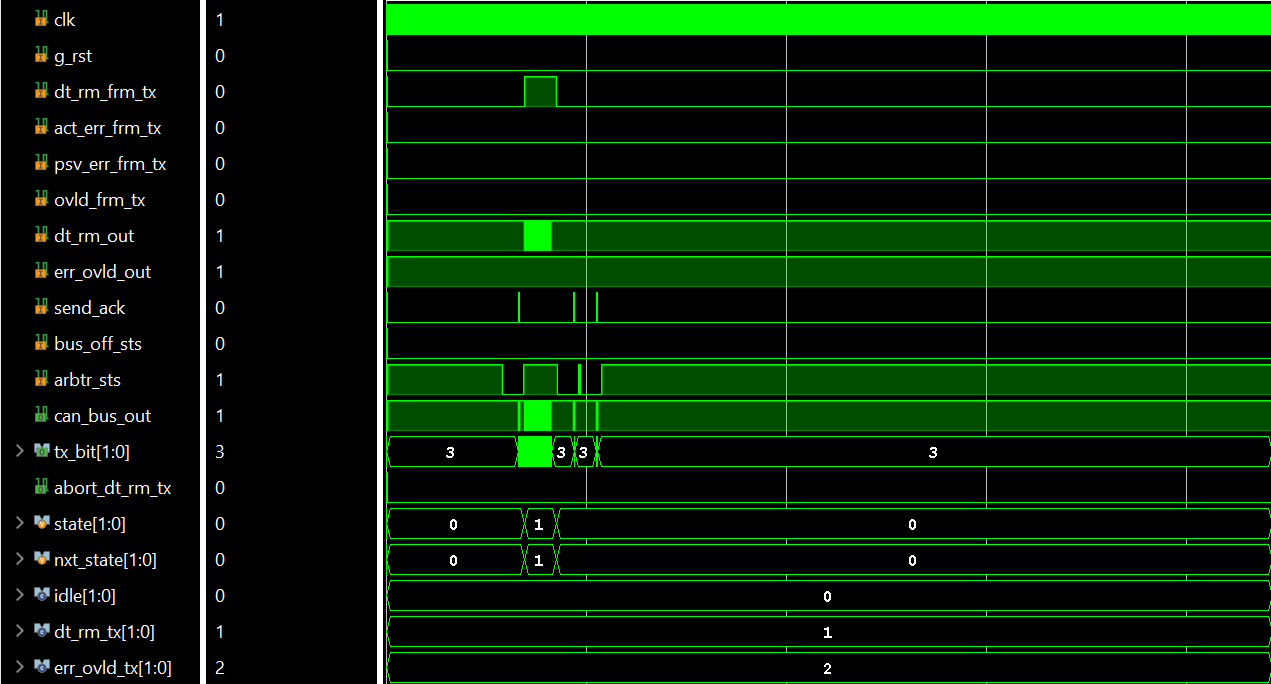
The Serialized Frame Transmitter manages various transmission scenarios in a structured manner, adhering to the CAN XL protocol for data, remote, error, and overload frames.

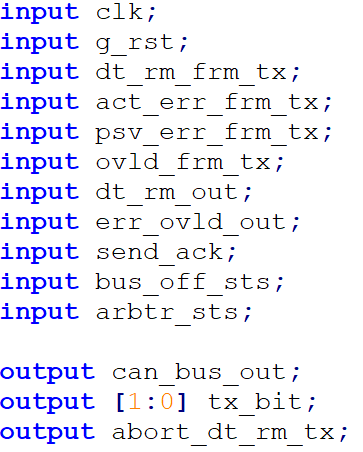
If “act\_err\_frm\_tx” or “psv\_err\_frm\_tx “(error frames) is active and “bus\_off\_sts” is inactive, or if “ovld\_frm\_tx “(overload frame) is active moves to err\_ovld\_tx state and Primarily transmits error or overload frames as needed.

If” dt\_rm\_frm\_tx” (data/remote frame transmission) and “arbtr\_sts” (arbitration success) are active, and “bus\_off\_sts” is inactive move to dt\_rm\_tx and sets “can\_bus\_out” to “dt\_rm\_out”.

If neither condition is met and acknowledgment (send\_ack) is needed it will set “can\_bus\_out” to zero for acknowledgment. **“Since there are 4 nodes we will get 3 send\_ack and all other nodes serialized frame transmitter will also need to have 3 send\_ack for code to work correctly”**.

The “tx\_bit” register stores the last two bits of “can\_bus\_out” output, with each bit delayed by one clock cycle.





11. Message processor :

The msg\_processor is responsible for processing message reception and transmission signals. This module manages error handling, transmission success, acknowledgment, retransmission, and overload conditions.

Error Detection and Handling State:

When errors like “stf\_err”, “frm\_err”, “pcrc\_err”, “fcrc\_err”,”bit\_err” are detected, the module sets “stf\_frm\_crc\_err\_pre” or “bt\_ack\_err\_pre” as needed.

It clears these error indicators upon successful frame transmission (“dt\_rm\_eof\_tx\_cmp” or txed\_lst\_bit\_ifs”).If errors occur, “re\_tran” initiates retransmission based on “msg\_due\_tx\_reg” and other conditions.

Message Reception State:

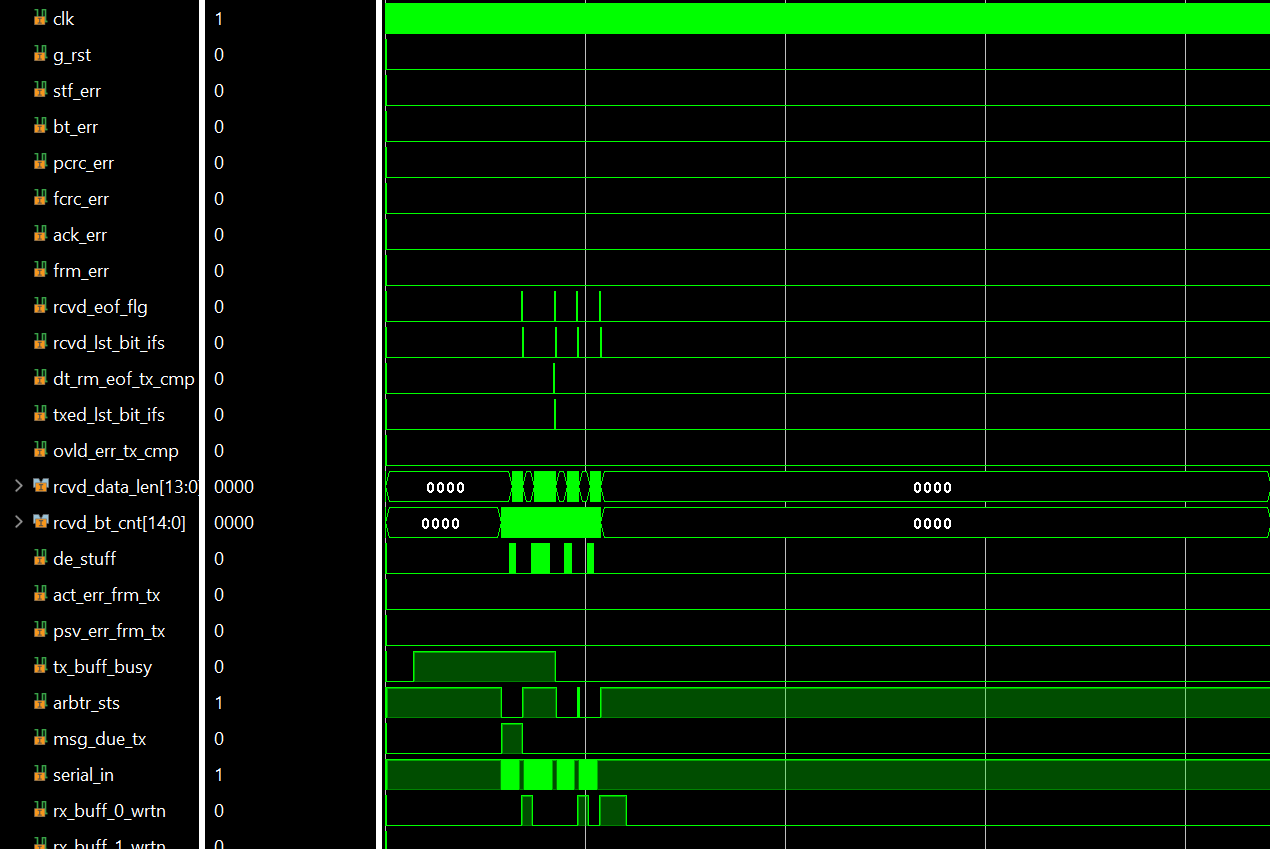
During reception,” rx\_eof\_success” and “rx\_success” signals are generated if the EOF is received and no errors occur. Acknowledge (“send\_ack”) signals are generated based on the conditions and bit count. “There has to be 3 “rx\_success” in all nodes only then the code is working correctly”

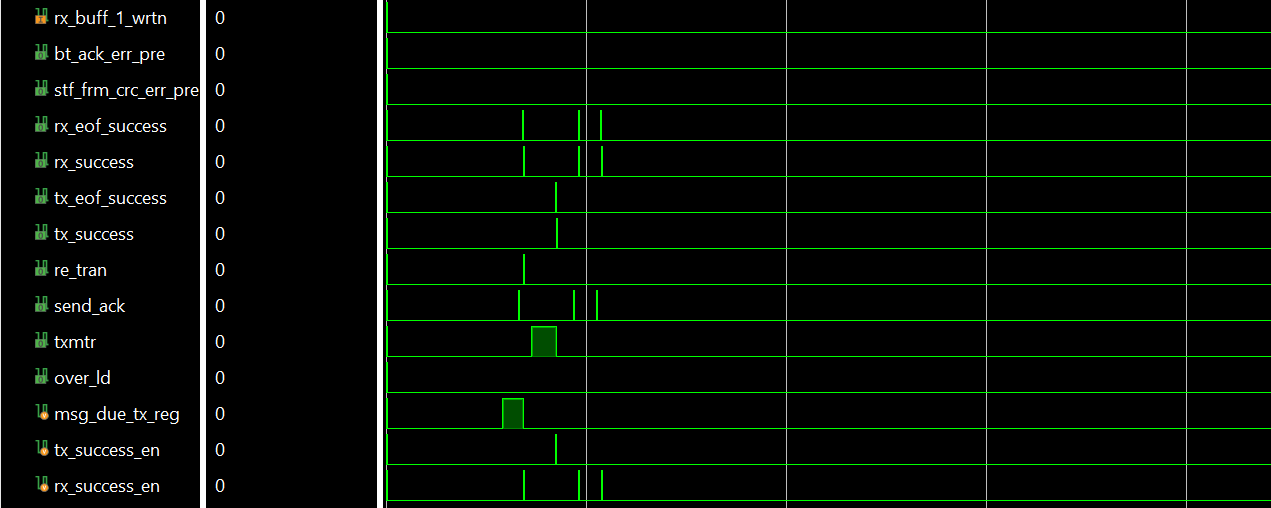
Message Transmission State:

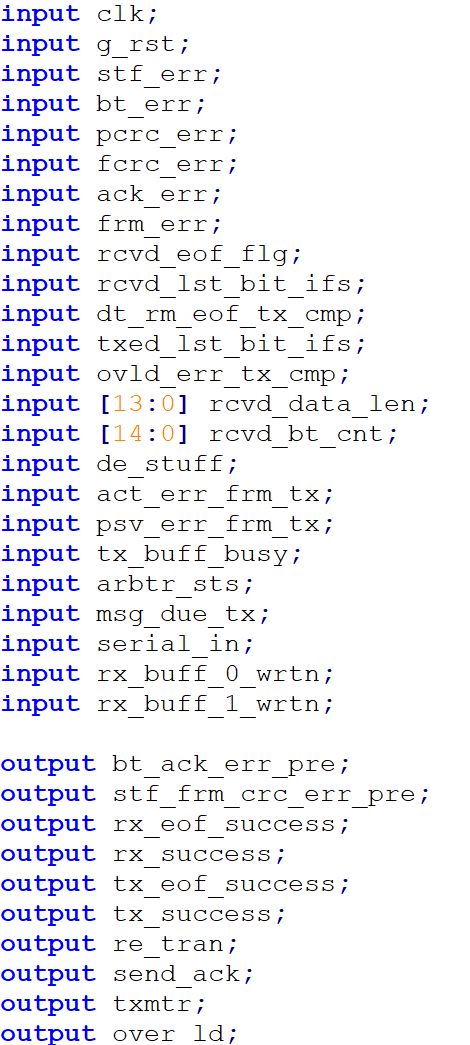
The module enables “tx\_eof\_success” and “tx\_success” when transmission is complete without errors, and acknowledgment is received from other nodes.

Overload and Retransmission State:

In case of overload or errors, the module generates the “over\_ld” signal to delay further transmissions and initiate a retransmission if needed (“re\_tran”). The module reverts to idle or reset logic after the overload is resolved or an error frame is sent.







12. Arbitration controller :

The arbtr\_ctrl module is an Arbitration Controller in the controller, responsible for managing arbitration between nodes and determining when a message is due for transmission.

In idle state ,On reset (g\_rst), the module sets arbtr\_sts to 1 (indicating it’s ready for arbitration) and msg\_due\_tx to 0 (indicating no pending message).This initializes the arbitration process, allowing the controller to participate in arbitration.

Arbitration Start State:

When arbitration conditions are met, such as when a message is due for transmission and no errors are present, arbtr\_sts is set high to indicate active arbitration.If the controller loses arbitration (e.g., another node’s priority is higher), arbtr\_sts is set to 0, and msg\_due\_tx is set to 1 to indicate that the message must be retransmitted when arbitration is reattempted.

Transmission Pending State:

When tx\_buff\_busy is active with certain error conditions (bt\_ack\_err\_pre, bit\_destf\_intl), the node keeps arbtr\_sts low, preventing it from accessing the bus.In this state, msg\_due\_tx is high, signaling that a message is still waiting to be sent but cannot currently contend for arbitration due to bus conditions.

Arbitration Loss/Retry State:

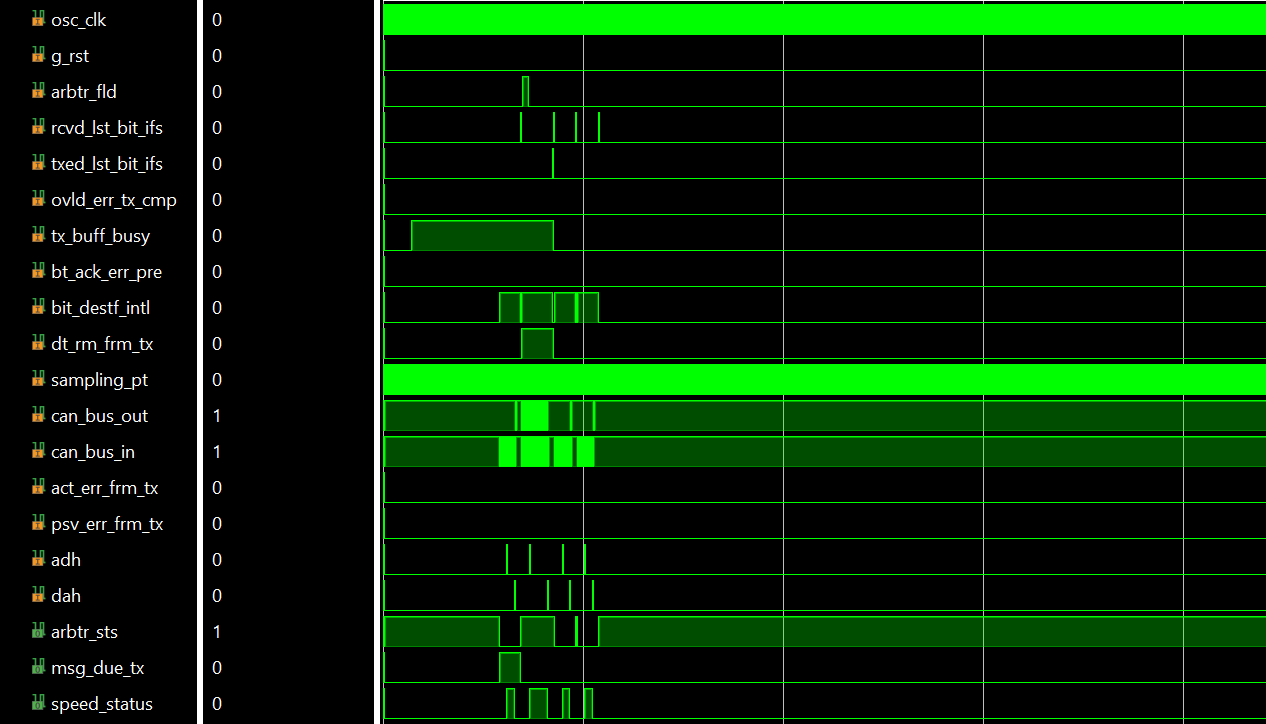
If arbitration fails (when transmitted and received bits don’t match during sampling\_pt), arbtr\_sts is set to 0, and msg\_due\_tx is set to 1. This allows the node to retry transmission in subsequent cycles when the bus is free.

