Análise, Modelagem e Implementação em Verilog de um Decodificador de Frames do Protocolo CAN 2.0

David Alain do Nascimento, Aluno, CIn/UFPE, e Luís Felipe Prado D'Andrada, Aluno, CIn/UFPE

***Abstract*— CAN is a multi-master serial communication protocol developed by Robert Bosch GmbH. It is widely used in the automotive applications, where it is responsible for, among other things, the communication of real-time control data in Safety-Critical systems. Among the main advantages of CAN are its cost, suitability for Safety-Critical systems, widely deployed, and the need for only one unshielded twisted pair cable.**

For an implementation of a CAN controller it is necessary to implement some modules, such as Bit Timing Logic and Decoder. The Decoder is the target of this work. One function of a CAN decoder is to identify the types of frames received by sorting all received data. The implemented decoder also treats Bit Stuffing, a technique used to ensure a synchronization between the nodes of the network, and CRC, which is used to identify errors in data transmission. To implement the decoder, we designed a FSM (Finite State Machine) which is the core of this work.

***Index Terms*—CAN, Controller Area Network, Verilog, FPGA, Máquina de estados.**

I. Introdução

CAN (Controller Area Network) é um protocolo de comunicação serial multi-master desenvolvido pela Robert Bosch GmbH e lançado em 1986 [1]. É principalmente utilizado em carros para comunicação de dados de controle de tempo real em sistemas do tipo Safety-Critical. No CAN, os pacotes (*frame*s) são transmitidos através de um barramento de dados com cabeamento de apenas dois fios [1]. E, por utilizar uma topologia de barramento, poderá haver contenda de *frame*s quando mais de um nó da rede CAN transmitir simultaneamente. Para contornar este problema, o CAN implementa como mecanismo de arbitração o protocolo *Carrier Sense Multiple Access Collision Resolution* (CSMA/CR) que soluciona o problema da contenda de *frame*s de maneira determinística, em que durante a arbitração 'vence', ou seja, é transmitido por completo, apenas o *frame* que possui a maior prioridade e os outros *frame*s param de ser transmitidos. O *frame* de maior prioridade é o que possuir o campo de arbitração com menor valor, em outras palavras, é o *frame* que possuir mais bits dominantes (valor 0) no campo de arbitração, comparando-se bit a bit da esquerda para direita [2].

O CAN possui diferentes versões: 1.0, 1.1, 1.2, 2.0 e CAN-FD [3]. A versão 2.0, lançada em 1991, possui duas especificações: 2.0A, para compatibilidade com a versão anteriores 1.x e a versão 2.0B, que possui como principal diferença um *frame* extendido com identificador de 29 bits [1] [3] [4].

Diferentemente dos mais comuns protocolos de comunicação, no CAN os frames utilizam apenas um único identificador e este identificador não é utilizado no endereçamento de um nó da rede, mas sim de um tipo de dado que está sendo transmitido, normalmente um determinado parâmetro de funcionamento do sistema de um automóvel, como por exemplo a rotação do motor, velocidade, nível de combustível, como também dados de controle como aceleração, freios ABS, airbags, etc [4] [1].

O CAN utiliza o *Non-Return to Zero* (NRZ) como mecanismo de codificação na transmissão dos bits no barramento, e por conta disso, podem acontecer erros de sincronismo da duração do bit após vários bits de mesmo valor serem transmitidos. Para solucionar este problema, o CAN implementa um mecanismo chamado *Bit Stuffing*, em que após cinco bits sequenciais de mesmo valor, é inserido um bit de valor oposto após o quinto bit sequencial [4].

II. Frames CAN 2.0

A versão 2.0 possui quatro diferentes tipos de *frame*s: *Data Frame*, *Remote Frame*, *Error Frame* e *Overload Frame* [4]. Este projeto demonstra uma modelagem e implementação de um decodificador de *frame*s CAN 2.0 que reconhece os quatro diferentes tipos de *frame*s, como também os bits transmitidos entre os *frame*s, chamado de *Interframe Spacing*.

A. Data Frame

O *Data Frame* pode ter dois formatos: o *Base Frame*, que possui identificador de 11 bits, definido na versão 2.0A, e o *Extended Frame*, que possui identificador de 29 bits, definido na versão 2.0B.

1) Base Frame

O *Base Frame* é composto respectivamente pelos seguintes campos: SOF (*Start of Frame*), *Identifier*, RTR (*Remote Transmission Request*), IDE (*Identifier Extension bit*), Reserved Bit, DLC (*Data Length Code*), *Data Field*, CRC (*Cyclic Redundancy Check*), CRC *Delimiter*, ACK *Slot*, ACK *Delimiter* e EOF (*End of Frame*).

a) SOF (Start of Frame)

- Descrição: O SOF determina o início de um *frame* de dados.

- Tamanho do campo: 1 bit.
- Valores esperados: O único valor possível para o *Start of Frame* é dominante (0).

b) Identifier

- Descrição: Um identificador único para um determinado tipo de mensagem a ser enviada. Seu valor é utilizado na arbitração e, por isso, é responsável pela prioridade de mensagens.
- Tamanho do campo: 11 bits.
- Valores esperados: Qualquer valor é aceitável, porém os 7 bits mais significativos não podem ser todos recessivos (1).

c) RTR (Remote Transmission Request)

- Descrição: Campo que identifica se uma mensagem é um *Data Frame* ou um *Remote Frame*.
- Tamanho do campo: 1 bit.
- Valores esperados: É utilizado o valor dominante (0) para um *Data Frame* e valor recessivo (1) para um *Remote Frame*.

d) IDE (Identifier Extension bit)

- Descrição: Identifica o tipo de *frame* de dados, Base Frame ou Extended Frame.
- Tamanho do campo: 1 bit.
- Valores esperados: Para um Base Frame, o valor deste campo deve ser sempre dominante (0). Caso contrário, o *frame* será um Extended Frame.

e) Reserved Bit

- Descrição: Bit reservado para uso futuro.
- Tamanho do campo: 1 bit.
- Valores esperados: Deve ser dominante (0), porém, caso um receptor receba um bit recessivo (1), não deve ser gerado nenhum tipo de erro.

f) DLC (Data Length Code)

- Descrição: Define a quantidade de dados que serão enviados no campo Data Field, de forma que o valor recebido neste campo é igual a quantidade de bytes que serão recebidos no campo Data Field.
- Tamanho do campo: 4 bits.
- Valores esperados: São esperados valores de 0 a 8. Como o campo possui 4 bits, seria possível preenchê-lo com valores até 15, entretanto caso um receptor receba um valor maior que 8, deve ser considerado apenas 8 e não deve ser gerado nenhum tipo de erro [4].

g) Data Field

- Descrição: Dados da mensagem.
- Tamanho do campo: É definido pelo valor recebido no campo. Pode variar de 0 a 64 bits. Sempre em múltiplo de 8.
- Valores esperados: Não há restrição para os valores a serem enviados no campo de dados.

h) CRC (Cyclic Redundancy Check)

- Descrição: Utilizado para identificar erros na transmissão dos dados.
- Tamanho do campo: 15 bits.
- Valores esperados: É esperado o valor resultante do cálculo de CRC. Um valor diferente deste levará a um CRC Error, que deve ser enviado apenas após o ACK Delimiter.

i) CRC Delimiter

- Descrição: É o bit seguinte ao CRC.
- Tamanho do campo: 1 bit.
- Valores esperados: É esperado um valor recessivo (1) para o CRC Delimiter. Um valor dominante (0) deve levar a um Form Error.

j) ACK Slot

- Descrição: Ao receber uma mensagem corretamente, um receptor deve enviar, através deste campo, sua confirmação de recebimento.
- Tamanho do campo: 1 bit.
- Valores esperados: O transmissor deverá enviar este campo com o valor recessivo (1). Todos os recepetores ao receberem o frame corretamente deverão transmitir o bit dominante (0) durante o tempo do ACK Slot. Quando um receptor recebe uma mensagem corretamente o valor esperado no barramento é dominante (0), indicando que pelo menos um nó reconheceu o frame corretamente. Quando nenhum receptor responde, ou seja, deixa o barramento em recessivo (1), o transmissor irá emitir um ACK Error.

k) ACK Delimiter

- Descrição: É o bit seguinte ao ACK Slot. Faz com que o ACK Slot seja cercado de bits recessivos.
- Tamanho do campo: 1 bit.
- Valores esperados: É esperado um valor recessivo (1) para o ACK Delimiter. Um valor dominante (0) deve levar a um Form Error.

l) EOF (End of Frame)

- Descrição: Determina o fim do *frame* de dados.
- Tamanho do campo: 7 bits.
- Valores esperados: Todos os bits devem ser recessivos (1), porém, para receptores, o último bit deve ser tratado como *don't care*, ou seja, pode ser tanto dominante (0) quanto recessivo (1). Um valor diferente do esperado nos 6 primeiros bits deve levar a um Form Error [2].