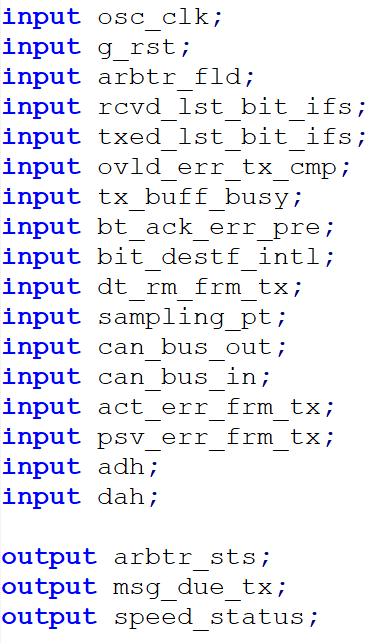
Speed Status State:

Independently of arbitration, the speed\_status flag is updated based on adh and dah signals, which affect the transmission speed.

**“Based on the priority id which ever node has the highest priority meaning which ever puts the 0th bit first will win arbitration and other nodes have to wait their turn to acquire arbitration which can cause some nodes to re transmit data which is completely correct”**

**“ Due to buffers always being on the priority id must be specific for the given combination of nodes to work correctly and this will depend on platform also for vivado given priority id in testbench work if any changes done to it then even in vivado it wont work . The same applies for cadence system ”.**

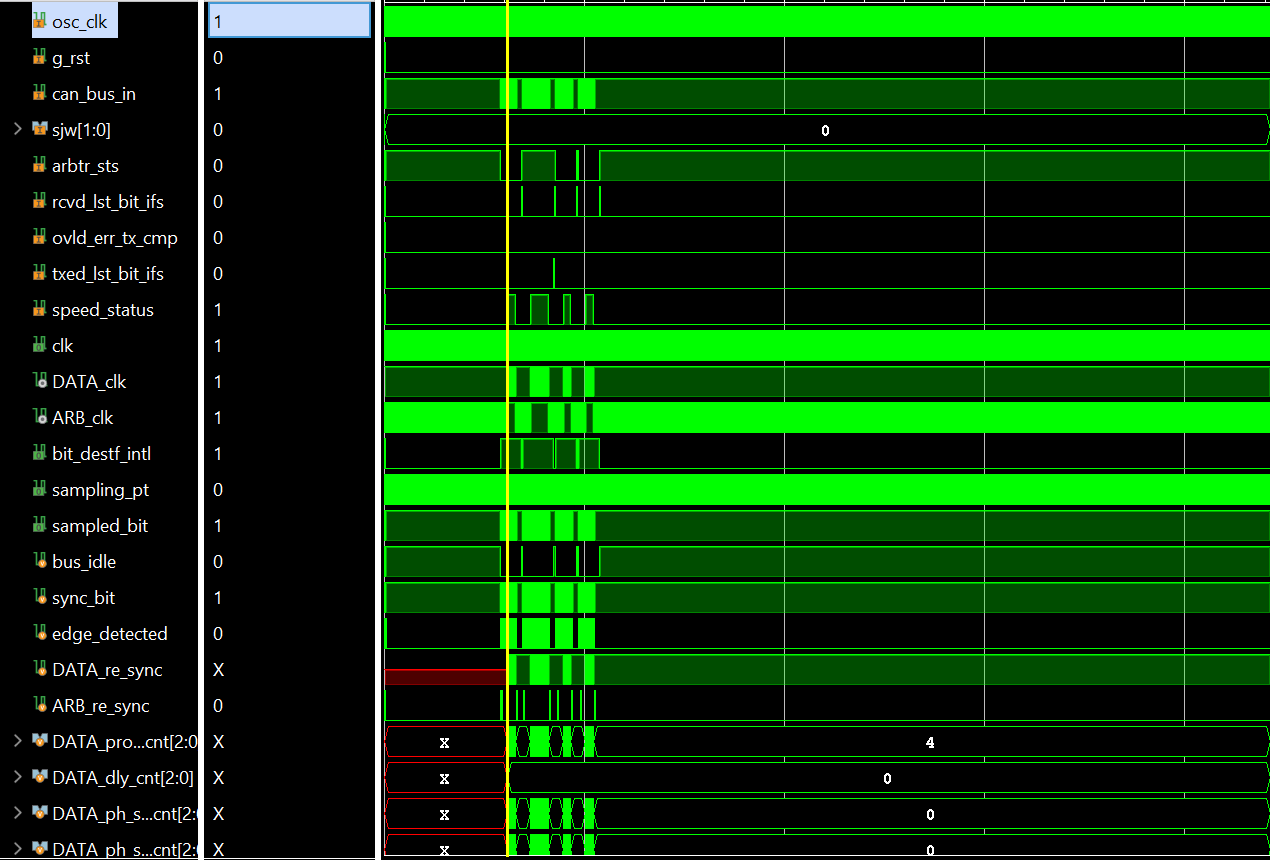


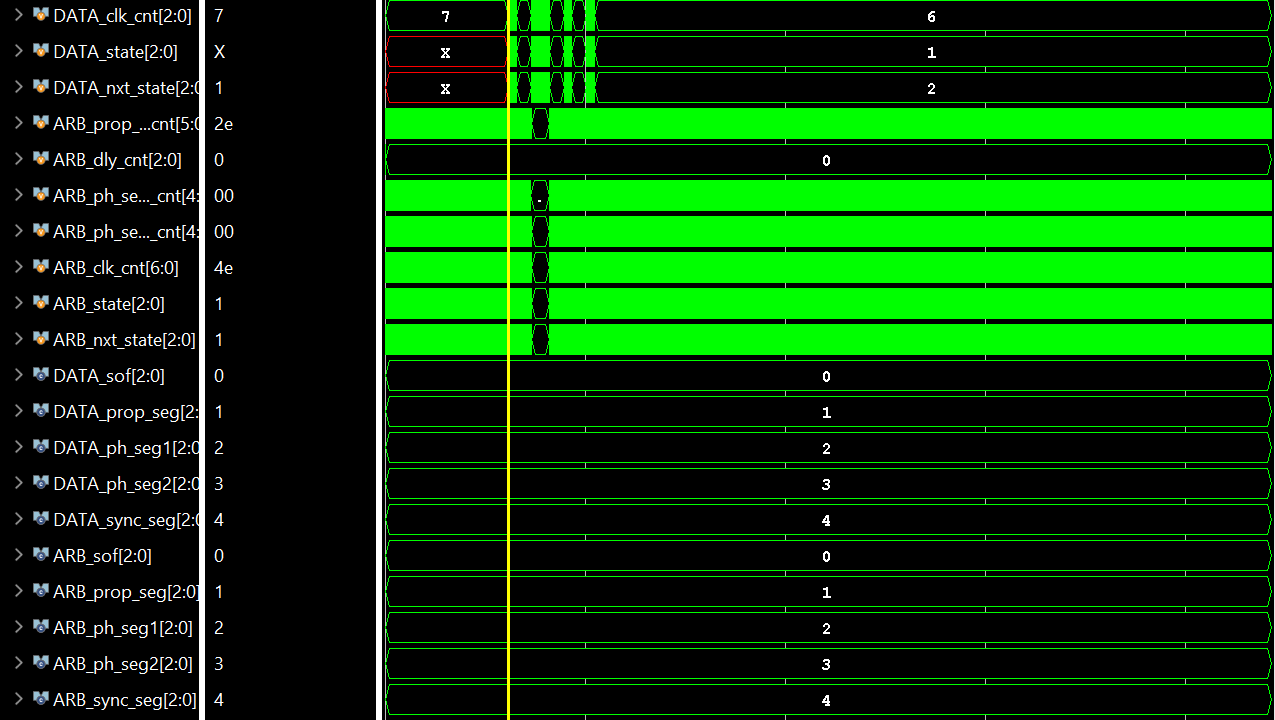


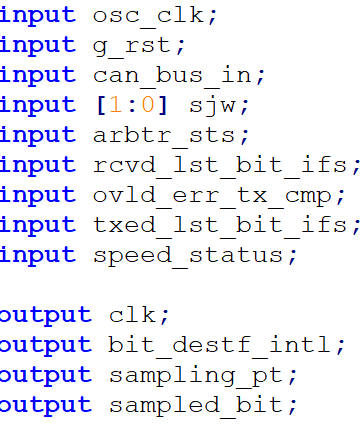
13. Synchronizer :

The Synchronizer in the controller ensures proper timing alignment between the controller and the bit stream on the bus. It achieves this by detecting recessive-to-dominant transitions in the received data and using these edges to synchronize or re-synchronize the controller's internal clock with the network. This synchronization is crucial for maintaining accurate timing, especially in a distributed network where nodes operate independently. The Synchronizer helps prevent errors caused by clock drift, ensuring reliable data reception and seamless communication across the network.

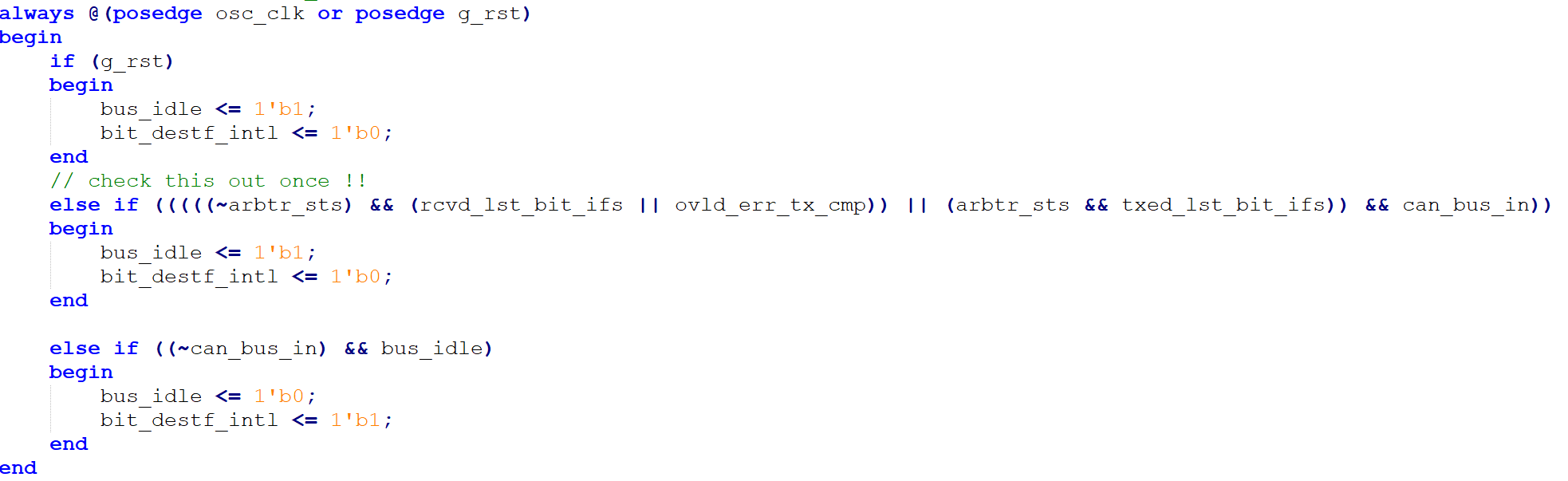
It also changes the clock speed upon receiving signal “speed\_status” which happens only when the bit crosses into the data phase.







**“This code is not changed it remains the same as provided but will need to be remade as it does not pass in synthesis command as it gives errors”**

****

**“In the above code the signal “bit\_dest\_intl” initialises the bit destuffing module to work if it does not happen at the correct bit then bit destuff will miss its values . For this reason only the code works perfectly only if SOF value was 0 as always as “~can\_bus\_in” line means it is searching for the first 0 bit as “can\_bus\_in” is initialised to high state from the beginning.”**

**“ In testbench it is provided that can\_bus\_in = (can\_bus\_out\_0 & can\_bus\_out\_1 & can\_bus\_out\_2 & can\_bus\_out\_3) , thus indicating that can\_bus\_in will have all the data form all 4 nodes present”.**

14. Bit De-Stuffing :

This module removes the stuff bit which bit stuffing module has put in the data and this module only starts when “bit\_destf\_intl” comes and the data comes form synchroniser as “sampled\_bit”

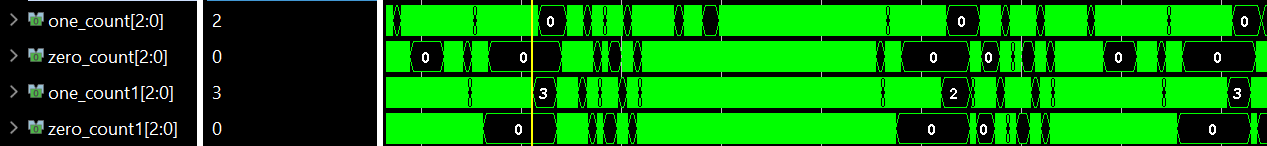
This module also separates each data into a separate register as it matches the CAN-XL frame and also provides enable signals for security checker on receiver side.



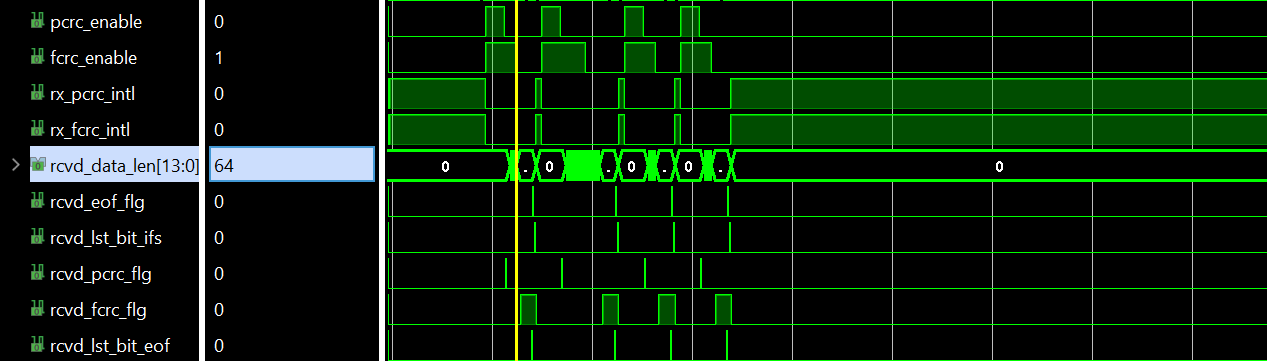
**In the above image we can see “bit\_destf\_intl” signal coming up 4 times as the destuffing module will handle all the data from all 4 nodes at the same time.**



In the above image rcvd\_bit\_cnt counts the total number of bits of CAN-XL frame that comes in.

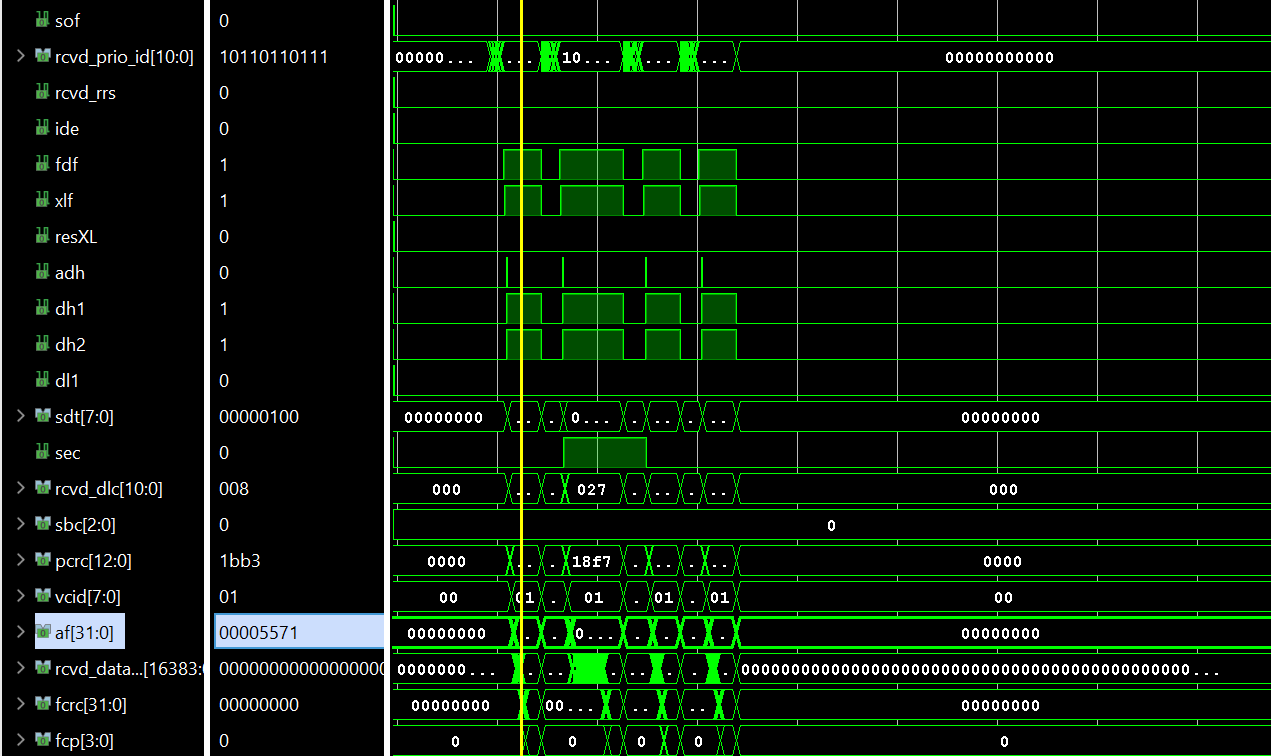


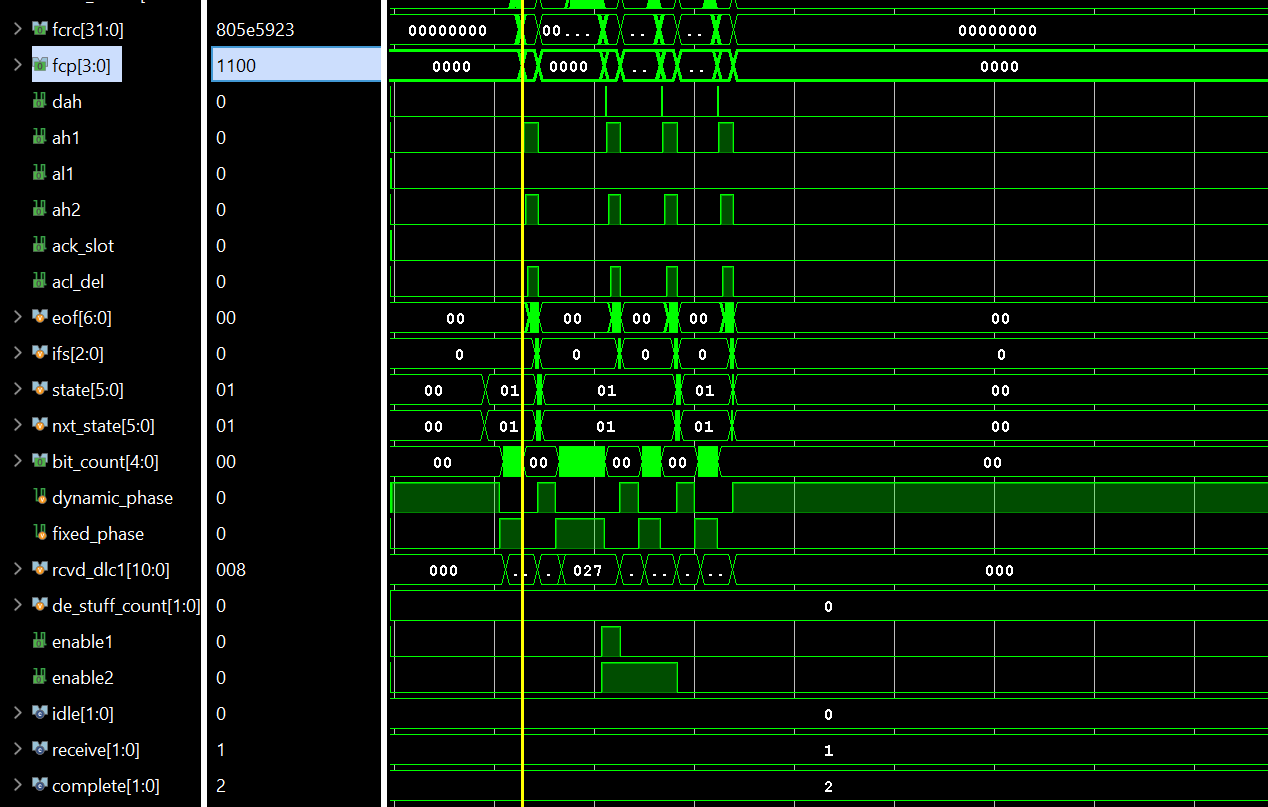
In the above image “one\_count” and “zero\_count” and “one\_count1” and “zero\_count1” are used to check for the stuff bit.



In the above image “pcrc\_enable” and “rx\_pcrc\_intl” are used to initialise and enable the receiver PCRC module to calculate PCRC. “fcrc\_enable” and “rx\_fcrc\_intl” are used to initialise and enable the receiver FCRC module to calculate FCRC.

Singals like “rcvd\_data\_len” is used to tell the number of bits present in data payload , “rcvd\_eof\_flg” is used to indicate when first bit of EOF (End Of File) has come ,“rcvd\_lst\_bit\_eof” is used to indicate when last bit of EOF (End Of File) has come, “rcvd\_lst\_bit\_ifs” is used to indicate when last bit of IFS (Inter Frame Spacing) has come, “rcvd\_pcrc\_flg” is used to indicate when last bit of PCRC has come, “rcvd\_fcrc\_flg” is used to indicate when last bit of FCRC has come.

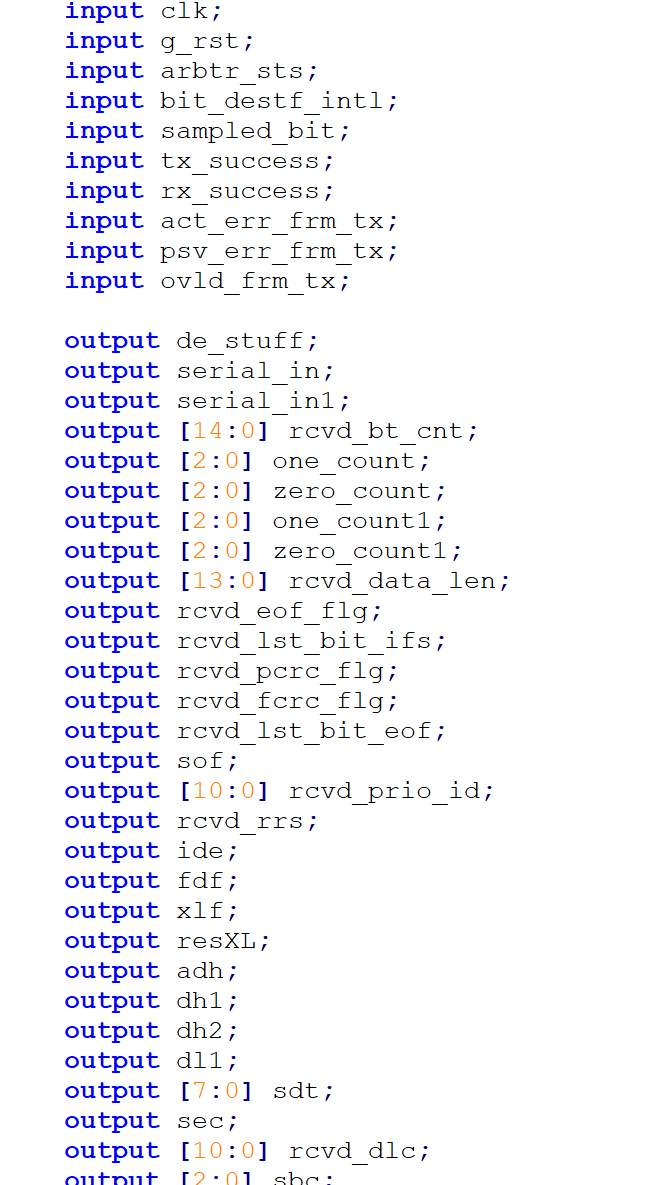


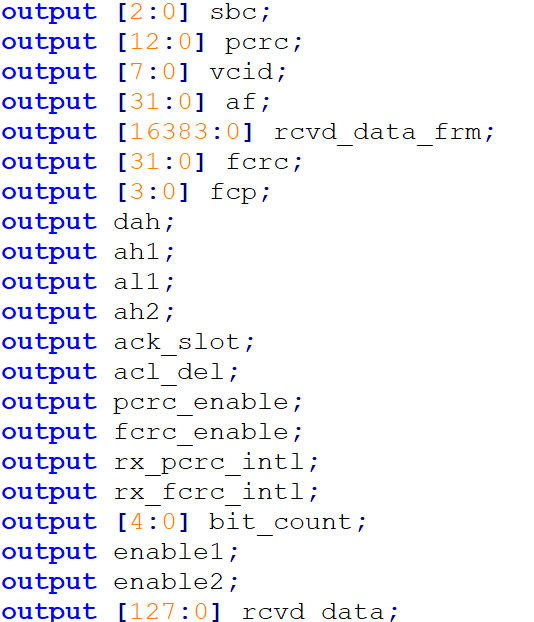


In the above image each register is matched accordingly to each segments of CAN-XL frame as in order and the value is the same as provided in testbench.

**““rcvd\_data\_frm” signal contains the data since it is 16384 bit long it must be reversed indexed and can be copied in vivado to indicate if correct data has come correctly but it is not possible to do this on cadence system hence using vivado is much better for verification and it is very important for security if no data comes into De-Stuffing module then security is not possible” .**

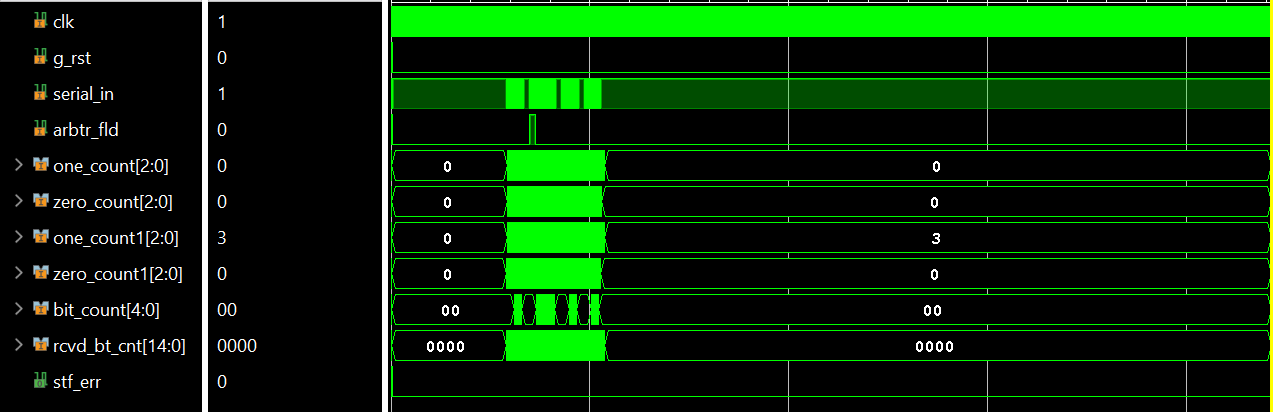
**“Example of data with security by copying data in reverse bit order : 0101001000000100101011001100101110110011000000110000001100010101010101100111000111110000001100010011111111000010011000100110001001011010101000110010110100001110000000011110110110110011000110110000110000100000110111100101011000011011000001110010001110010110100000000000000000000000000000000001100001000000000110000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000...”**





15. Bit Stuff Monitor :

This module checks if the stuff bit is correctly placed and not put in wrong accordance against the rules of stuffing for CAN-XL , if any problem found then it results in “stuff\_err”.

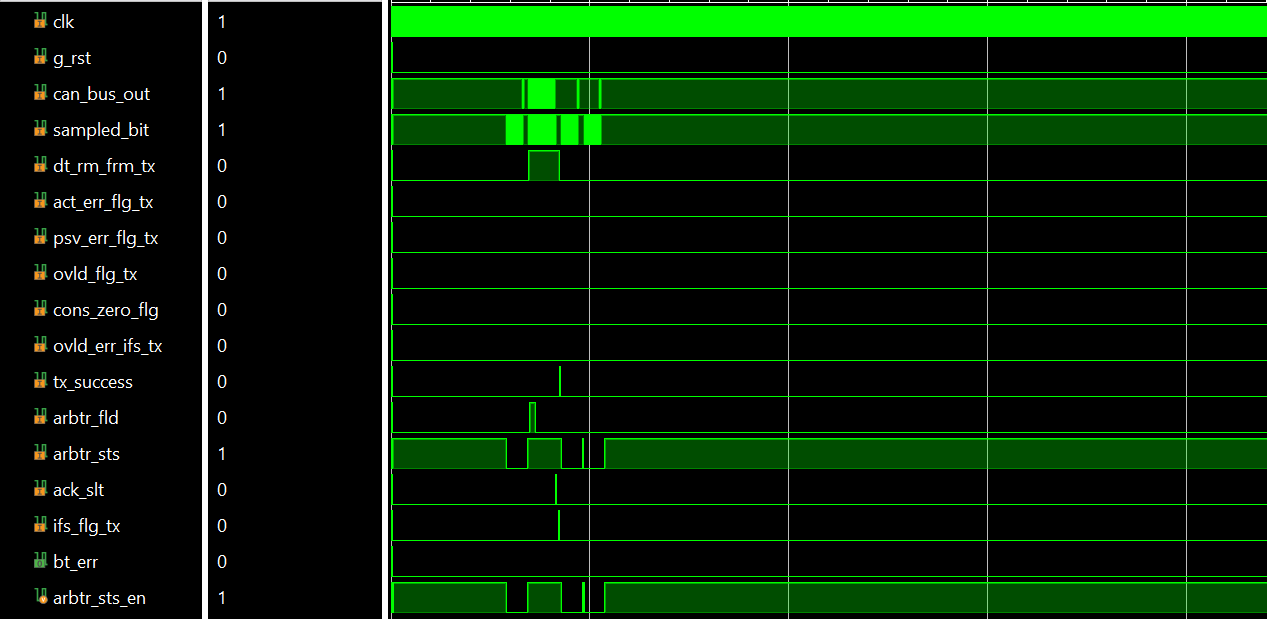
****

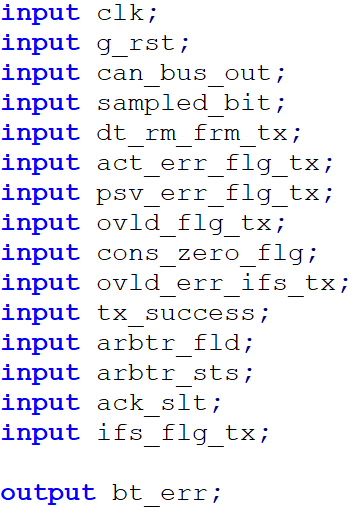
ln the above image “stuff\_err” must be always be low even after all data has comes to indicate no errors are present.