2) Extended Frame

O Extended Frame é composto pelos respectivos campos: SOF (*Start of Frame*), Identifier A, SRR (Substitute remote request), IDE (Identifier Extension bit), Identifier B, RTR (Remote transmission request), Reserved Bits, DLC (Data

Length Code), Data Field, CRC (Cyclic Redundancy Check), CRC Delimiter, ACK Slot, ACK Delimiter e EOF (*End of Frame*).

a) SOF (Start of Frame)

- Descrição: O SOF determina o início de um *frame* de dados.
- Tamanho do campo: 1 bit.
- Valores esperados: O único valor possível para o *Start of Frame* é dominante (0).

b) Identifier A

- Descrição: A primeira parte do identificador único para a mensagem a ser enviada. Seu valor é utilizado na arbitração e, por isso, é responsável pela prioridade de mensagens.
- Tamanho do campo: 11 bits.
- Valores esperados: Qualquer valor é aceitável, porém os 7 bits mais significativos não podem ser todos recessivos (1).

c) SRR (Substitute Remote Request)

- Descrição: Campo que substitui o RTR, em termos de localização no *frame*, no formato do Extended Frame.
- Tamanho do campo: 1 bit.
- Valores esperados: Deve ser recessivo (1), porém, caso um receptor receba um bit dominante (0), não deve ser gerado nenhum tipo de erro.

d) IDE (Identifier Extension Bit)

- Descrição: Identifica o tipo de *frame* de dados, Base Frame ou Extended Frame.
- Tamanho do campo: 1 bit.
- Valores esperados: Para um Extended Frame, o valor deste campo deve ser sempre recessivo (0). Caso contrário, o *frame* será um Base Frame. É interessante notar que, através dos valores codificados para os dois tipos de *frame*s de dados, o Base Frame tem maior prioridade que o Extended Frame.

e) Identifier B

- Descrição: A segunda parte do identificador único para a mensagem a ser enviada. Seu valor é utilizado na arbitração e, por isso, é responsável pela prioridade de mensagens.
- Tamanho do campo: 18 bits.
- Valores esperados: Não há restrição para os valores a serem enviados no campo de dados.

f) RTR (Remote Transmission Request)

- Descrição: Campo que identifica se uma mensagem é um *Data Frame* ou um *Remote Frame*.
- Tamanho do campo: 1 bit.
- Valores esperados: É utilizado o valor dominante (0) para um *Data Frame* e valor recessivo (1) para um *Remote Frame*.

g) Reserved Bits

- Descrição: Bits reservado para uso futuro.
- Tamanho do campo: 2 bits.
- Valores esperados: Ambos os bits devem ser dominantes (0), porém, caso um receptor receba qualquer valor diferente, não deve ser gerado nenhum tipo de erro.

h) DLC (Data Length Code)

- Descrição: Define a quantidade de dados que serão enviados no campo Data Field, de forma que o valor recebido neste campo é igual a quantidade de bytes que serão recebidos no campo Data Field.
- Tamanho do campo: 4 bits.
- Valores esperados: São esperados valores de 0 a 8. Como o campo possui 4 bits, seria possível preenchê-lo com valores até 15, entretanto caso um receptor receba um valor maior que 8, deve ser considerado apenas 8 e não deve ser gerado nenhum tipo de erro [4].

i) Data Field

- Descrição: Dados da mensagem.
- Tamanho do campo: É definido pelo valor recebido no campo. Pode variar de 0 a 64 bits, sempre em múltiplo de 8.
- Valores esperados: Não há restrição para os valores a serem enviados no campo de dados.

j) CRC (Cyclic Redundancy Check)

- Descrição: Utilizado para identificar erros na transmissão dos dados.
- Tamanho do campo: 15 bits.
- Valores esperados: É esperado o valor resultante do cálculo de CRC. Um valor diferente deste levará a um CRC ERROR, que deve ser enviado apenas após o ACK Delimiter.

k) CRC Delimiter

- Descrição: É o bit seguinte ao CRC.
- Tamanho do campo: 1 bit.
- Valores esperados: É esperado um valor recessivo (1) para o CRC Delimiter. Um valor dominante (0) deve levar a um Form Error.

l) ACK Slot

- Descrição: Ao receber uma mensagem corretamente, um receptor deve enviar, através deste campo, sua confirmação de recebimento.
- Tamanho do campo: 1 bit.
- Valores esperados: O transmissor deverá enviar este campo com o valor recessivo (1). Todos os recepetores ao receberem o frame corretamente deverão transmitir o bit dominante (0) durante o tempo do ACK Slot. Quando

um receptor recebe uma mensagem corretamente o valor esperado no barramento é dominante (0), indicando que pelo menos um nó reconheceu o frame corretamente. Quando nenhum receptor responde, ou seja, deixa o barramento em recessivo (1), o transmissor irá emitir um ACK Error.

m) ACK Delimiter

- Descrição: É o bit seguinte ao ACK Slot. Faz com que o ACK Slot seja cercado de bits recessivos.
- Tamanho do campo: 1 bit.
- Valores esperados: É esperado um valor recessivo (1) para o ACK Delimiter. Um valor dominante (0) deve levar a um Form Error.

n) EOF (End of Frame)

- Descrição: Determina o fim do *frame* de dados.
- Tamanho do campo: 7 bits.
- Valores esperados: Todos os bits devem ser recessivos (1), porém, para receptores, o último bit deve ser tratado como *don't care*, ou seja, pode ser tanto dominante (0) quanto recessivo (1). Um valor diferente do esperado nos 6 primeiros bits deve levar a um Form Error [2].

B. Remote Frame

O *Remote Frame* possui o mesmo padrão de um *Data Frame*. O *Remote Frame* é utilizado como requisição de um determinado tipo de dado identificado pelo campo *Identifier*, de maneira que apenas o nó responsável por fornecer aquele dado é que irá responder à requisição (com um *Data Frame*). No *Remote Frame* o campo RTR é recessivo (1); o campo DLC indica o tamanho do dado que está sendo requisitado, ou seja, o valor do DLC não representa o tamanho do campo de dados do *Remote Frame*, mas sim o tamanho do campo de dados da resposta (*Data Frame*) que virá daquela requisição; e o campo de dados do *Remote Frame* não é utilizado, tendo sempre comprimento de zero bits.

C. Error Frame

O *Error Frame* é tipo de mensagem transmitida quando um nó detecta um erro e pode ser de dois tipos: Erro Ativo ou Erro Passivo. É composto por dois campos *Error Flags* e *Error Delimiter*.

Um nó ao detectar um erro deverá iniciar a transmissão do *Error Flags* a partir do bit posterior ao qual o erro foi detectado, com exceção do CRC Error que irá iniciar a transmissão do *Error Flags* após o ACK *Delimiter*.

1) Superposição

Por conta de possíveis atrasos de um nó para outro na detecção de um Form Error em um *Data Frame* ou *Remote Frame*, um nó pode iniciar a transmissão do *Error Flags* depois de outro, fazendo com que haja superposição na transmissão do campo *Error Flags*, ou seja, pode existir mais de um nó transmitindo o *Error Flags* simultaneamente. E, no pior dos casos, se um determinado nó não conseguir detectar um Form Error em um *Data Frame* ou *Remote Frame* e apenas vier a conseguir detectar um erro de *Bit Stuffing* na transmissão do *Error Flags* porque ainda estava tratando os bits recebidos como um *Data Frame* ou *Remote Frame*, então este nó (atrasado) irá iniciar a transmissão do *Error Flags* fazendo com que os outros nós que já reconheceram o *Error Flags* transmitidos anteriormente, os recebam novamente. Este é o motivo do campo *Error Flags* ter tamanho variável que vai até o dobro do seu tamanho padrão.

2) Error Flags

- Descrição: Parte inicial do *Error Frame*. Possui tamanho variável por conta da superposição na transmissão do *Error Frame* por diferentes nós ao mesmo tempo com atraso no início da transmissão.
- Tamanho do campo: 6 a 12 bits. Deverá transmitir 6 bits, mas deverá suportar a recepção de até 12 bits por conta da superposição.
- Valores esperados: Todos dominantes (0), para Erro Ativo, todos recessivos (1), para Erro Passivo.

3) Error Delimiter

- Descrição: Parte final do *Error Frame*.
- Tamanho do campo: 8 bits.
- Valores esperados: Todos os bits recessivos (1).

D. Overload Frame

É um tipo de mensagem transmitida em três situações: 1) Quando um nó necessita de um tempo adicional para receber o próximo *Data Frame* ou *Remote Frame*. 2) Detecção de um bit dominante (0) no primeiro ou segundo bit do *Intermission*. 3) Se um nó CAN receber um bit dominante no oitavo bit (último bit) do *Error Delimiter* ou *Overload Delimiter*, então neste último caso será transmitido um *Overload Frame* e não um *Error Frame* (como seria esperado) e o contador de erros não será alterado. O *Overload Frame* é composto pelos campos *Overload Flags* e *Overload Delimiter* [4].

1) Overload Flags

- Descrição: Parte inicial do *Overload Frame*. Possui tamanho variável por conta da superposição, assim como no *Error Frame*. Mais detalhes, vide seção II.C.1) do *Error Frame*.
- Tamanho do campo: 6 a 12 bits. Deverá transmitir 6 bits, mas deverá suportar a recepção de até 12 bits por conta da superposição.
- Valores esperados: Todos dominantes (0).

2) Overload Delimiter

- Descrição: Parte final do *Overload Frame*.
- Tamanho do campo: 8 bits.
- Valores esperados: Todos os bits recessivos (1).