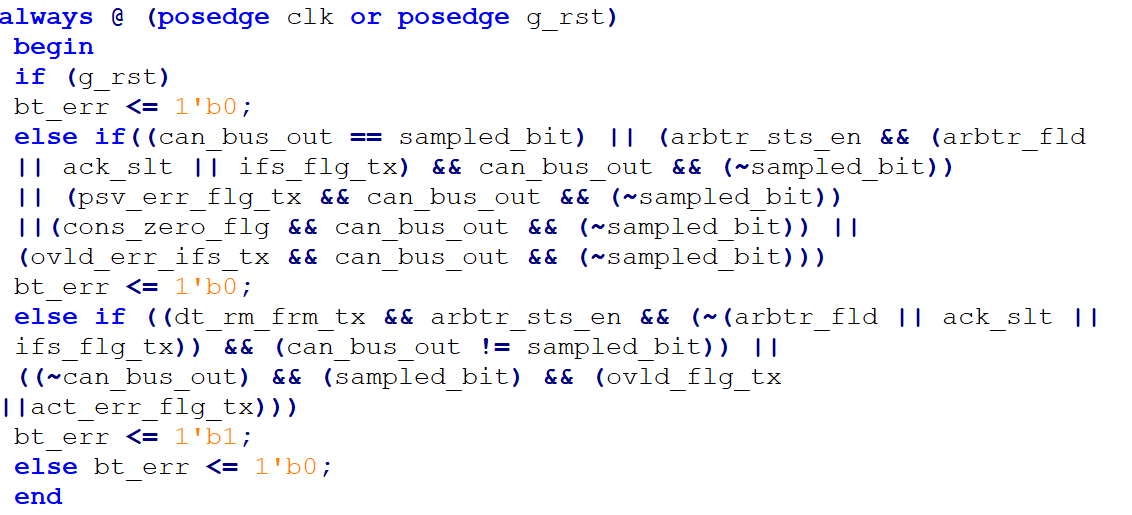
****

16. Bit Monitor :

The bit\_monitor module is responsible for monitoring bit-level errors in the CAN bus during transmission and reception. It checks for conditions such as arbitration mismatches, acknowledgment errors, and overload situations.







In the above image it must undergo these conditions to check if error is present.

Mismatch During Transmission:

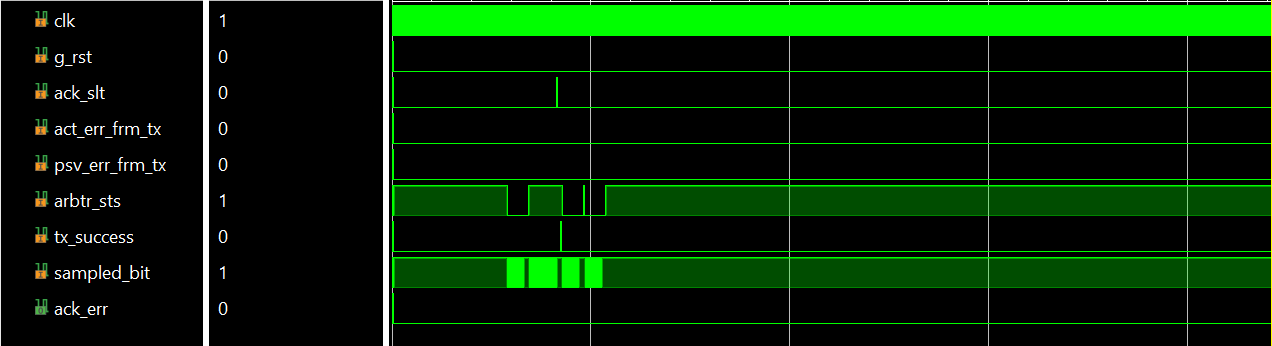
When data transmission is ongoing (dt\_rm\_frm\_tx is high),Arbitration status is enabled (arbtr\_sts\_en is high),None of the arbitration-related fields (arbtr\_fld), acknowledgment slot (ack\_slt), or inter-frame space (ifs\_flg\_tx) are active,The transmitted bit (can\_bus\_out) does not match the received (sampled) bit (sampled\_bit).This indicates a bit error during normal data transmission.

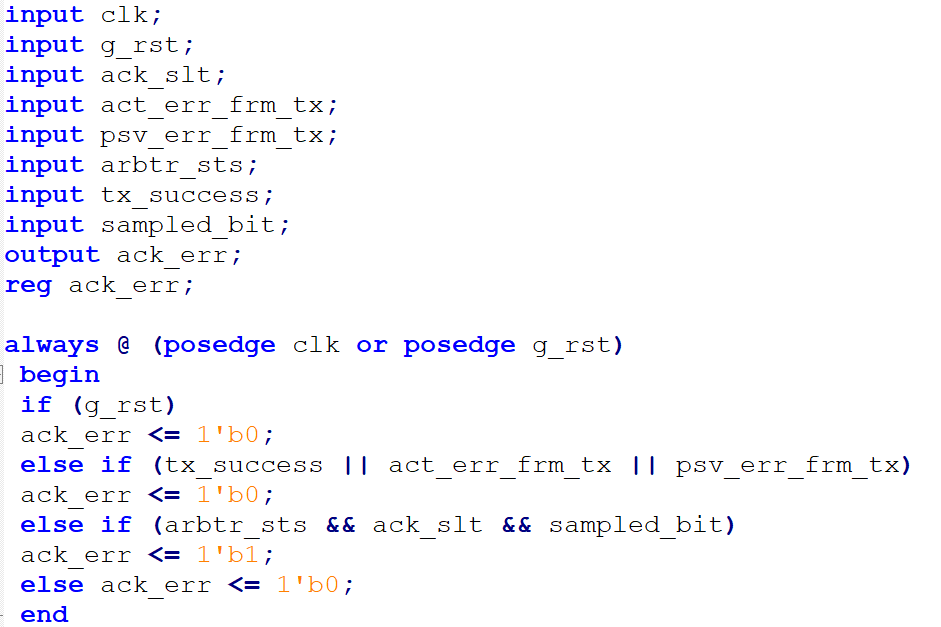
Overload or Active Error States:

When the transmitted bit (can\_bus\_out) is recessive (0) and the received bit (sampled\_bit) is dominant (1),Either the overload flag (ovld\_flg\_tx) or active error flag (act\_err\_flg\_tx) is set.This condition suggests an error caused by overload or an active error state.

17. Acceptance Checker :

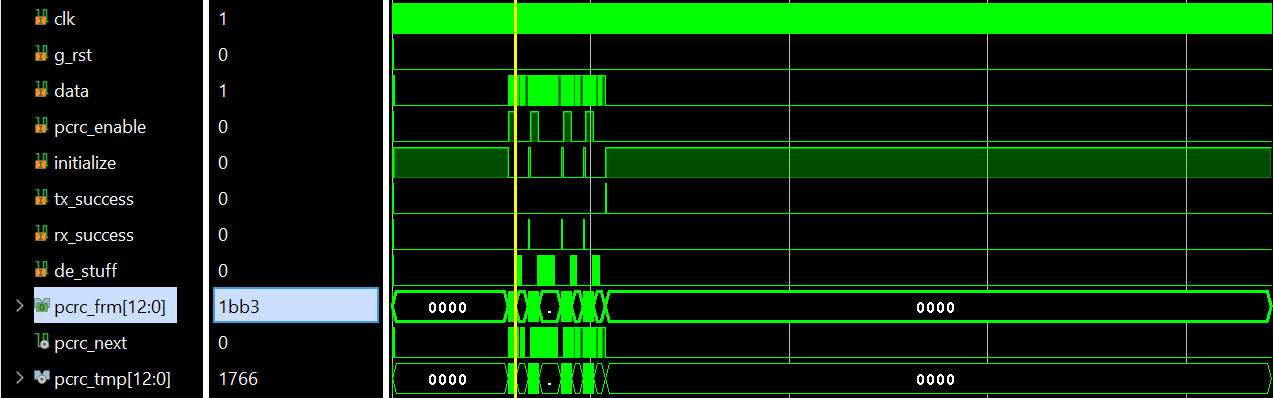
The Acknowledgment Checker in the controller monitors the acknowledgment process during message transmission. When the controller transmits a frame, it sends a recessive bit in the acknowledgment slot and expects to receive a dominant bit from at least one node on the CAN bus, indicating successful reception of the message. If a dominant bit is not detected during the acknowledgment slot, the Acknowledgment Checker flags an ACK error.

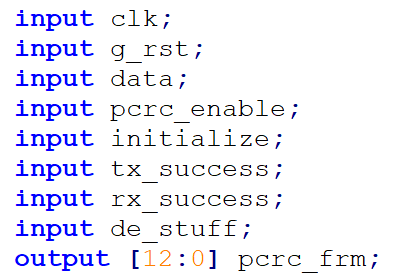




18. Receiver PCRC :

This module calculates the PCRC value by a polynomial division with polynomial value of hexadecimal 19E7 and the remainder from this will be the calculated PCRC value that must be appended to the CAN-XL frame.



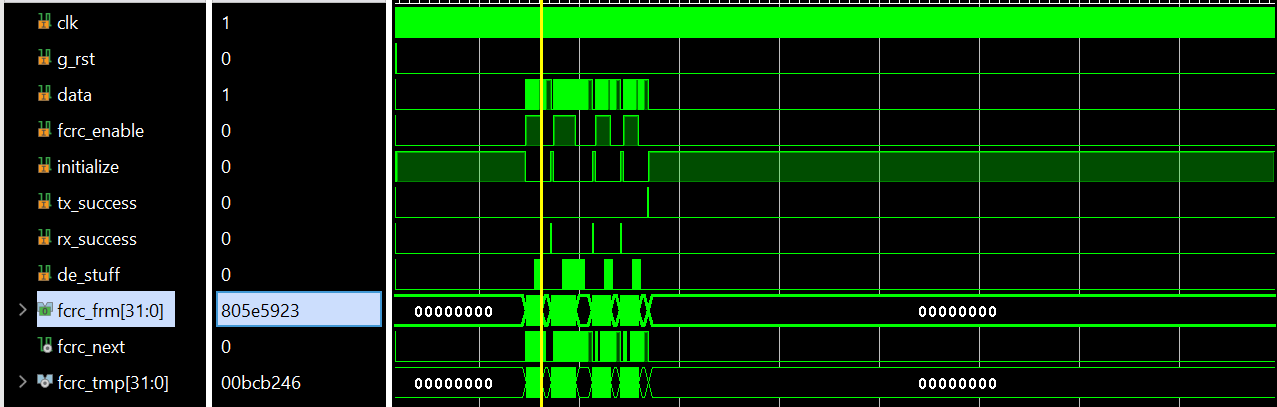




The above image is the polynomial division and “pcrc\_frm” will be the appended PCRC to the CAN-Xl frame.

19. Receiver FCRC :

This module calculates the FCRC value by a polynomial division with polynomial value of hexadecimal FA567D89 and the remainder from this will be the calculated FCRC value that must be appended to the CAN-XL frame.



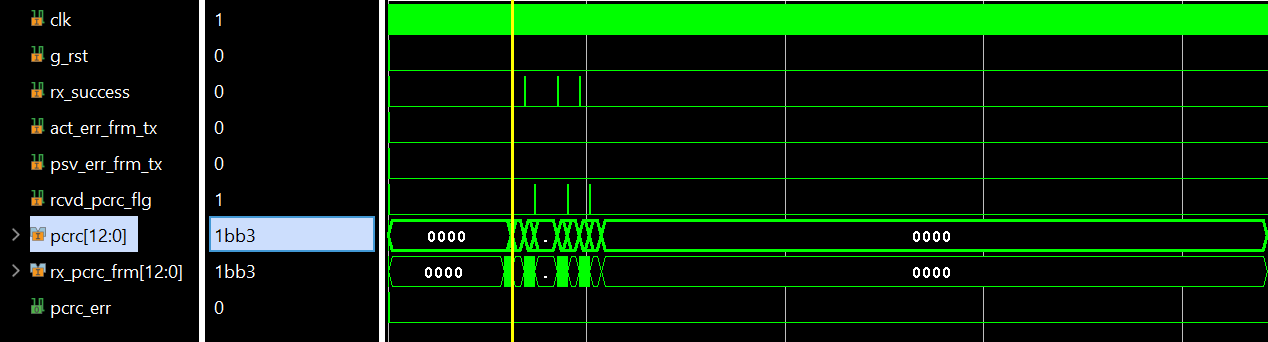


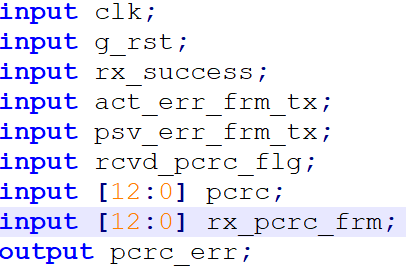


The above image is the polynomial division and “fcrc\_frm” will be the appended FCRC to the CAN-Xl frame.

20. PCRC Checker :

This module checks if the value of calculated PCRC on receiver side matcher with the PCRC value that has been transmitted. If value does not match it will lead to and “pcrc\_err”.

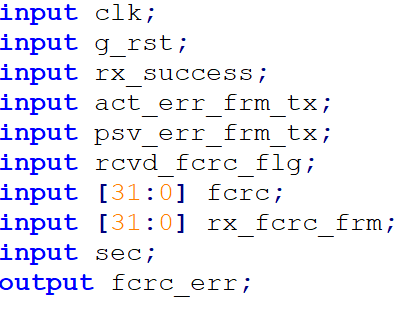




21. FCRC Checker :

This module checks if the value of calculated FCRC on receiver side matcher with the PCRC value that has been transmitted. If value does not match it will lead to and “fcrc\_err”.





22. Form Checker :

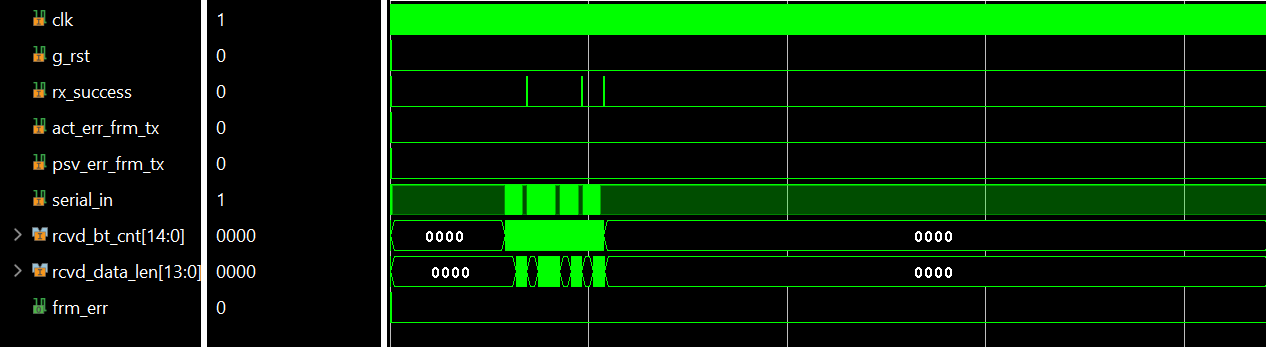
The form\_checker module is responsible for detecting form errors in a CAN-XL frame. A form error occurs when specific bits in the frame are not in the expected format, particularly in fields like the acknowledgment delimiter, and end-of-frame (EOF) segment or if it the number of bits exceed the limit.The module continuously checks the bit count (rcvd\_bt\_cnt) and serial\_in to identify form errors at specific positions in the frame.