E. Interframe Spacing

*Data Frame*s ou *Remote Frame*s sempre serão precedidos

por um campo de bits chamado *Interframe Spacing* e este campo pode ter sido precedido de um *frame* de qualquer tipo (*Data Frame*, *Remote Frame*, *Error Frame* ou *Overload Frame*). É composto de duas partes: o *Intermission* e o Bus Idle.

1) Intermission

- Descrição: Durante o *Intermission* nenhum nó poderá transmitir um *Data Frame* ou *Remote Frame*. O único *frame* que pode ser transmitido por outro nó durante a transmissão de um *Intermission* é um *Overload Frame*, então se na transmissão do primeiro ou segundo bit do *Intermission* for lido o valor dominante (0), este bit será interpretado como o primeiro bit do *Overload Flags*. E, por conta da tolerância da duração do bit, caso o terceiro bit (último bit) do *Intermission* seja lido o valor dominante (0), então este bit será reconhecido como o SOF (*Start of Frame*) do próximo *Data Frame* ou *Remote Frame*.
- Tamanho do campo: 3 bits.
- Valores esperados: Todos recessivos (1).

2) Bus Idle

- Descrição: Quando nenhum *frame* está sendo transmitido, ou seja, o barramento está livre para transmissão de novos *Data Frames* ou *Remote Frames*. *Overload Frames* e *Error Frames* não são transmitidos no *Bus Idle*.
- Tamanho do campo: 0 a infinitos bits.
- Valores esperados: Todos recessivos (1). Caso seja recebido um bit dominante (0), então este bit será interpretado como o SOF (*Start of Frame*) do próximo *Data Frame* ou *Remote Frame*.

III. Cálculo do CRC

O cálculo do CRC é realizado sobre os bits recebidos do campo SOF (*Start of Frame*) ao *Data Field*, não contabilizando os bits adicionados pelo *Bit Stuffing*, o que acaba por aumentar a vulnerabilidade a erros [5]. O valor inicial utilizado é zero e o polinômio é X15 + X14 + X10 + X8 + X7 + X4 + X3 + 1. O algoritmo, conforme a especificação [4], funciona da seguinte forma:

```
CRC_RG = 0;                // initialize shift register
REPEAT
    CRCNXT = NXTBIT EXOR CRC_RG(14);
    CRC_RG(14:1) = CRC_RG(13:0); // shift left by
    CRC_RG(0) = 0;         // 1 position
    IF CRCNXT THEN
        CRC_RG(14:0) = CRC_RG(14:0) EXOR (4599hex);
    ENDIF
UNTIL (CRC SEQUENCE starts or there is an ERROR condition)
```

Fig. 1 - Pseudocódigo do cálculo do CRC, obtido em [4].

IV. Tratamento do *Bit Stuffing*

O uso da técnica de *Bit Stuffing* fica ativo em um *Data Frame* ou *Remote Frame* entre os campos SOF (*Start of Frame*) e CRC, não incluindo o CRC Delimiter. Após o envio de 5 bits de mesmo nível lógico é necessário enviar um bit de nível lógico oposto, causando uma transição, necessária para efeitos de sincronização. Este bit adicionado artificialmente deve ser descartado pelo receptor, já que não representa nenhuma parte de nenhum campo. Porém, caso seja detectada uma sequência com 6 bits de mesmo nível lógico, o receptor irá detectar um Stuff Error e um *Error Frame* deverá ser transmitido no bit posterior.

V. Máquina de Estados Finitos

Para representar o funcionamento do processo de decodificação dos *frames*, foi feita uma máquina de estados finitos. Nesta máquina de estados, os estados têm os mesmos nomes dos campos dos frames e estão representando os campos que estão para serem lidos, e quando um novo bit é recebido, acontece uma transição de estado ou permanece no mesmo estado, à depedender de quantos bit são esperados para serem recebidos naquele estado. Por exemplo, o campo CRC de um *Data Frame* possui 15 bits, então o sistema após entrar no estado CRC permanecerá neste estado até que receba os 15 bits ou aconteça um erro.

Algumas nomenclaturas foram adotadas na modelagem realizada. São elas:

- rx_bit representa o último bit recebido no estado atual.
- rx_bit == X representa que a leitura do último bit pode ter um valor qualquer.
- contador_ESTADO representa a quantidade bits recebidos no determinado ESTADO.
- Em todos os estados em que um *Bit Stuffing* pode ocorrer, ou seja, do estado S1 (ID_A) ao S10 (CRC), o bit recebido não é armazenado se for um *Bit Stuffing*, neste caso o próximo será armazenado.
- O CRC é calculado em um módulo separado. Os bits que serão utilizados no cálculo do CRC são os bits recebidos do estado S1 (ID_A) até o estado S9 (Data), exceto os *Bit Stuffing*.

Como pode ser visto na Fig. 2, a máquina de estados possui diversos estados. Por conta de limitações na representação gráfica da máquina de estados neste documento, as ações em cada estado e as regras para as transições e permanência nos estados não estão sendo representadas na Fig. 2, no entanto a imagem em alta definição e com todas as informações pode ser obtida em [6]. Estas informações estão descritas de maneira textual da seguinte forma:

A. *S0 - IDLE*

Condições de entrada:

- Ligar o sistema (Estado inicial da máquina de estados)
- Estado == S16 (*Intermission*) && rx_bit == 1 && contador_intermission == 3

Condições de saída:

- rx_bit == 0, vai para S1 (ID_A).

Ação:

- Esperar por rx_bit == 0

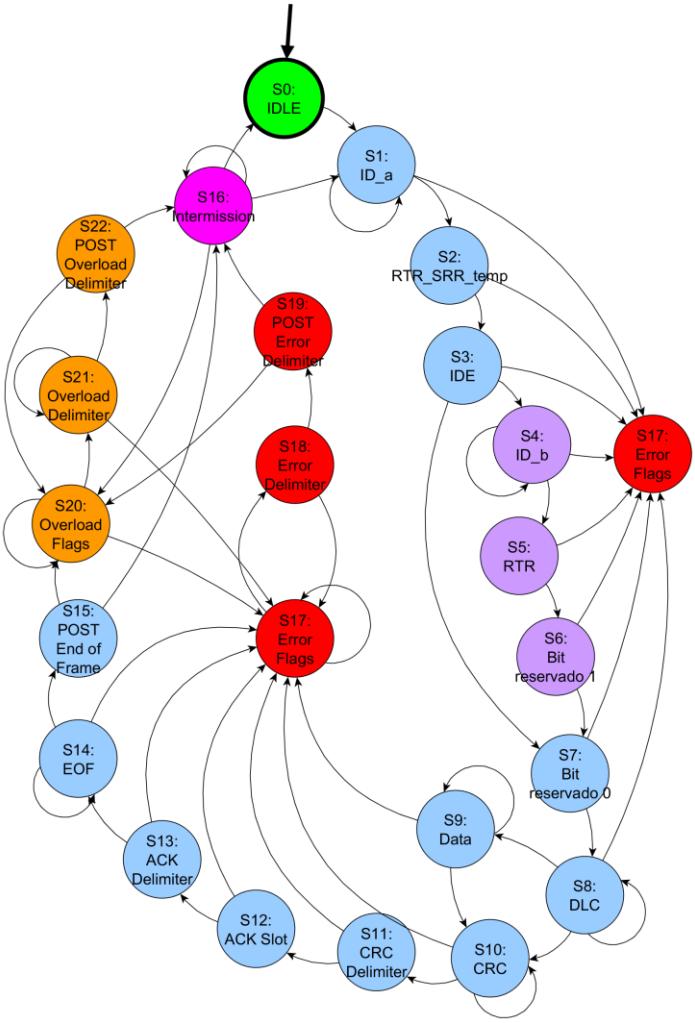

Fig. 2 - Máquina de estados para decodificação do protocolo CAN 2.0.

B. *S1 - ID_A*

Condições de entrada:

- Estado == S0 (Idle) && rx_bit == 0
- Estado == S16 (*Intermission*) && rx_bit == 0 && contador_intermission == 3

Condições de saída:

- contador_id_a == 11 (Leitura de quaisquer 11 bits), vai para S2 (RTR_SRR_temp).
- Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:

- Contador inicia com valor 0 (contador_id_a = 0)
- Habilitar verificação de bit stuffing
- Armazenar o bit
- Incrementar contador_id_a

C. *S2 - RTR_SRR_temp*

Condições de entrada:

- Estado == S1 (ID_A) && contador_ID_A == 11

Condições de saída:

- rx_bit == X, vai para S3(IDE).
- Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:

- Armazenar o bit

D. *S3 - IDE*

Condições de entrada:

- Estado == S2 && rx_bit == X

Condições de saída:

- ide == 0, vai para S7(Bit reservado 0)
- ide == 1, vai para S4(ID_B)
- Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:

- Armazenar o bit

E. *S4 - ID_B*

Condições de entrada:

- Estado == S3 (IDE) && ide == 1

Condições de saída:

- contador_ID_B == 18, vai para S5 (RTR)
- Erro de *Bit Stuffing*, vai para S17 (*Error Flags*).

Ação:

- Contador inicia com valor 0 (contador_id_b = 0)
- SRR = RTR_SRR_temp (ao entrar no estado)
- Armazenar o bit
- Incrementar contador_id_b

F. *S5 - RTR*

Condições de entrada:

- Estado == S4 (ID_B) && contador_ID_B == 18

Condições de saída:

- rx_bit == X, vai para S6(Bit reservado 1).
- Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:

- Armazenar o bit

G. *S6 - Bit reservado 1*

Condições de entrada:

- Estado == S5 (RTR) && rx_bit == X

Condições de saída:

- rx_bit == X, vai para S7(Bit reservado 0)
- Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:

- Armazenar o bit

H. *S7 - Bit reservado 0*

Condições de entrada:

Estado == S6 (Bit reservado 1) && rx_bit == X
Estado == S3 (IDE) && ide == 0

Condições de saída:

rx_bit == X, vai para S8(DLC)
Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:
Armazenar o bit

I. S8 - DLC

Condições de entrada:
- Estado == S7 (Bit reservado 0) && rx_bit == X

Condições de saída:
- contador_dlc == 4 && rtr == 0 && dlc != 0, vai para S9 (Data)
- contador_dlc == 4 && (rtr == 1 || dlc ==0), vai para S10 (CRC)
- Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:
- Contador inicia com valor 0 (contador_dlc = 0)
- Armazenar o bit
- Se DLC > 8, DLC = 8
- Incrementar contador_dlc

J. S9 - Data

Condições de entrada:
- Estado == S8 (DLC) && contador_dlc == 4 && rtr == 0 && dlc != 0

Condições de saída:
- contador_data == 8 * DLC (Adicionar condição na imagem), vai para S10 (CRC)
- Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:
- Contador inicia com valor 0 (contador_data = 0)
- Armazenar o bit
- Incrementar contador_data

K. S10 - CRC

Condições de entrada:
- Estado == S9 (Data) && contador_data == 8 * DLC
- Estado == S8 (DLC) && contador_dlc == 4 && (rtr == 1 || dlc ==0)

Condições de saída:
- contador_crc == 15, vai para S11 (CRC Delimiter)
- Erro de bit stuffing, vai para S17 (*Error Flags*).

Ação:
- Contador inicia com valor 0 (contador_crc = 0)
- Armazenar o bit
- Incrementar contador_crc

L. S11 - CRC Delimiter

Condições de entrada:
- Estado == S10 (CRC) && contador_crc == 15

Condições de saída:
- crc_delimiter == 1, vai para S12 (ACK Slot)
- crc_delimiter == 0, Erro de forma, vai para S17 (*Error Flags*).

Ação:
- Armazenar o bit

M. S12 - ACK Slot

Condições de entrada:
- Estado == S11 (CRC Delimiter) && crc_delimiter == 1

Condições de saída:
- ack_slot == 0, vai para S13 (ACK Delimiter)
- ack_slot == 1, espera-se que o transmissor emita um ACKNOWLEDGMENT ERROR, vai para S17 (*Error Flags*).

Ação:
- Armazenar o bit

N. S13 - ACK Delimiter

Condições de entrada:
- Estado == S12 (ACK Slot) && ack_slot == 0

Condições de saída:
- ack_delimiter == 1, vai para S14 (EOF)
- ack_delimiter == 0 || crc != calculated_crc, Erro de forma ou de CRC, vai para S17 (*Error Flags*).

Ação:
- Armazenar o bit

O. S14 - End of Frame (EOF)

Condições de entrada:
- Estado == S13 (ACK Slot) && ack_delimiter == 1

Condições de saída:
- contador_eof == 7, vai para S15 (POST EOF)
- contador_eof < 7 && rx_bit ==0, Erro de forma, vai para S17 (*Error Flags*).

Ação:
- Contador inicia com valor 0 (contador_eof = 0)
- Armazenar o bit
- Incrementar contador_eof

P. S15 - POST EOF

Condições de entrada:
- Estado == S14 (EOF) && contador_end_of_*frame* == 7 && eof == b111111

Condições de saída:
- rx_bit == 1, vai para S16 (*Intermission*)
- rx_bit == 0, vai para S20 (*Overload Flags*)

Ação:
- Armazenar o bit

Q. S16 - Intermission

Condições de entrada:
- Estado == S15 (POST EOF) && rx_bit == 1
- Estado == S19 (POST *Error Delimiter*) && rx_bit ==1
- Estado == S22 (POST *Overload Delimiter*) && rx_bit ==1

Condições de saída:
- contador_intermission < 3 && rx_bit == 0, vai para S20 (*Overload Flags*)

- contador_intermission == 3 && rx_bit ==0, vai pra S1 (ID_A)
- contador_intermission == 3 && rx_bit ==1, vai pra S0 (IDLE)

Ação:

- Contador inicia com valor 1 (contador_intermission = 1)
- Armazenar o bit
- Incrementar contador_intermission

R. S17 - Error Flags

Condições de entrada:

- Estado == S1 (ID_A) && Erro de bit Stuffing
- Estado == S2 (RTR_SRR) && Erro de bit Stuffing
- Estado == S3 (IDE) && Erro de bit Stuffing
- Estado == S4 (ID_B) && Erro de bit Stuffing
- Estado == S5 (RTR) && Erro de bit Stuffing
- Estado == S6 (Bit Reservado 1) && Erro de bit Stuffing
- Estado == S7 (Bit Reservado 0) && Erro de bit Stuffing
- Estado == S8 (DLC) && Erro de bit Stuffing
- Estado == S9 (Data) && Erro de bit Stuffing
- Estado == S10 (CRC) && Erro de bit Stuffing
- Estado == S11 (CRC Delimiter) && crc_delimiter == 0 (Erro de forma)
- Estado == S12 (ACK Slot) && ack_slot == 1 (Erro de ACK)
- Estado == S13 (ACK Delimiter) && ack_delimiter == 1 (Erro de forma)
- Estado == S13 (ACK Delimiter) && crc != calculated_crc (Erro de CRC)
- Estado == S14 (EOF) && contador_eof < 7 && rx_bit ==0
- Estado == S20 (*Overload Flags*) && contador_flags > 12
- Estado == S20 (*Overload Flags*) && contador_flags < 6 && rx_bit == 1
- Estado == S21 (*Overload Delimiter*) && contador_delimiter < 8 && rx_bit == 0
- Estado == S18 (*Error Delimiter*) && contador_delimiter < 8 && rx_bit == 0

Condições de saída:

- contador_flags >= 6 && contador_flags <12 && rx_bit == 1, vai para S18 (*Error Delimiter*)

Ação:

- Contador inicia com valor 0 (contador_flags = 0)
- Armazenar o bit
- Incrementar contador_flags
- Zera o contador_flags caso detecte um bit de valor 1 quando o contador estiver menor que 6
- Zera o contador_flags caso o contador tenha valor maior que 12

S. S18 - Error Delimiter

Condições de entrada:

- Estado == S17 (*Error Flags*) && contador_flags >= 6 && contador_flags < 12 && rx_bit == 1

Condições de saída:

- contador_delimiter == 8 && delimiter == b1111111, vai para S19 (POST *Error Delimiter*)
- contador_delimiter < 8 && rx_bit == 0, vai para S17 (*Error Flags*)

Ação:

- Contador inicia com valor 0 (contador_delimiter = 0)
- Armazenar o bit
- Incrementar contador_delimiter

T. S19 - POST Error Delimiter

Condições de entrada:

- Estado == S18 (*Error Delimiter*) && contador_delimiter == 8 && delimiter == b1111111

Condições de saída:

- rx_bit == 1, vai para S16 (*Intermission*)
- rx_bit == 0, vai para S20 (*Overload Flags*)

Ação:

- Armazenar o bit

U. S20 - Overload Flags

Condições de entrada:

- Estado == S15 (POST EOF) && rx_bit == 0
- Estado == S16 (*Intermission*) && contador_intermission < 3 && rx_bit == 0
- Estado == S19 (POST *Error Delimiter*) && rx_bit == 0
- Estado == S22 (Last Bit *Overload Delimiter*) && rx_bit == 0

Condições de saída:

- contador_flags >= 6 && contador_flags < 12 && rx_bit == 1, vai para S21 (*Overload Delimiter*)
- contador_flags > 12, vai para S17 (*Error Flags*)
- contador_flags < 6 && rx_bit == 1, vai para S17 (*Error Flags*)

Ação:

- Para as condições de entrada 1,3 e 4 : contador inicia com valor 1 (contador_flags = 1)
- Para a condição de entrada 2 : contador inicia com valor 0 (contador_flags = 0)
- Armazenar o bit
- Incrementar contador_flags

V. S21 - Overload Delimiter

Condições de entrada:

- Estado == S20 (*Overload Flags*) && contador_flags >= 6 && contador_flags < 12 && rx_bit == 1

Condições de saída:

- contador_delimiter == 8 && delimiter == b1111111, vai para S22 (POST *Overload Delimiter*)
- contador_delimiter < 8 && rx_bit == 0, vai para S17 (*Error Flags*)

Ação:

- Armazenar o bit

- Incrementar contador_delimiter

W. *S22 - POST Overload Delimiter*

Condições de entrada:

- Estado == S21 (*Overload Delimiter*) && contador_delimiter == 8 && delimiter == b1111111

Condições de saída:

- rx_bit == 1, vai para S16 (*Intermission*)
- rx_bit == 0, vai para S20 (*Overload Flags*)

Ação:

- Armazenar o bit

VI. Implementação em Verilog de um Decodificador de *Frames* CAN 2.0

A implementação foi dividida em três módulos:

- tester.v - Responsável por gerar os sinais de entrada para teste.
- can_decoder.v - Contém toda a implementação do decoder, incluindo a máquina de estados e o tratamento do bit stuffing.
- can_crc.v - Responsável pelo cálculo do CRC.