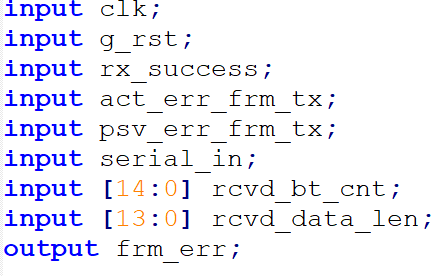
If any of the following conditions match and serial\_in is not recessive (1), it sets frm\_err high:

(rcvd\_bt\_cnt == (12'd97 + rcvd\_data\_len + 12'd32 + 12'd8 + 12'd2))

(rcvd\_bt\_cnt >= (12'd97 + rcvd\_data\_len + 12'd32 + 12'd8 + 12'd3))

(rcvd\_bt\_cnt <= (12'd97 + rcvd\_data\_len + 12'd32 + 12'd8 + 12'd9 ))

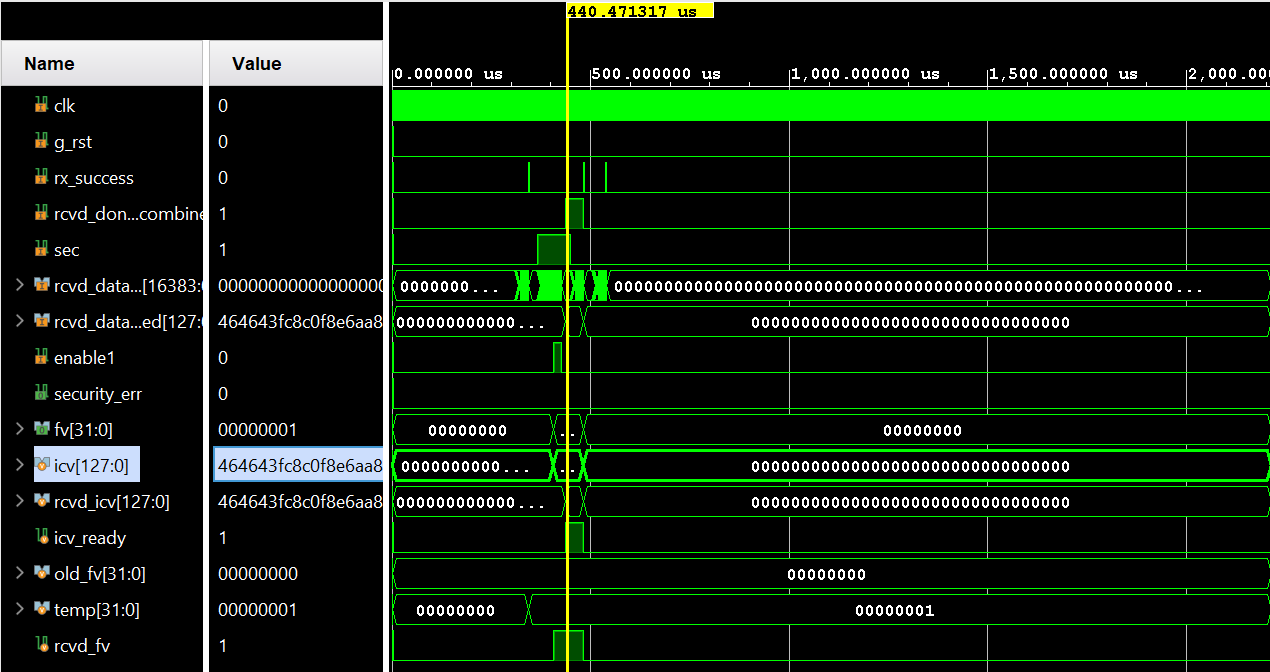


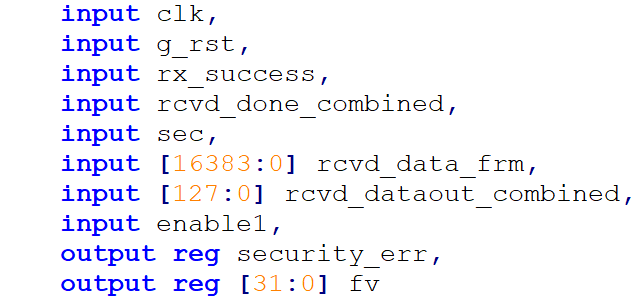


23. Security by AES-128 at Receiver side :

This module recalculates the ICV value and matches with the ICV value that has been transmitted if they are not the same it will indicate that data payload has been tampered with and will result in “security\_err”.

This module also checks if the FV (Freshness Value ) is higher than the previous FV value if not it will indicate error.

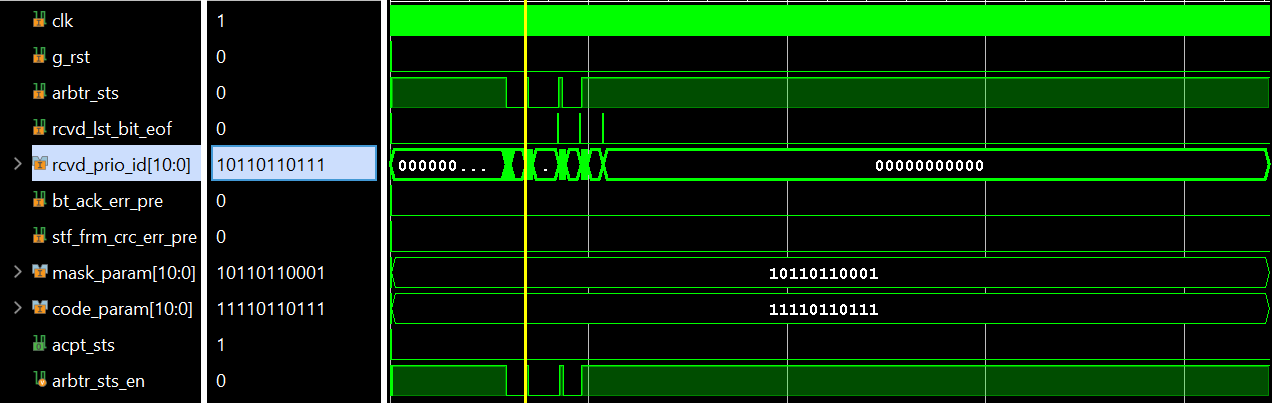


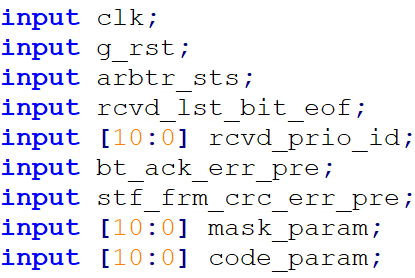


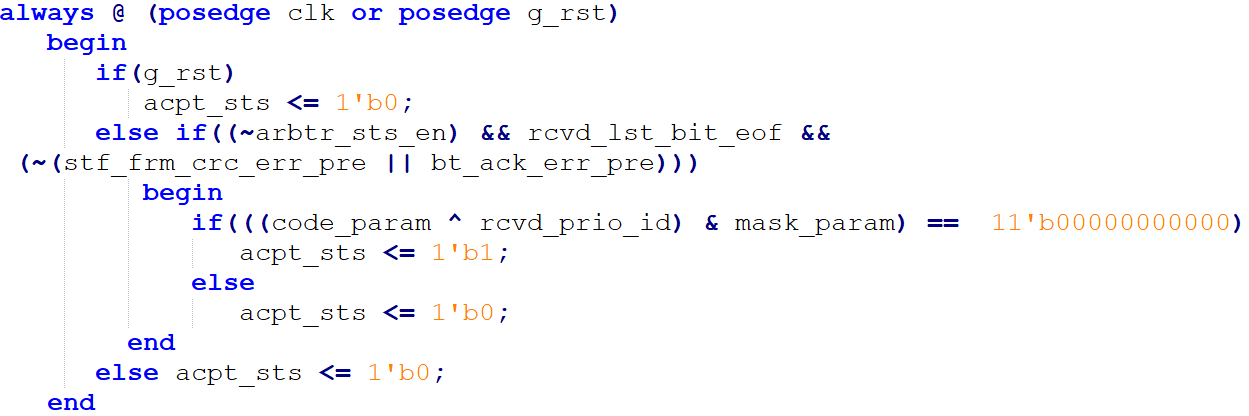
**“FV value is not fully operational as it requires extensive memory management system which is not possible to implement as the thesis code cannot handle it , it is required to be there as each node will have different FV values and each CAN-XL frame may have different FV values .”**

24. Acceptance Checker :

The Acceptance Checker in the controller determines whether an incoming message is relevant to the node. It examines the message identifier in the arbitration field of the received frame and compares it against the acceptance filters, which are configured using the code and mask parameters. If the identifier matches the configured criteria, the message is accepted and forwarded to the receive buffer for processing by the host CPU.







In the above code as we get “code\_param” and “rcvd\_prio\_id” the same XORing them will result in 0 which ANDed with mask\_param will lead to 0 itself , only if this happens signal “acpt\_sts” will be asserted high because of which the receiver buffer will take in the data.

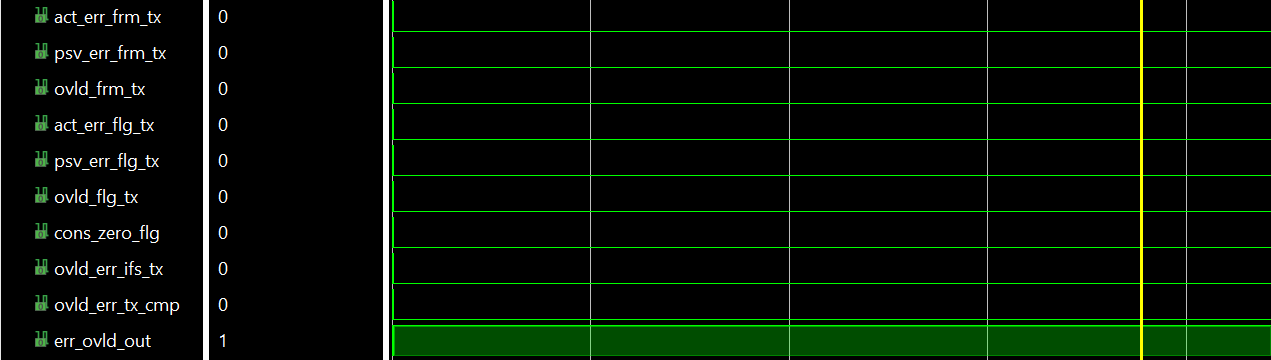
25. Overload Frame Generator :

The Overload Frame Generator in the CAN controller creates overload frames when the controller needs to delay the transmission of subsequent messages. This situation typically arises when the receive buffers are full, and the controller requires additional time to process the current data before accepting new messages. The overload frame consists of a predefined sequence of bits that temporarily halts bus activity, signalling all nodes to wait before resuming communication. By managing bus delays, the Overload Frame Generator ensures that no messages are lost and that the controller can handle data efficiently without being overwhelmed.

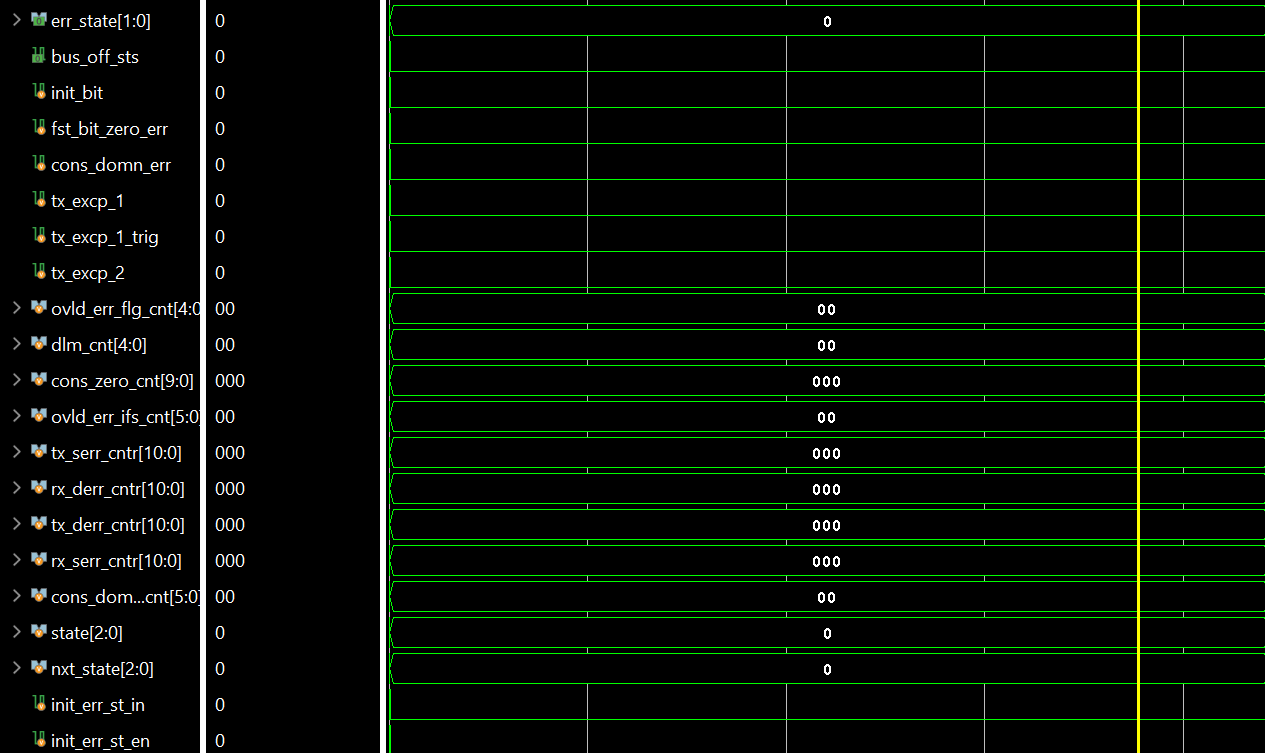


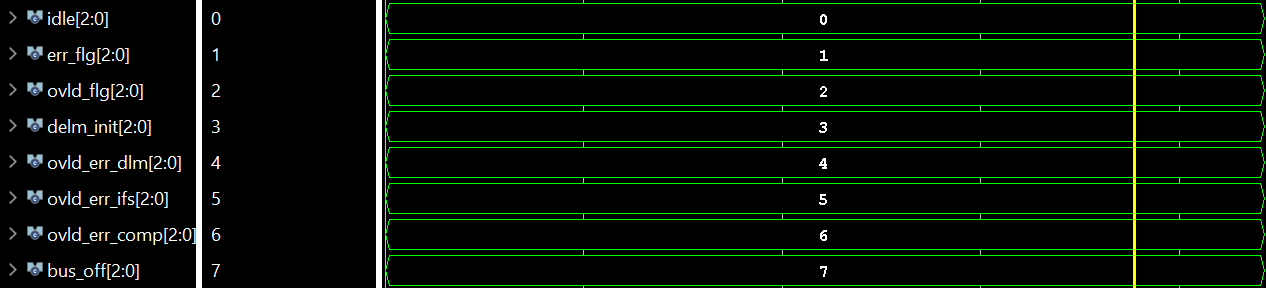
In the above image it is seen that no error has come or an overload signal has come throughout the end of simulation which is the correct way of working. It is also seen that there is on “tx\_success” and 3 “rx\_success” as intended and signal “txtmr” indicates that that during the period when it is asserted high the node is transmitting data.

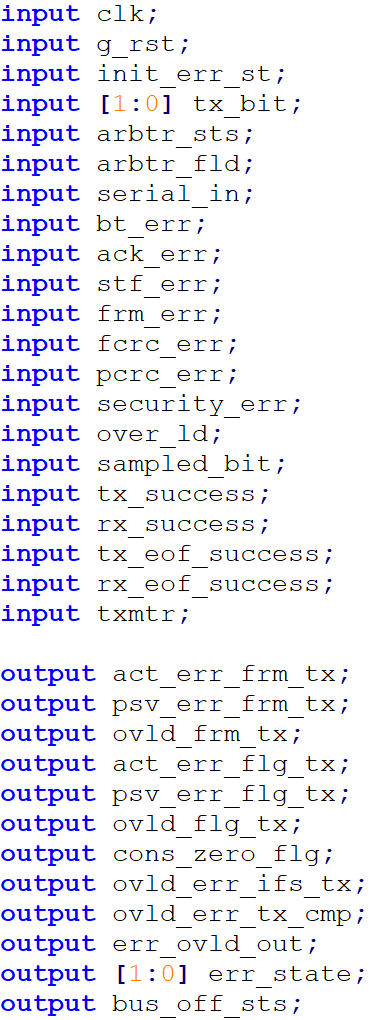
**“The Overload Frame Generator is the same code as the original thesis code”**

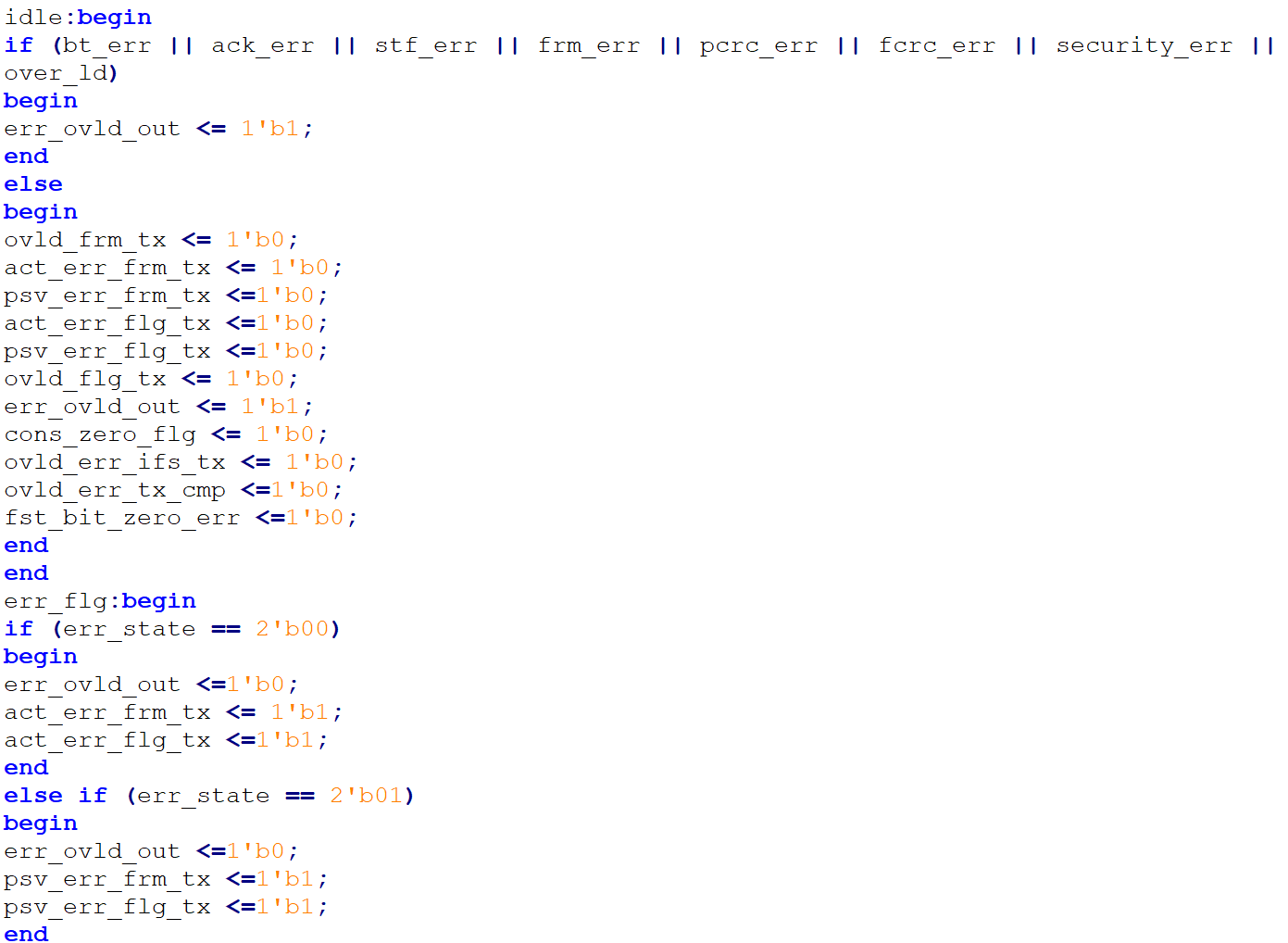


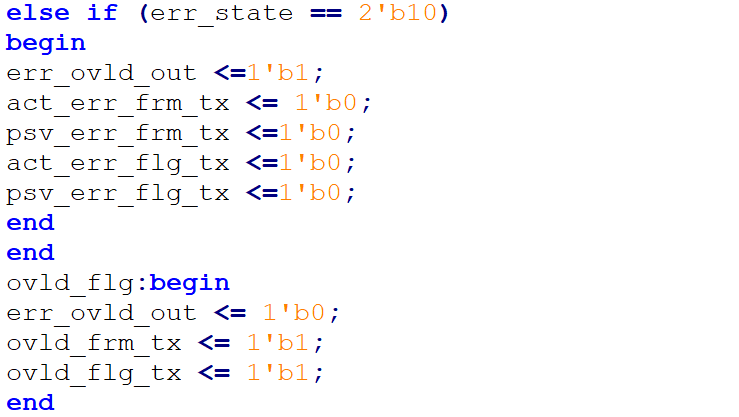
In the above image the signals “act\_err\_frm\_tx” which is for active error and “psv\_err\_frm\_tx” which is for passive error and “ovld\_frm\_tx” which is for overload error are low as no error is detected and the flag singals are the same. The “cons\_zero\_flg” tracks the occurrence of consecutive dominant bits (0) in the bit stream which also helps in finding any stuff error. The signal “err\_ovld\_out” is used to indicate the generation of an error frame or an overload frame. The value of err\_ovld\_out will typically be asserted low when either an error frame or an overload frame is being generated.





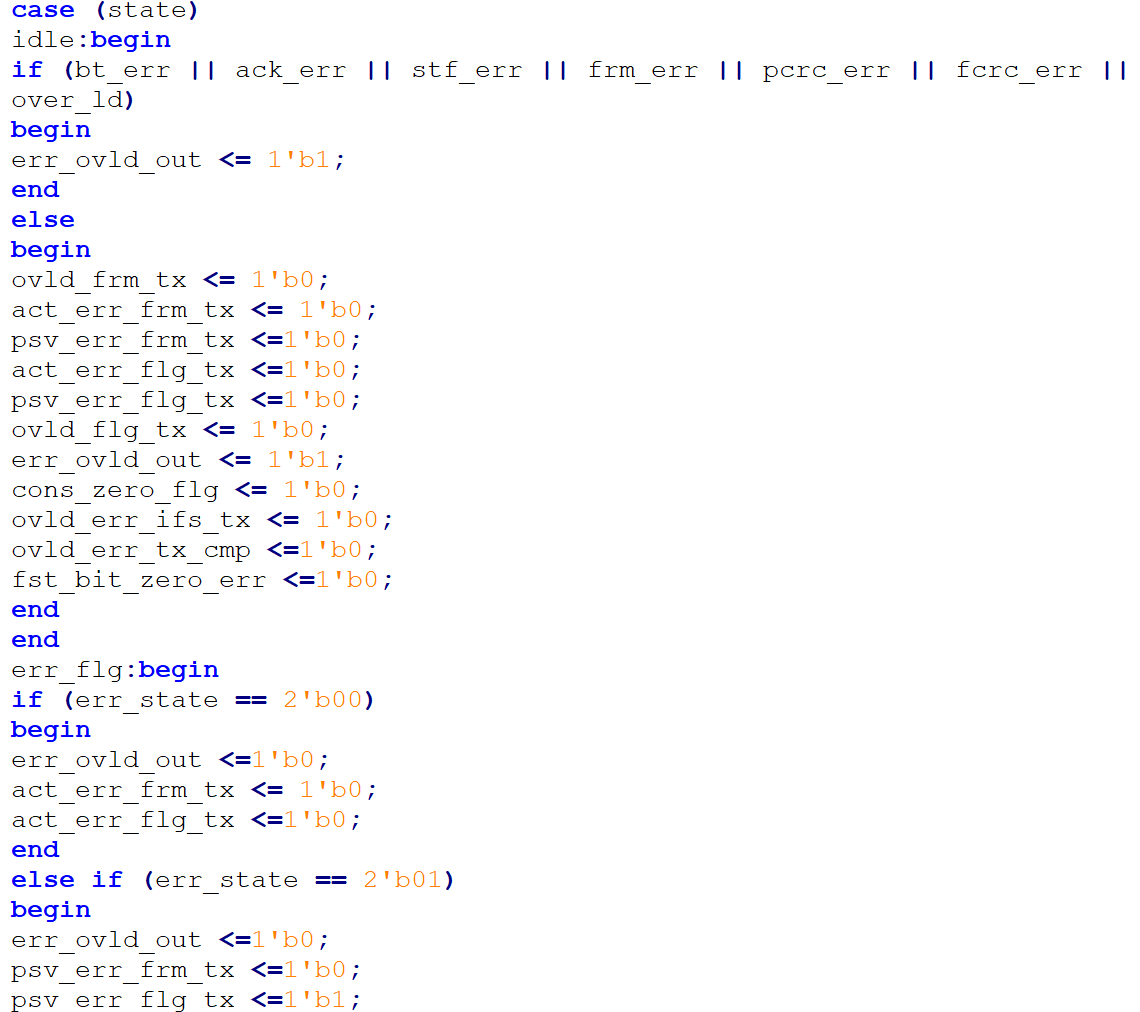


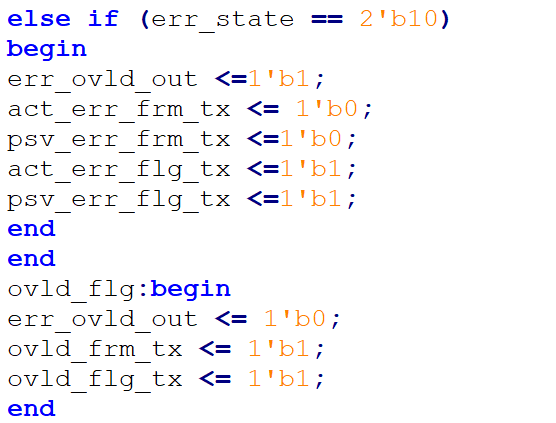




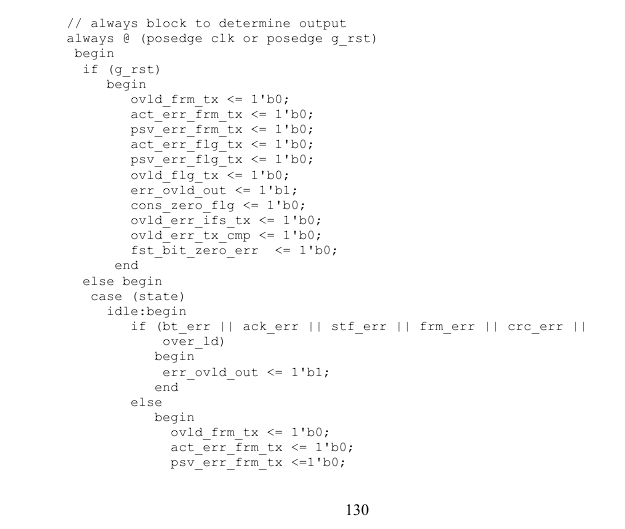
**“IF THESE ARE MADE IN CODE TO BE 0 AND NEVER TO BE ASSERTED THEN THE IMPACT OF ANY ERROR WILL BE NEGLECTED EVEN IF ANY ERROR COMES AS THESE SIGNALS ACT AS THE BARRIER THAT INDICATE THE CODE IS WORKING CORRECTLY , IF THESE SIGNALS COME THEN CODE GOES INTO ERROR STATE AND WILL WANT RETRANSMISSION AND WILL GO IN LOOP AS CAN BUS IN IS THE SAME AS CAN BUS OUT WHICH IS MADE LIKE THAT IN THE TESTBENCH AS THE ORGINAL CODE”.**

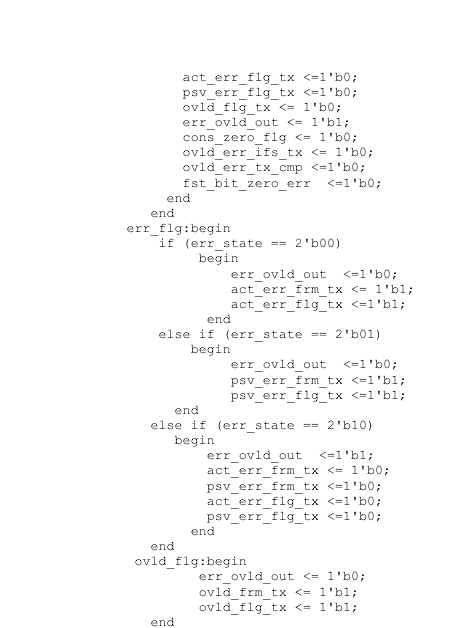
**EXAMPLE OF WRONG/FALSE CODE IS SHOWN BELOW :**