A. *Módulo CAN Decoder (can_decoder.v)*

1) *Implementação da Máquina de Estados*

Para implementar a máquina de estados foram utilizados dois blocos always para cada estado, sendo um responsável pelo gerenciamento do estado e outro pelo armazenamento dos dados lidos naquele estado.

Para exemplificação da lógica utilizada na implementação dos estados, as subseções a seguir demonstram a implementação do Estado DLC.

a) Gerenciamento do estado

```
// Estado dlc
always @(posedge clock or posedge reset)
begin
if(reset)
    state_dlc <= 1'b0;
else if(go_state_data | go_state_crc | go_state_error_flags)
    state_dlc <= 1'b0;
else if(go_state_dlc)
    state_dlc <= 1'b1;
end
```

Fig. 3 - Código em Verilog para o tratamento do estado DLC.

Como mostrado no código da Fig. 3, utilizando o estado DLC como exemplo, o sistema trata os estados da seguinte maneira:

- Após um reset, o estado não será o DLC.
- Se forem atingidas as condições para ir para os estados DATA, CRC ou ERROR FLAGS, que são os estados para os quais existe uma transição partindo de DLC, a variável state_dlc terá o valor 0, indicando que o estado atual não é o DLC.
- Se as condições para ir para o estado DLC forem atingidas (o valor de go_state_dlc é 1), então a variável state_dlc terá valor 1, indicando que o estado atual é o DLC.

Em resumo, este bloco always é responsável por gerenciar se o sistema está ou não estado DLC.

b) Armazenamento dos dados lidos no estado

```
// Campo Data Length Count (DLC)
always @ (posedge clock or posedge reset)
begin
  if (reset)
  begin
    field_dlc <= 4'b0;
    contador_dlc <= 3'd0;
  end
  else if (sample_point & state_dlc & (~bit_de_stuffing))
  begin
    /*
    * Se DLC == 4'b1XXX (em que X representa um valor
    * qualquer 0 ou 1), ou seja, se o bit de indice 3
    * for igual a 1, aplica-se a máscara 4'b1000 para
    * garantir ser no máximo 8 o valor do DLC.
    *
    * Nota: field_dlc[2] representará o bit de
    * indice 3 após a execução da linha abaixo.
    */
    field_dlc <= (~field_dlc[2]) ?
        ({field_dlc[2:0], rx_bit}) :
        ({field_dlc[2:0], rx_bit} & 4'b1000);
    contador_dlc <= contador_dlc + 1'b1;
  end
  else if (sample_point & ~state_dlc)
    contador_dlc <= 3'd0;
end
```

Fig. 5 - Código em Verilog para tratamento do valor lido no estado DLC.

Como mostrado no código da Fig. 5, utilizando como exemplo o estado DLC, o sistema tem o seguinte comportamento:

- Após um reset, o contador utilizado no estado DLC e o valor armazenado no campo DLC serão zerados.
- Se houve a leitura de um bit que não deve ser ignorado (*Bit Stuffing*) no estado DLC, então o bit é armazenado no campo DLC e o contador DLC é incrementado.
- Se o valor do campo DLC for maior que 8, então é alterado para 8.
- Se houve a leitura de um bit em um estado que não é o DLC, o contador do DLC é zerado, assim começando com valor zero quando entrar novamente no estado DLC.

2) *Tratamento do Bit Stuffing*

O tratamento do *Bit Stuffing* foi implementado utilizando um bloco *always* (vide Fig. 4) e dois *assign* (vide Fig. 6). O bloco always tem como função armazenar os últimos 5 bits lidos, enquanto os assigns detectam quando existe um erro de *Bit Stuffing* ou quando o bit lido deverá ser ignorado.

```
// Salva os últimos 5 bits lidos para o bit stuffing
always @(posedge clock or posedge reset)
begin
if(reset)
  last_rx_bits <= 5'b10101;
else if (sample_point)
  last_rx_bits <= {last_rx_bits[3:0],rx_bit};
end
```

Fig. 4 - Código em Verilog da lógica de armazenamento dos últimos 5 bits recebidos que serão utilizados na checagem do Bit Stuffing.

Como mostrado na Fig. 4, o sistema faz o tratamento do bit stuffing da seguinte maneira:

- Após um reset, a variável contendo os últimos 5 bits lidos é atribuída a um valor qualquer, que não interfere na corretude da funcionalidade.
- Se houve a leitura de um bit, será realizado um *shift left* na variável contendo os últimos 5 bits e o bit menos significativo será atribuído ao valor do bit lido.

```
assign  bit_de_stuffing =
    sample_point &
    enable_bitstuffing  &
    (((last_rx_bits == 5'h00) & rx_bit) |
    ((last_rx_bits == 5'h1F) & ~rx_bit));

assign  bit_error_bit_stuffing =
    sample_point &
    enable_bitstuffing  &
    (((last_rx_bits == 5'h00) & ~rx_bit) |
    ((last_rx_bits == 5'h1F) & rx_bit));
```

Fig. 6 - Código em Verilog da lógica de checagem do bit stuffing.

Como mostrado na Fig. 6, o sistema faz a checagem do *Bit Stuffing* da seguinte maneira:

- Para detectar que o bit lido deve ser ignorado, os últimos 5 bits lidos devem ser 0's e o bit lido atual deve ser 1; **OU** os últimos 5 bits lidos devem ser 1's e o bit lido atual deve ser 0.
- Para detectar um erro de *Bit Stuffing*, os últimos 5 bits lidos devem ser 0's e o bit lido atual deve ser 0; **OU** os últimos 5 bits lidos devem ser 1's e o bit lido atual deve ser 1.

B. *Módulo CAN CRC (can_crc.v)*

1) *Cálculo do CRC*

```
assign crc_next = data_in ^ crc[14];
assign crc_tmp = {crc[13:0], 1'b0};

always @ (posedge clock)
begin
  if(reset)
    crc <= 15'h0;
  else if (enable)
    begin
      if (crc_next)
        crc <= crc_tmp ^ 15'h4599;
      else
        crc <= crc_tmp;
    end
end
```

Fig. 7 - Código em Verilog do cálculo do CRC.

Para o cálculo do CRC foi implementado em verilog exatamente o mesmo algoritmo presente na especificação do protocolo CAN 2.0 (vide Fig. 7).

VII. TESTES E RESULTADOS

Os casos de teste foram separados em duas partes: Casos simples, para checagem dos frames simples como *Data Frame (Base e Extended), Remote Frame (Base e Extended), Overload Frame e Error Frame* e Casos complexos, que são sequências com dois ou mais *frames*.

Para cada caso de teste é apresentado o resultado da execução do decoder, mostrando os valores lidos, estados, erros e o tipo de frame recebido. O resultado referente aos casos A.1, A.2, A.3, A.4, A.5, A.6.1, A.6.2, A.6.3, B.1, B.2 e B.3 encontram-se nas Fig. 8 a Fig. 17. Nestas figuras cada tipo de informação está sendo motrada com fontes de cores diferentes: os estados estão em verde, os valores dos campos em branco, o tipo do frame recebido em azul e os erros detectados em vermelho.

A. *Casos simples*

1) *Base Data Frame*

Tabela I - Base Data Frame

Campo	Valor em binário (valor em hexadecimal)
SOF	0
ID	11001110010 (0x672)
RTR	0
IDE	0
Reserved	0
DLC	1000 (0x8)
DATA	1010101010101010101010101010101010101010 10101010101010101010101010 (0xAAAAAAAAAAAAAAAA)
CRC	000000001010001 (0x51)
CRC Delimiter	1
ACK Slot	0
ACK Delimiter	1
EOF	1111111
Frame completo com ***Bit Stuffing***	0110011100100001000101010101010101010101 0101010101010101010101010101010101010101 0101010000010000101000011011111111

```
DEBUG: Estado Start of Frame
DEBUG: ============== INICIOU =================
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado IDLE
```

Fig. 8 - Resultado da execução do teste descrito na Tabela I.

2) Base Remote Frame

Tabela II - Base Remote Frame

Campo	Valor em binário (valor em hexadecimal)
SOF	0
ID	11001110010 (0x672)
RTR	1
IDE	0
Reserved	0
DLC	0000 (0x0)
CRC	100000100010000 (0x4110)
CRC Delimiter	1
ACK Slot	0
ACK Delimiter	1
EOF	1111111
Frame completo com ***Bit Stuffing***	01100111001010000**1**0100000**1**10001000010 11111111

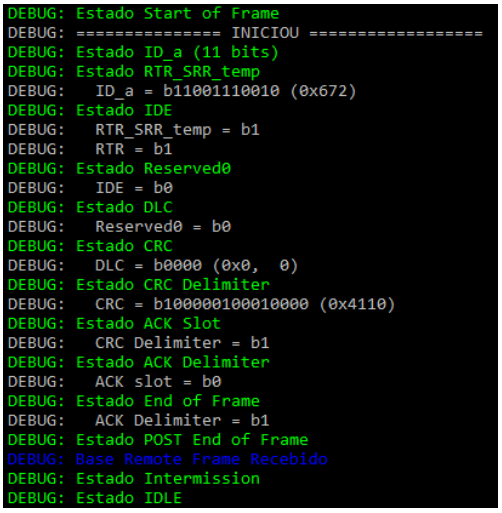

Fig. 9 - Resultado da execução do teste descrito na Tabela II.