****

****

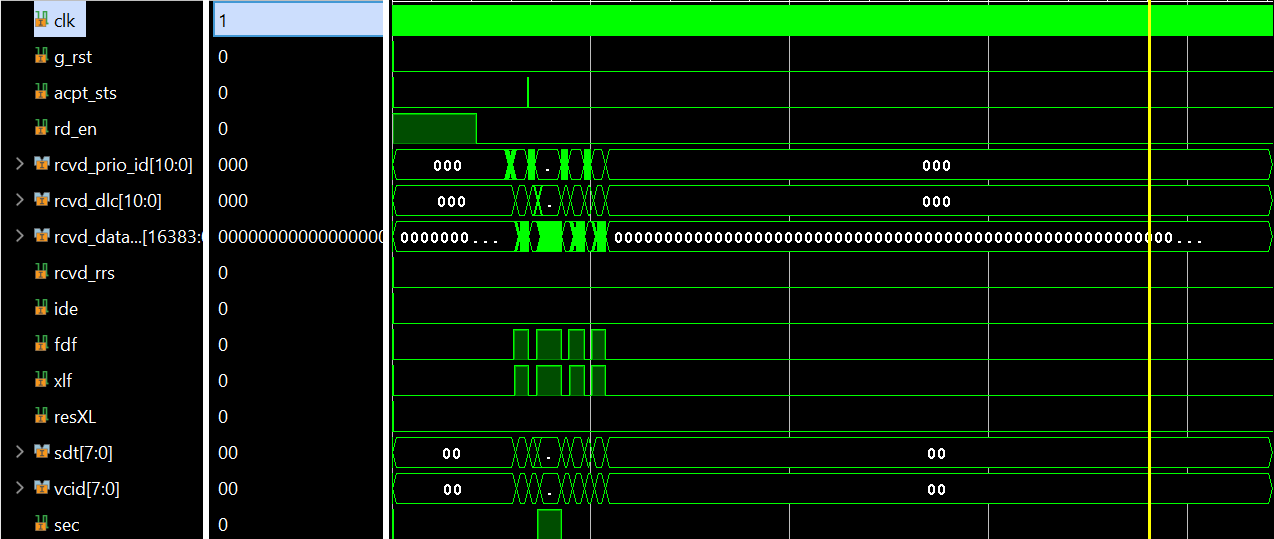
**THESIS CODE :**

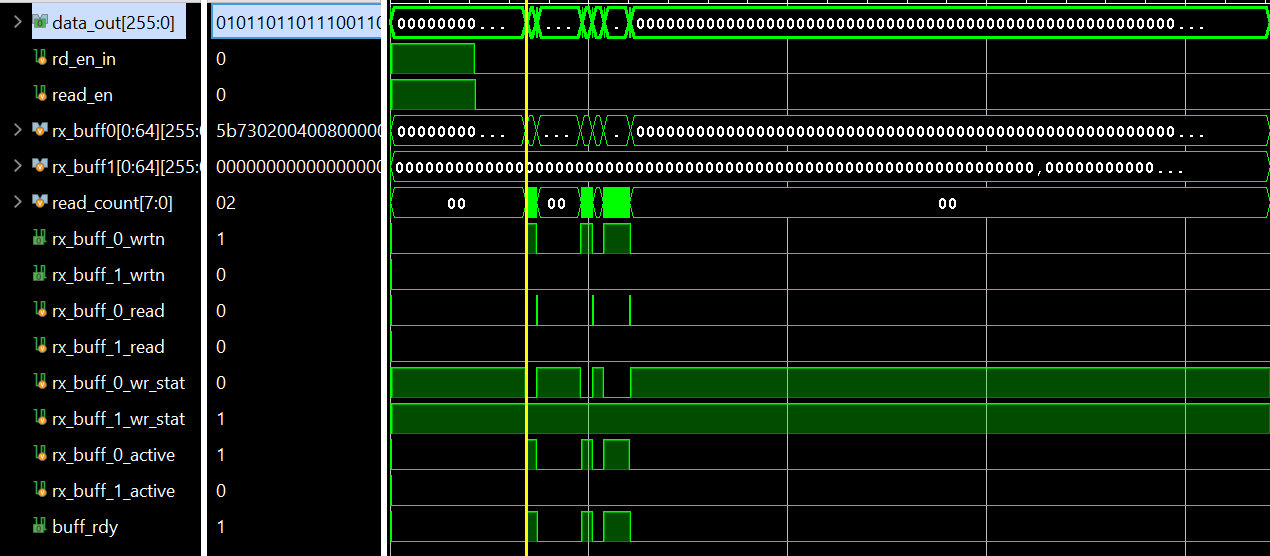
****

****

25. Receiver Buffer :

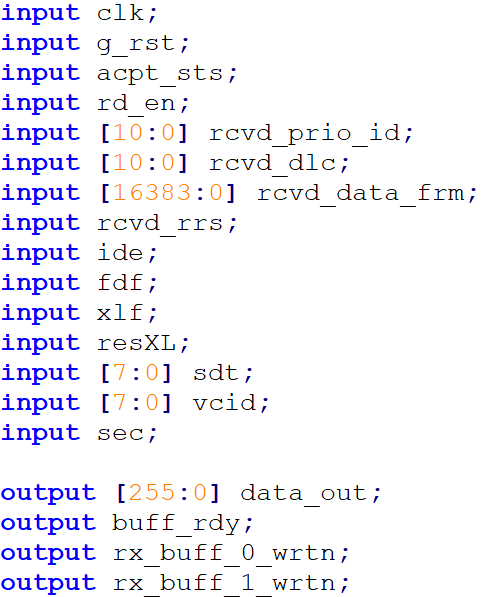
The Receiver Buffer in the controller temporarily stores incoming messages from the CAN bus before they are processed by the host CPU. It acts as a staging area where received frames are stored, ensuring that no data is lost even if the host is busy. Typically, the receiver buffer is designed with a limited capacity, such as two or more frames, allowing the controller to process one frame while simultaneously receiving another. The receiver buffer also works in tandem with the Acceptance Checker, which filters out irrelevant messages based on their identifiers. Only accepted messages are stored in the buffer when signal “acpt\_sts” is asserted high.





In the above image we can see the final output to the code is signal “data\_out” which will be sent to the CPU/host.

**“THERE IS NO DECRYPTION CODE CURRENTLY AS THE RECIVER BUFFER TAKES ALL THE DATA AS IT HAS NO IDEA ABOUT WHICH NODE IS SECURITY OR NOT AS IT REQUIRES EXTENSIVE MEMORY MANAGEMENT, SO THE DATA GOES TO HOST ENCRYPTED”**

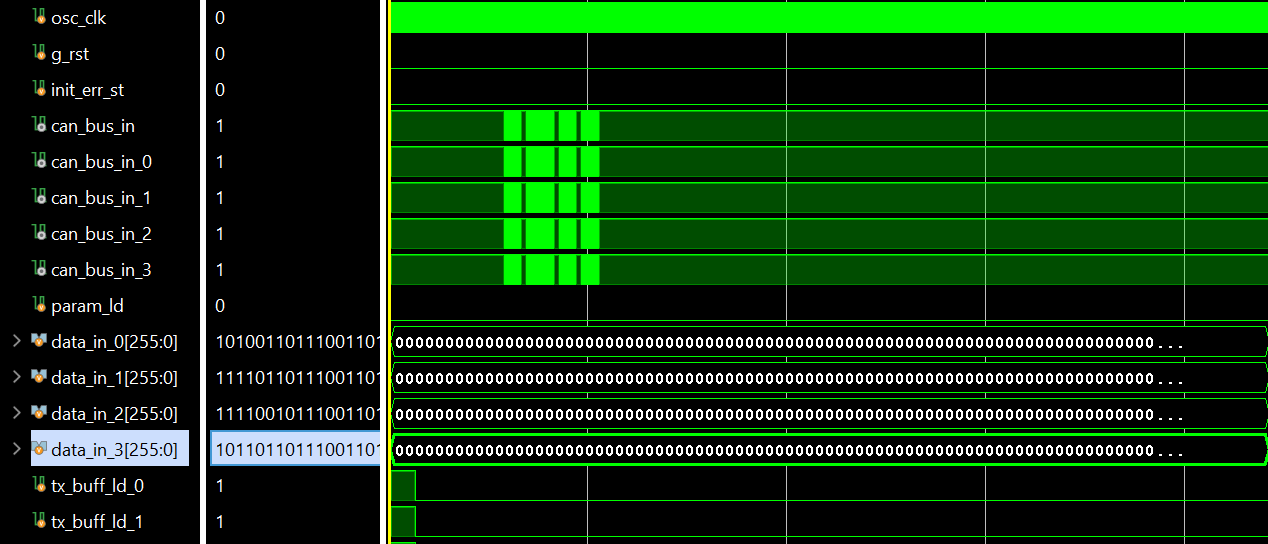


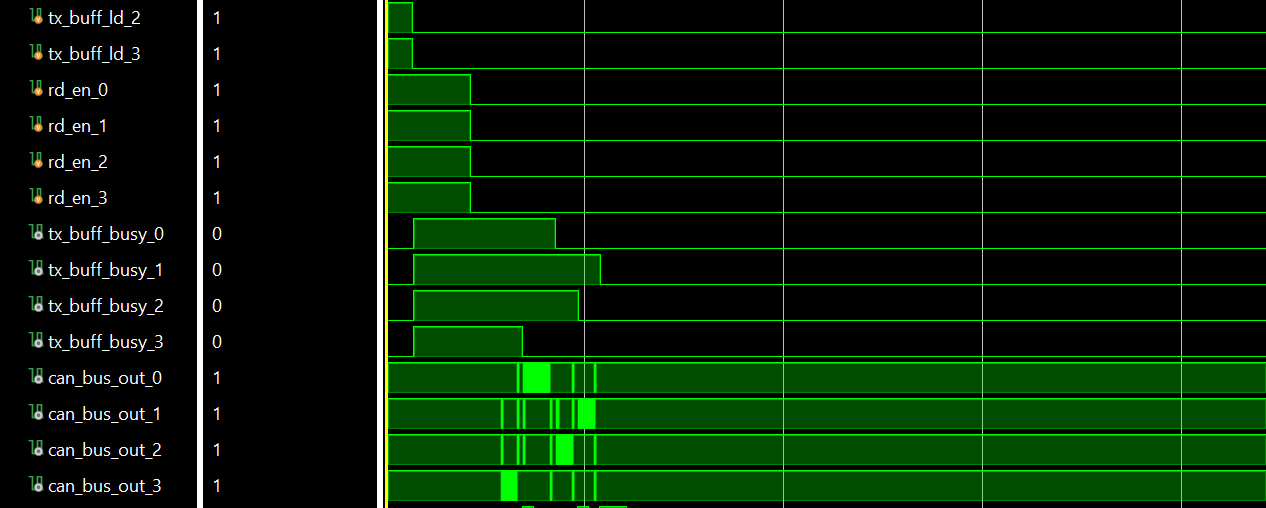


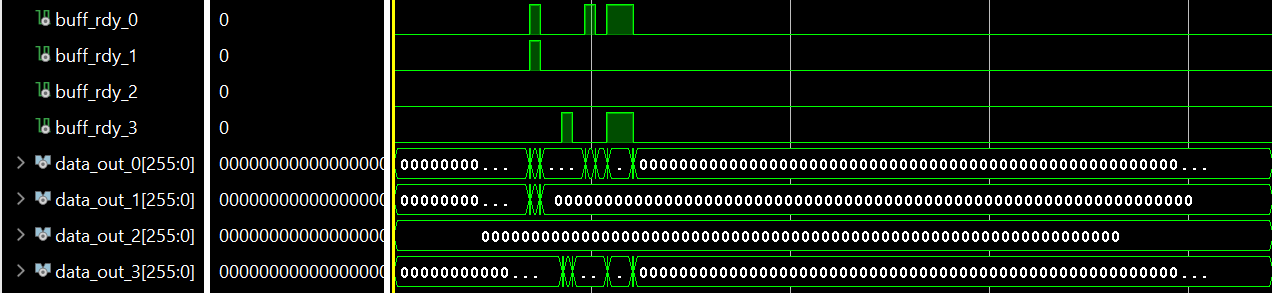


26. Final Output :

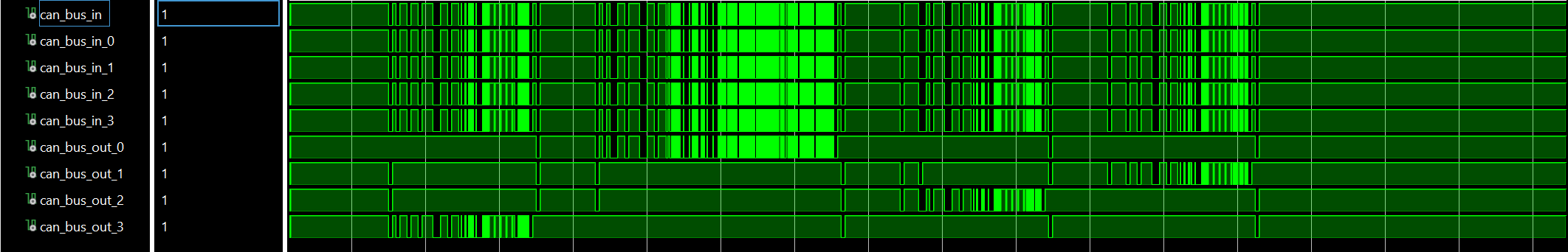
This is the final output as seen by host as this is the top modules final output.







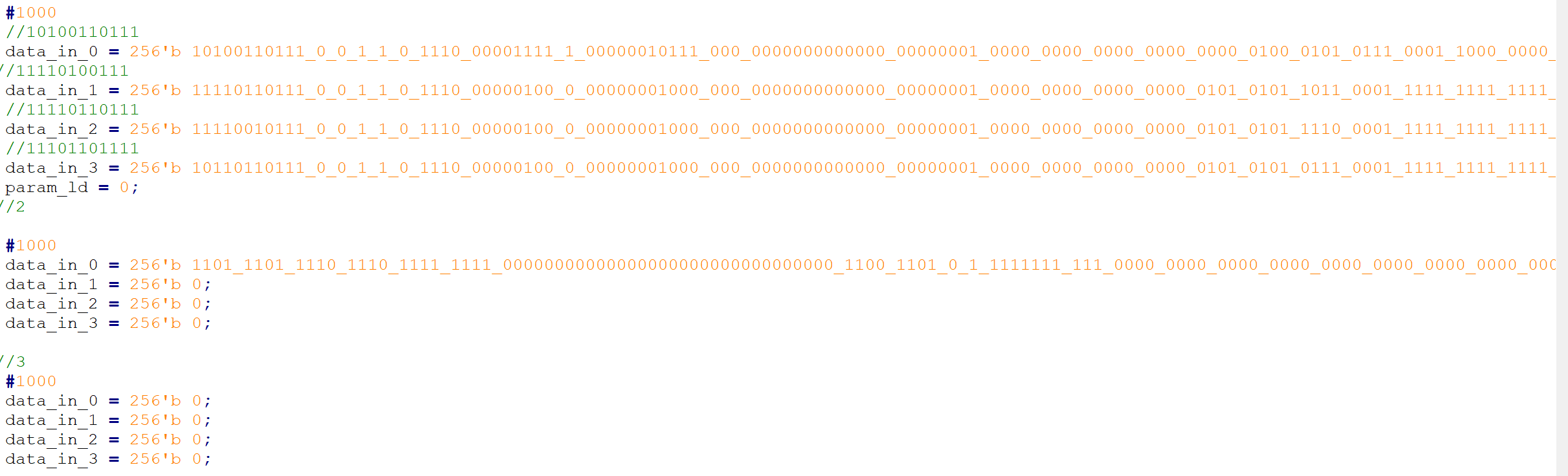
**CAN BUS IN AND CAN BUS OUT IS SHOWN BELOW:**

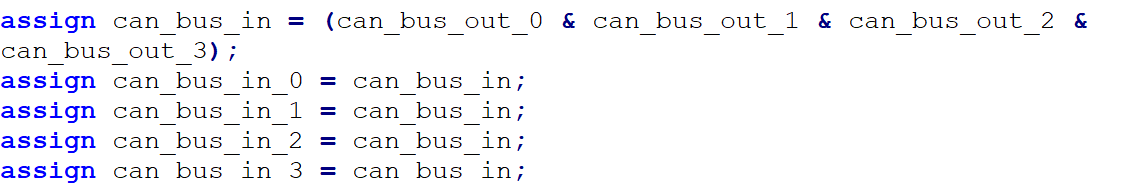
****

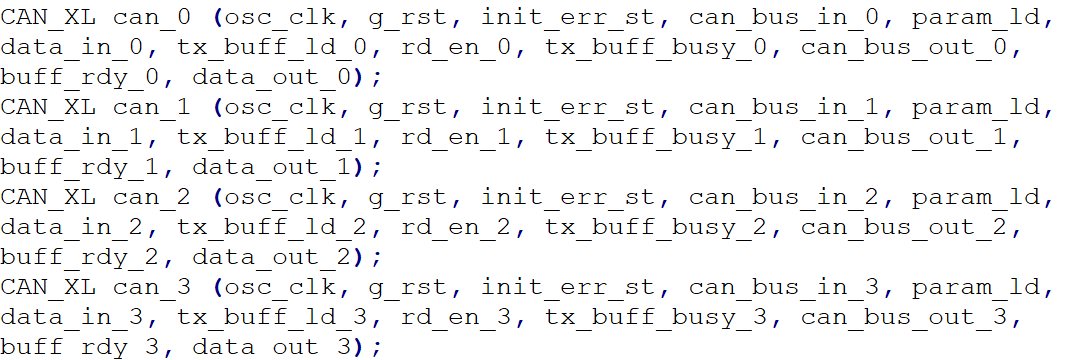
In the above image it is seen that the CAN-XL frames as provided in testbench the 1,3,4 are the nodes which have 64 bits of same data and the 2nd node has the 184 bits of data and is secure.

**The “can\_bus\_out” signal goes down when to indicate acknowledgment is received by the other nodes and it also will go down whenever re-transmission is needed or arbitration is lost. As seen all the “can\_bus\_out” are getting acknowledgement from other nodes indicating code is working correctly as intended.**

27. Testbench :

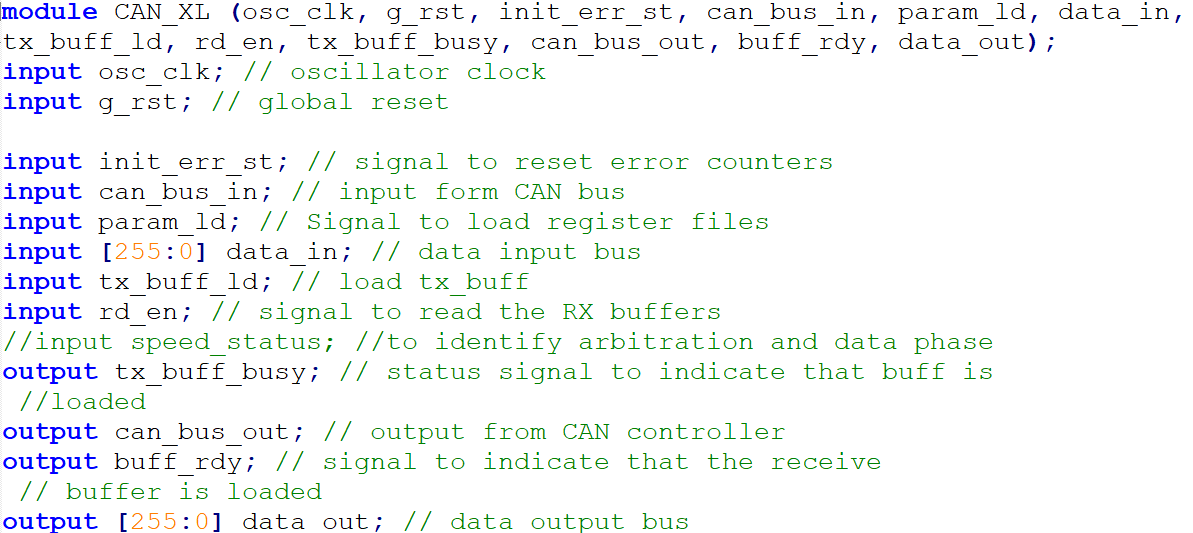


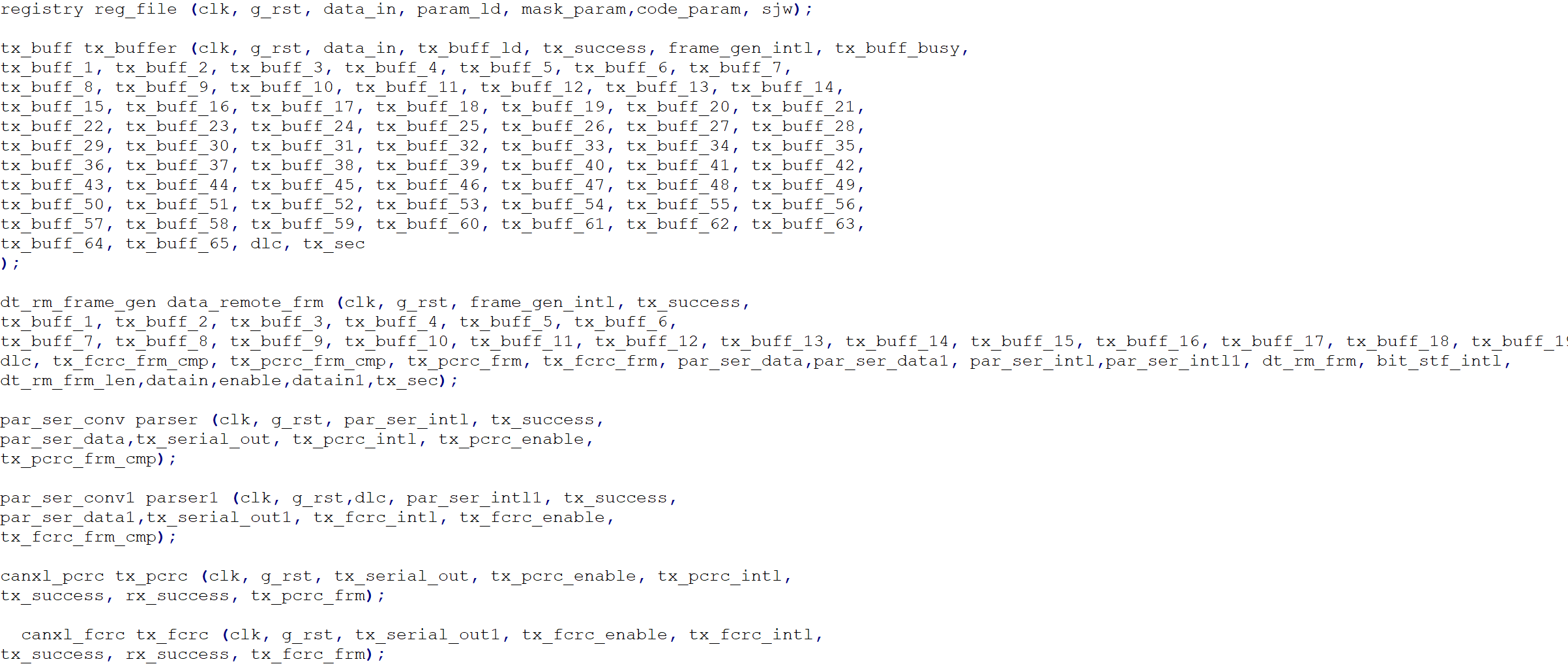


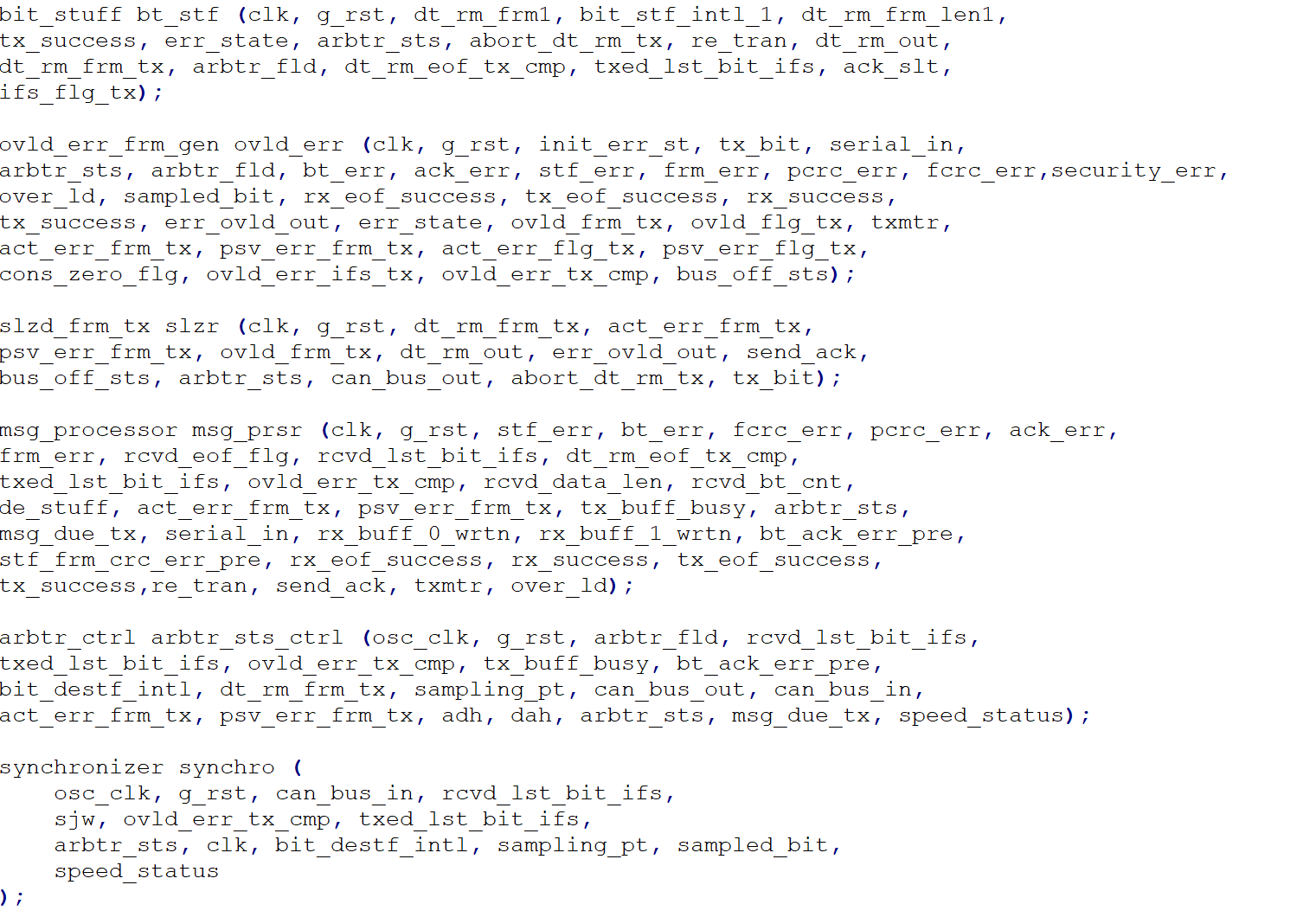


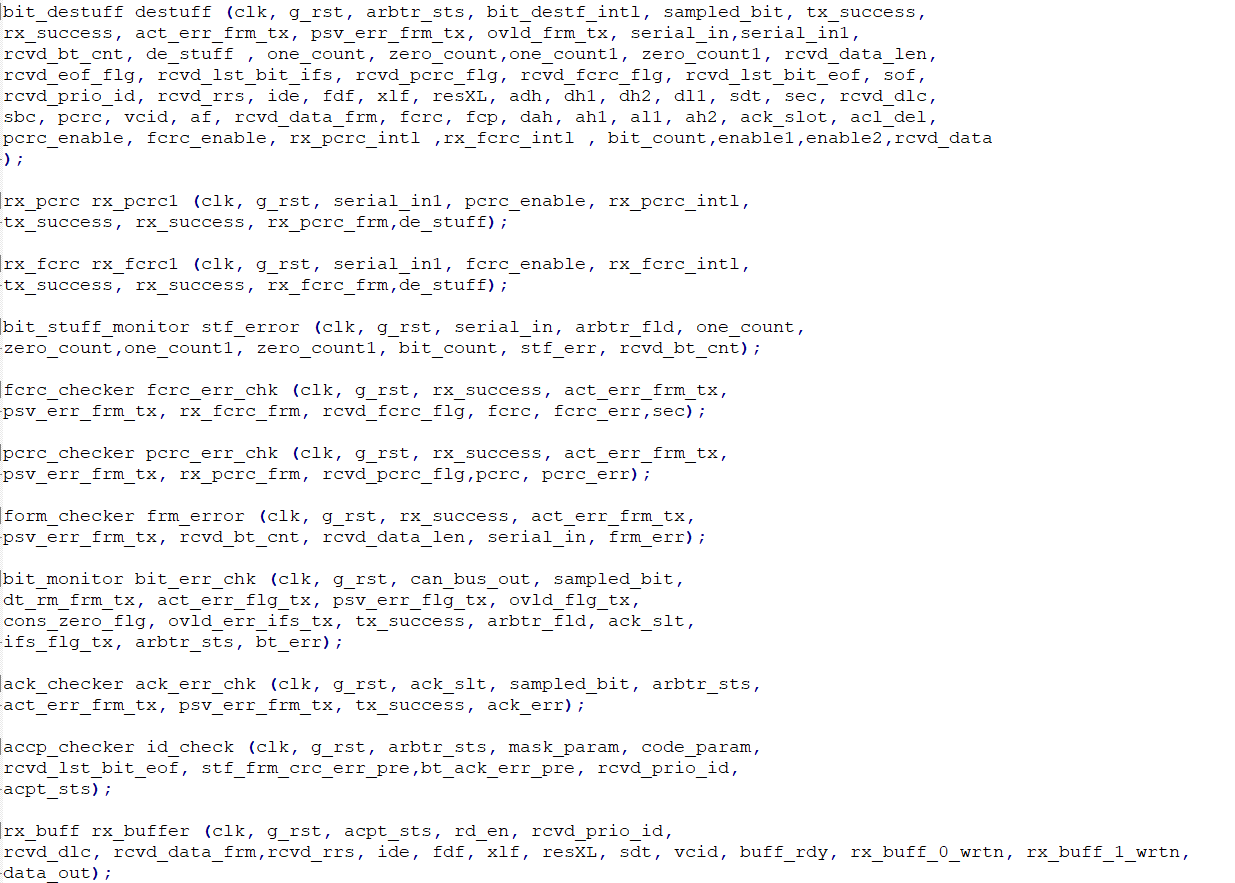
**“In the above image it is seen that the main code is called 4 times”**

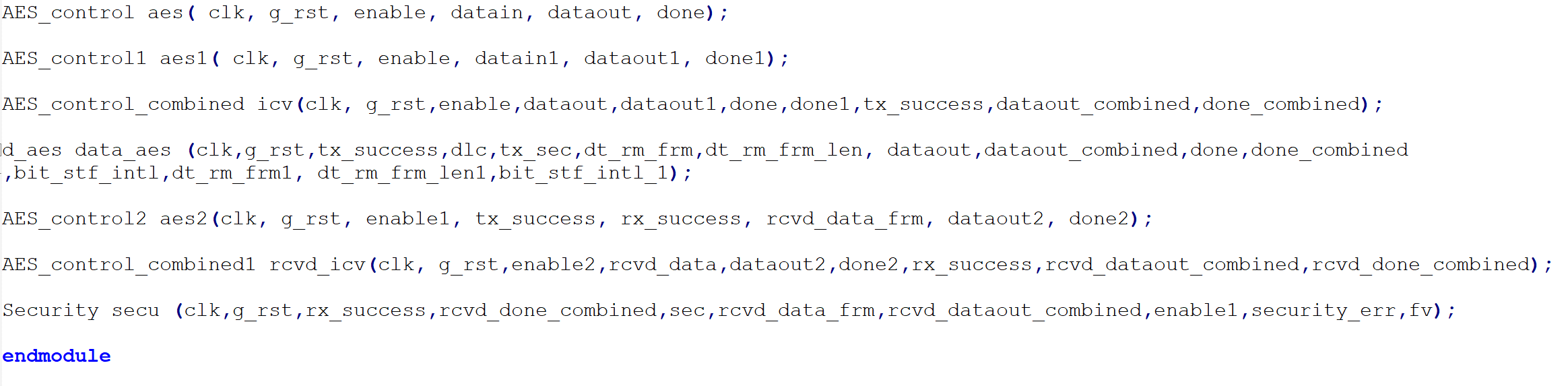
28. Design (Top module) :











THE ABOVE IMAGES SHOW INSTATION OF ALL MODULES AND IS THE TOP MODULE FOR THE ENITRE PROJECT.