3) Extended Data Frame

Tabela III - Extended Data Frame

Campo	Valor em binário (valor em hexadecimal)
SOF	0
ID_A	10001001001 (0x449)
SRR	1
IDE	1
ID_B	110000000001111010 (0x3007A)
RTR	0
Reserved1	0
Reserved0	0
DLC	1000 (0x8)
DATA	1010101010101010101010101010101010 1010101010101010101010101010 (0xAAAAAAAAAAAAAAAA)
CRC	111101111110101 (0x7BF5)
CRC Delimiter	1
ACK Slot	0
ACK Delimiter	1
EOF	1111111
Frame completo com ***Bit Stuffing***	0100010010011111**0**0000**1**00000**1**1111**0**010 0001000101010101010101010101010101 0101010101010101010101010101010101 111011111**0**101011011111111

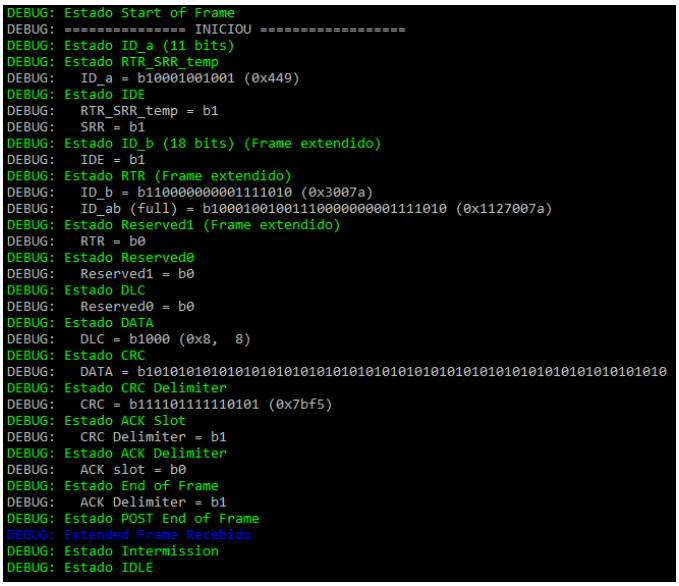

Fig. 10 - Resultado da execução do teste descrito na Tabela III.

4) Extended Remote Frame

Tabela IV - Extended Remote Frame

Campo	Valor em binário (valor em hexadecimal)
SOF	0
ID_A	10001001001 (0x449)
SRR	1
IDE	1
ID_B	110000000001111010 (0x3007A)
RTR	1
Reserved1	0
Reserved0	0
DLC	1000 (0x8)
CRC	010100111110110 (0x29F6)
CRC Delimiter	1
ACK Slot	0
ACK Delimiter	1
EOF	1111111

Frame completo com ***Bit Stuffing***	01000100100111111**0**0000**1**00000**1**1111**0**010 10010000101001111**0**01101011111111

```
DEBUG: Estado Start of Frame
DEBUG: ============== INICIOU =================
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b10001001001 (0x449)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b1
DEBUG:   SRR = b1
DEBUG: Estado ID_b (18 bits) (Frame extendido)
DEBUG:   IDE = b1
DEBUG: Estado RTR (Frame extendido)
DEBUG:   ID_b = b110000000001111010 (0x3007a)
DEBUG:   ID_ab (full) = b10001001001110000000001111010 (0x1127007a)
DEBUG: Estado Reserved1 (Frame extendido)
DEBUG:   RTR = b1
DEBUG: Estado Reserved0
DEBUG:   Reserved1 = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado CRC
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b010100111110110 (0x29f6)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Extended Remote Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado IDLE
```

Fig. 11 - Resultado da execução do teste descrito na Tabela IV.

5) *Overload Frame*

O caso de teste para o *Overload Frame* será em uma situação que pode acontecer na prática. Será enviado um *Data Frame* igual ao utilizado no caso de teste mostrado na Tabela I seguido de um *Overload Frame*, um *Intermission* e o mesmo *frame* enviado inicialmente. O *Overload Frame* enviado contém seis bits dominantes no campo *Overload Flags* e oito bits recessivos no campo *Overload Delimiter*. O *intermission* enviado contém 3 bits recessivos.

Tabela V - Overload Frame.

Campo	Valor em binário (valor em hexadecimal)
Data Frame	Vide Tabela I.
Overload Flags	000000
Overload Delimiter	11111111
Intermission	111
Data Frame	Vide Tabela I.
Frame completo com ***Bit Stuffing***	01100111001000010001010101010101010 10101010101010101010101010101010101 01010101010100000**1**00001010001101111 11110000001111111101100111001000010 00101010101010101010101010101010101 01010101010101010101010101010100000 **1**0000101000110111111111

```
DEBUG: Estado Start of Frame
DEBUG: ============== INICIOU =================
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Overload Flags
DEBUG: Estado Overload Delimiter
DEBUG: Overload Frame Recebido
DEBUG: Estado POST Overload Delimiter
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado IDLE
```

Fig. 12 - Resultado da execução do teste descrito na Tabela V.

6) *Error Frame*

Foram utilizados 3 casos de teste básicos para *Error Frame*. O primeiro simula um Erro de bit Stuffing, ou seja, contém 6 bits iguais e consecutivos. O segundo caso contém um erro no CRC, ou seja, o valor do CRC é diferente do valor esperado. O terceiro caso será enviado um bit recessivo no campo de *Ack Slot*, caracterizando um *ACK Error*. Em todos os casos, o nosso decoder, mesmo sem ter a capacidade de enviar um *frame* de erro no barramento, espera que algum nó envie o erro.

a) Caso 1: Erro de Bit Stuffing

Tabela VI - Caso 1: Erro de Bit Stuffing

Campo	Valor em binário (valor em hexadecimal)
SOF	0
ID	11001110010 (0x672)
RTR	0
IDE	0
Reserved	0
DLC	1000 (0x8)
DATA	10101010101010101010101010101010101010 01010101010101010101010101010 (0xAAAAAAAAAAAAAAAA)
CRC Esperado	000000001010001 (0x51)

Dados Recebidos no campo de CRC	000000 – Detectado Erro de Bit Stuffing
Error Flags	000000
Error Delimiter	11111111
Frame completo com Erro de ***Bit Stuffing***	01100111001000010001010101010101010 10101010101010101010101010101010101 0101010100000000000011111111

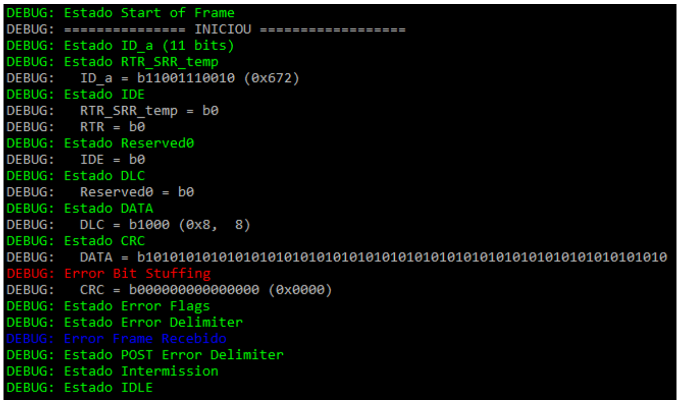

Fig. 13 - Resultado da execução do teste descrito na Tabela VI.

b) Caso 2: Erro de CRC

Tabela VII - Caso 2: Erro de CRC

Campo	Valor em binário (valor em hexadecimal)
SOF	0
ID	11001110010 (0x672)
RTR	0
IDE	0
Reserved	0
DLC	1000 (0x8)
DATA	10101010101010101010101010101010101 01010101010101010101010101010 (0xAAAAAAAAAAAAAAAA)
CRC Esperado	000000001010001 (0x51)
CRC Recebido	000000001110001 (0x71)
CRC Delimiter	1
ACK Slot	0
ACK Delimiter	1
Error Flags	000000
Error Delimiter	11111111
Frame completo com ***Bit Stuffing*** e Erro de CRC	01100111001000010001010101010101010 10101010101010101010101010101010101 0101010100000100001110001101000000111 11111

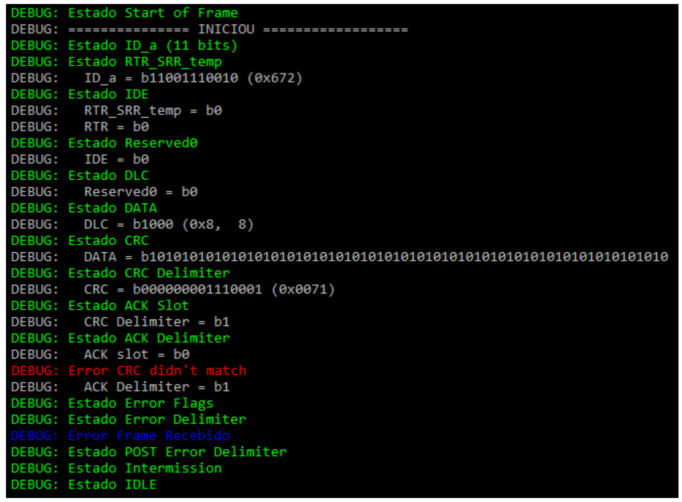

Fig. 14 - Resultado da execução do teste descrito na Tabela VII.

c) Caso 3: Erro de ACK

Tabela VIII - Caso 3: Erro de ACK

Campo	Valor em binário (valor em hexadecimal)
SOF	0
ID	11001110010 (0x672)
RTR	0
IDE	0
Reserved	0
DLC	1000 (0x8)
DATA	10101010101010101010101010101010101 01010101010101010101010101010 (0xAAAAAAAAAAAAAAAA)
CRC	000000001010001 (0x51)
CRC Delimiter	1
ACK Slot	1
Error Flags	000000
Error Delimiter	11111111
Frame completo com ***Bit Stuffing*** e Erro de ACK	01100111001000010001010101010101010 10101010101010101010101010101010101 0101010100000100001010001110000001111 1111

```
DEBUG: Estado Start of Frame
DEBUG: ============== INICIOU ==================
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010 (
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Error ACK Slot = 1
DEBUG:   ACK slot = b1
DEBUG: Estado Error Flags
DEBUG: Estado Error Delimiter
DEBUG: Error Frame Recebido
DEBUG: Estado POST Error Delimiter
DEBUG: Estado Intermission
DEBUG: Estado IDLE
```

d) Fig. 15 - Resultado da execução do teste descrito na Caso 3: Erro de ACK

Tabela VIII.

B. *Casos complexos*

1) Sequência 1

A sequência 1 é a concatenação dos dados de 4 frames iguais ao mostrado na Tabela I. O principal objetivo deste caso de teste foi avaliar a capacidade do decoder receber corretamente uma sequência de frames.

Tabela IX - Sequencia 1: Quatro Data Frames separados por Intermission

Campo	Valor em binário (valor em hexadecimal)
Data Frame	Vide Tabela I.
Intermission	111
Data Frame	Vide Tabela I.
Intermission	111
Data Frame	Vide Tabela I.
Intermission	111
Data Frame	Vide Tabela I.
Frame completo com ***Bit Stuffing***	01100111001000010001010101010101010 10101010101010101010101010101010101 01010101010100000**1**00001010001101111 11111110110011100100001000101010101 01010101010101010101010101010101010 10101010101010101010000**1**0000101000 11011111111111011001110010000100010 10101010101010101010101010101010101 01010101010101010101010100000**1**000 01010001101111111111101100111001000 01000101010101010101010101010101010 10101010101010101010101010101010100 000**1**000010100011011111111

```
DEBUG: Estado Start of Frame
DEBUG: ============== INICIOU ==================
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
```

Fig. 16 - Resultado da execução do teste descrito na Tabela IX, parte 1.

```
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010 (
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado IDLE
```

Fig. 17 - Resultado da execução do teste descrito na Tabela IX, parte 2.

2) Sequência 2

A sequência 2 possui dois frames de dados estendidos, sendo cada um destes precedido por 78 bits recessivos. Os dois frames enviados são iguais e contêm várias sequências de bits de mesmo nível lógico nos campos de ID e Data, assim sendo também um bom caso de teste para o tratamento correto do Bit Stuffing. Neste caso de teste o campo DLC é enviado com valor 15, assim testando o caso em que DLC > 8, no qual o comportamento esperado é considerar o valor 8. Além disso, o CRC do frame utilizado termina com 4 bits 1's, fazendo com que o bit referente ao CRC delimiter, que também deve ter valor 1, seja o quinto bit de mesmo nível lógico seguido, porém o comportamento desejado é que não seja inserido um bit referente ao Bit Stuffing, pois o campo CRC Delimiter não deve sofrer a influência desta técnica.

Tabela X - Extended Data Frame utilizado na sequência 2.

Campo	Valor em binário (valor em hexadecimal)
SOF	0
ID_A	00000000011 (0x3)
SRR	1
IDE	1
ID_B	000000000000000000 (0x0)
RTR	0
Reserved1	0
Reserved0	0
DLC	1111 (0xF)
DATA	111111111111111111111111111111111111 1111111111111111111111111111 (0xFFFFFFFFFFFFFFFF)
CRC	010110000101111 (0x2C2F)
CRC Delimiter	1
ACK Slot	0
ACK Delimiter	1
EOF	1111111
Frame completo com ***Bit Stuffing***	00000**1**00000**1**1111**0**0000**1**00000**1**00000**1**00 000**1**0011111**0**11111**0**11111**0**11111**0**11111**0** 11111**0**11111**0**11111**0**11111**0**11111**0**11111**0** 11111**0**11111**0**111010110000101111101111 1111

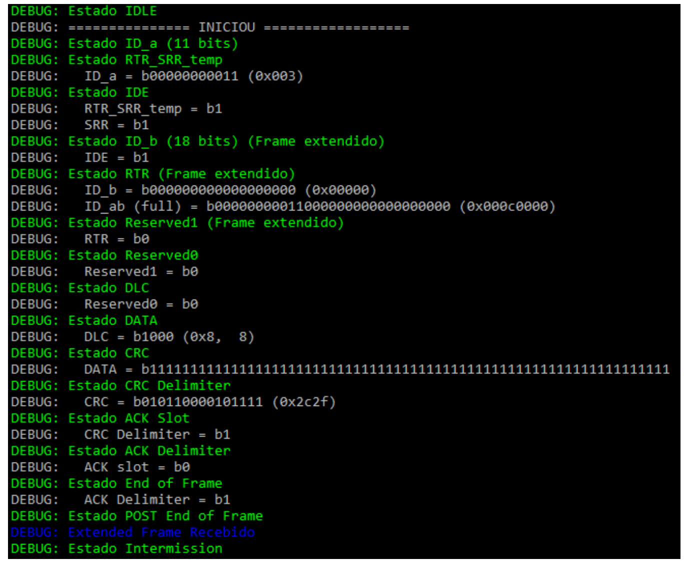

Fig. 18 - Resultado da execução do teste descrito na Tabela X.

Tabela XI - Sequência 2 com diferentes tipos de frame.

Campo	Valor em binário (valor em hexadecimal)
IDLE	78 x 1b
Frame	Vide Tabela X.
Intermission	111
IDLE	75 x 1b
Frame	Vide Tabela X.
Caso de teste completo com ***Bit Stuffing***	11111111111111111111111111111111111 11111111111111111111111111111111111 1111111100000**1**00000**1**1111**0**0000**1**00000 **1**00000**1**00000**1**0011111**0**11111**0**11111**0**11 111**0**11111**0**11111**0**11111**0**11111**0**11111**0**1 1111**0**11111**0**11111**0**11111**0**111010110000 10111110111111111111111111111111111 11111111111111111111111111111111111 11111111111111111111111100000**1**00000 **1**1111**0**0000**1**00000**1**00000**1**00000**1**001111 1**0**11111**0**11111**0**11111**0**11111**0**11111**0**111 11**0**11111**0**11111**0**11111**0**11111**0**11111**0**11 111**0**111010110000101111101111111

```
DEBUG: Estado Intermission
DEBUG: Estado IDLE
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b00000000011 (0x003)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b1
DEBUG:   SRR = b1
DEBUG: Estado ID_b (18 bits) (Frame extendido)
DEBUG:   IDE = b1
DEBUG: Estado RTR (Frame extendido)
DEBUG:   ID_b = b000000000000000000 (0x00000)
DEBUG:   ID_ab (full) = b00000000011000000000000000000 (0x000c0000)
DEBUG: Estado Reserved1 (Frame extendido)
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   Reserved1 = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b11111111111111111111111111111111111111111111111111111111111111111111111111111111
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b010110000101111 (0x2c2f)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Extended Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado IDLE
```

Fig. 19 - Resultado da execução do teste descrito na Tabela XI.

3) *Sequência 3*

A sequência 3 é uma sequência longa contendo 8 *Data Frames*, 1 *Remote Frame*, e 2 *Error Frames*. O objetivo desta sequência de teste foi avaliar a capacidade do decodificador em receber uma grande quantidade de bits e continuar funcionando, mesmo após alguns *Error Frames*. O primeiro frame da sequência é o mesmo representado na Tabela I - Base Data Frame. Os três frames seguintes são semelhantes, porém com campo de dados de tamanhos 5, 3 e 0, respectivamente. O quinto frame é o frame descrito na Tabela III - Extended Data Frame. Em seguida é adicionada a sequência de bits utilizada na Tabela VII - Caso 2: Erro de CRC. Continuando a sequencia, é adicionado um frame igual ao representado na Caso 3: Erro de ACK

Tabela VIII - Caso 3: Erro de ACK, seguido de um frame semelhante ao da Tabela IV - Extended Remote Frame, porém com DLC = 0 e o CRC modificado devido a esta mudança. Por fim é adicionado um frame semelhante ao da Tabela I - Base Data Frame, porém com DLC = 6. É importante notar que foram adicionados os bits referentes ao Intermission sempre que necessário.

Tabela XII - Sequência 3 com diversos tipos de frames.

Campo	Valor em binário com **Bit Stuffing** (valor em hexadecimal)
Frame 1	01100111001000010001010101010101010 10101010101010101010101010101010101 01010101010100000**1**00001010001101111 1111
Intermission	111
Frame 2	01100111001000001**1**01101010101010101 01010101010101010101010110011100 1 001101011111111
Intermission	111
Frame 3	01100111001000001**1**01110101010101010101 01010101001001011100000**1**11011111111
Intermission	111
Frame 4	01100111001000001**1**000011001011010101 1011111111
Intermission	111
Frame 5	01000100100111111**0**0000**1**00000**1**1111**0**01 00001000101010101010101010101010101 01010101010101010101010101010101010 10111101111**0**101011011111111
Intermission	111
Frame 6 (Erro de CRC)	01100111001000010001010101010101010 10101010101010101010101010101010101 01010101010100000**1**00001110001101
Error Frame	00000011111111
Intermission	111
Frame 7 (ACK Error)	01100111001000010001010101010101010 10101010101010101010101010101010101 01010101010100000**1**00001010001110000 0011111111
Intermission	111
Frame 8	01000100100111111**0**0000**1**00000**1**1111**0**01 0100000**1**0101110111111000011011111111
Intermission	111
Frame 9	01100111001000001110101010101010101 01010101010101010101010101010101000 01001001011101011111111
Caso de teste completo	01100111001000010001010101010101010 10101010101010101010101010101010101 01010101010100000**1**00001010001101111 11111110110011100100000**1**10110101010 10101010101010101010101010101010110 01110010011010111111111110110011100 100000**1**01110101010101010101010101010100 10010111000001**1**1011111111111110110011 10010000**1**0000110010110101011011111 11111101000100100111110**0**0000**1**00000**1**1 111**0**0100001000101010101010101010101 01010101010101010101010101010101010 10101010111101111**0**1010110111111111 11011001110010000100010101010101010 10101010101010101010101010101010101 01010101010101000001000011100011010 00000111111111111011001110010000100 0 10101010101010101010101010101010101 01010101010101010101010101000000**1**0 00010100011100000011111111111010001 00100111110**0**0000**1**00000**1**1111**0**01010000 0**1**0101110111110000110111111111111011 00111001000001110101010101010101010 10101010101010101010101010101000010 01001011101011111111

```
DEBUG: Estado Start of Frame
DEBUG: ============== INICIOU =================
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b1010101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b0101 (0x5,  5)
DEBUG: Estado CRC
DEBUG:   DATA = b1010101010101010101010101010101010101010 (0xaaaaaaaaaa)
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b110011100100110 (0x6726)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
```

Fig. 20 - Resultado da execução do teste descrito na Tabela XII, parte 1.

```
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b0011 (0x3,  3)
DEBUG: Estado CRC
DEBUG:   DATA = b101010101010101010101010 (0xaaaaaa)
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b010010111000001 (0x25c1)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado CRC
DEBUG:   DLC = b0000 (0x0,  0)
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b011001011010101 (0x32d5)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
```

Fig. 21 - Resultado da execução do teste descrito na Tabela XII, parte 2.

```
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001001001 (0x449)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b1
DEBUG:   SRR = b1
DEBUG: Estado ID_b (18 bits) (Frame extendido)
DEBUG:   IDE = b1
DEBUG: Estado RTR (Frame extendido)
DEBUG:   ID_b = b110000000001111010 (0x3007a)
DEBUG:   ID_ab (full) = b10001001001110000000001111010 (0x1127007a)
DEBUG: Estado Reserved1 (Frame extendido)
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   Reserved1 = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010101010 (
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b111101111110101 (0x7bf5)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Extended Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010101010 (
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001110001 (0x0071)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Error CRC didn't match
DEBUG:   ACK Delimiter = b1
DEBUG: Estado Error Flags
DEBUG: Estado Error Delimiter
DEBUG: Error Frame Recebido
DEBUG: Estado POST Error Delimiter
DEBUG: Estado Intermission
```

Fig. 22 - Resultado da execução do teste descrito na Tabela XII, parte 3.

```
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b1000 (0x8,  8)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000000001010001 (0x0051)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Error ACK Slot = 1
DEBUG:   ACK slot = b1
DEBUG: Estado Error Flags
DEBUG: Estado Error Delimiter
DEBUG: Error Frame Recebido
DEBUG: Estado POST Error Delimiter
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b10001001001 (0x449)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b1
DEBUG:   SRR = b1
DEBUG: Estado ID_b (18 bits) (Frame extendido)
DEBUG:   IDE = b1
DEBUG: Estado RTR (Frame extendido)
DEBUG:   ID_b = b110000000001111010 (0x3007a)
DEBUG:   ID_ab (full) = b10001001001110000000001111010 (0x1127007a)
DEBUG: Estado Reserved1 (Frame extendido)
DEBUG:   RTR = b1
DEBUG: Estado Reserved0
DEBUG:   Reserved1 = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado CRC
DEBUG:   DLC = b0000 (0x0,  0)
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b101110111110001 (0x5df1)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Extended Remote Frame Recebido
DEBUG: Estado Intermission
```

Fig. 23 - Resultado da execução do teste descrito na Tabela XII, parte 4.

```
DEBUG: Estado Intermission
DEBUG: Estado ID_a (11 bits)
DEBUG: Estado RTR_SRR_temp
DEBUG:   ID_a = b11001110010 (0x672)
DEBUG: Estado IDE
DEBUG:   RTR_SRR_temp = b0
DEBUG:   RTR = b0
DEBUG: Estado Reserved0
DEBUG:   IDE = b0
DEBUG: Estado DLC
DEBUG:   Reserved0 = b0
DEBUG: Estado DATA
DEBUG:   DLC = b0110 (0x6,  6)
DEBUG: Estado CRC
DEBUG:   DATA = b10101010101010101010101010101010101010101010101010101010101010
DEBUG: Estado CRC Delimiter
DEBUG:   CRC = b000100100101110 (0x092e)
DEBUG: Estado ACK Slot
DEBUG:   CRC Delimiter = b1
DEBUG: Estado ACK Delimiter
DEBUG:   ACK slot = b0
DEBUG: Estado End of Frame
DEBUG:   ACK Delimiter = b1
DEBUG: Estado POST End of Frame
DEBUG: Base Frame Recebido
DEBUG: Estado Intermission
DEBUG: Estado IDLE
```

Fig. 24 - Resultado da execução do teste descrito na Tabela XII, parte 5.

VIII. FERRAMENTAS UTILIZADAS

Para desenvolvimento deste projeto foram utilizadas as seguintes ferramentas:

- Altera/Intel Quartus – para compilação do código implementado em Verilog.

- Altera/Intel ModelSim – para execução do código em Verilog e verificação dos casos de teste.
- Notepad++ – como editor de códigos em Verilog, uma vez as funcionalidades providas pelo editor de código do Quartus são bastante limitadas.
- yWorks yEd – para modelagem gráfica da máquina de estados do decodificador de frames.
- Git – como sistema de versionamento.
- Github – como repositório do projeto [7].

IX. Conclusão

Para o desenvolvimento deste projeto foram feitas análises detalhadas sobre o funcionamento do procoloco CAN 2.0 e foram descobertas diversas minúcias sobre a lógica de funcionamento deste protocolo. Esses detalhes são complexos de serem percebidos através da leitura e interpretação da especificação do protocolo em [4], uma vez que a documentação é confusa ou contraditória em alguns pontos e faz-se necessário a utilização de fontes auxiliares sobre o assunto para sanar algumas dúvidas sobre o funcionamento do protocolo.

A máquina de estados finitos modelada e descrita neste trabalho representa detalhadamente o funcionamento do protocolo para decodificação de todos os tipos de frames CAN 2.0.

Os casos de teste utilizados foram criados para testar diferentes combinações de fluxos da máquina de estados através dos diferentes tipos de frames criados. As imagens das impressões no console obtidos através das execuções dos casos de teste no ModelSim demonstram a plena funcionalidade do sistema em conseguir decodificar todos os tipos de frames do protocolo CAN 2.0.

X. Referências

[1] Wikipedia, “CAN bus,” 20 June 2017. [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus. [Acesso em 21 June 2017].

[2] J. A. Cook e J. S. Freudenberg, “Controller Area Network (CAN),” 2008.

[3] A. J. F. F. Vieira, P. J. A. d. Santos, P. M. d. R. Caldeira e R. M. d. S. Fernandes, “PROTOCOLO DE COMUNICAÇÕES CAN,” Porto, 2002.

[4] BOSCH, “CAN Specification - Version 2.0,” 1991.

[5] M. D. Natale, H. Zeng, P. Giusto e A. Ghosal, Understanding and Using the Controller Area Network Communication Protocol, Springer-Verlag New York, 2012.

[6] D. A. Nascimento e L. F. P. D'Andrada, “Máquina de estados do decodificador de frames CAN 2.0: CAN Controller - Automotive Networking Project,” 20 Junho 2017. [Online]. Available: https://raw.githubusercontent.com/davidalain/can-controller/master/Documentação%20do%20projeto/Máquina%20de%20estados/máquina%20de%20estados%20decodificador%20frame%20CAN.png. [Acesso em 21 Junho 2017].

[7] D. A. Nascimento e L. F. P. D'Andrada, “Repositório: CAN Controller - Automotive Networking Project,” 21 June 2017. [Online]. Available: https://github.com/davidalain/can-controller/. [Acesso em 21 June 2017].