

Universidade Federal de Pernambuco

Redes Automotivas IF747

Relatório

Roger Luiz
Wallace Soares

Dezembro - 2017

Conteúdo

1	Introdução	1
2	Diagrama de Blocos	2
3	Maquinas de Estado	4
4	Bit Timing	6
5	Design	8
5.1	BR Frame Maker	8
5.1.1	Estados do CAN BASE	11
5.1.2	Estados do CAN Extended:	13
5.1.3	Estados Finais:	14
5.1.4	As Flags:	15
5.1.5	Tratando Intermission, Error e Overload:	15
5.2	Overload and Error Frame Maker	17
5.3	Type Frame	19
5.4	Stuff	19
5.5	ERROR Block	21
5.5.1	Bit Stuffing Error Block	22
5.5.2	EOF Error Block	22
5.5.3	Form Error Block	23
6	Test Bench e Waveforms	23
7	RTLs	28
8	Referências	30

1 Introdução

Possibilitando o contato com o protocolo CAN através do estudo do formato dos seus frames e entendendo a sincronização entre ECUs, este trabalho tem como objetivo a implementação de um CAN Decoder para que formato, tamanho e tipo de frames sejam identificados. Também será possível identificar alguns erros do tipo passivo, assim como realizar o tratamento das devidas exceções.

Este relatório irá abordar máquinas de estado, diagrama de blocos, waveforms e RTLs. Comparando a fase de prototipação em FSM e Diagrama de Blocos com a implementação de fato.

[illegible]

3

3 Maquinas de Estado

Aqui esta a representação das máquinas de estado do decoder. Todas serão devidamente explanadas na sessão 5.

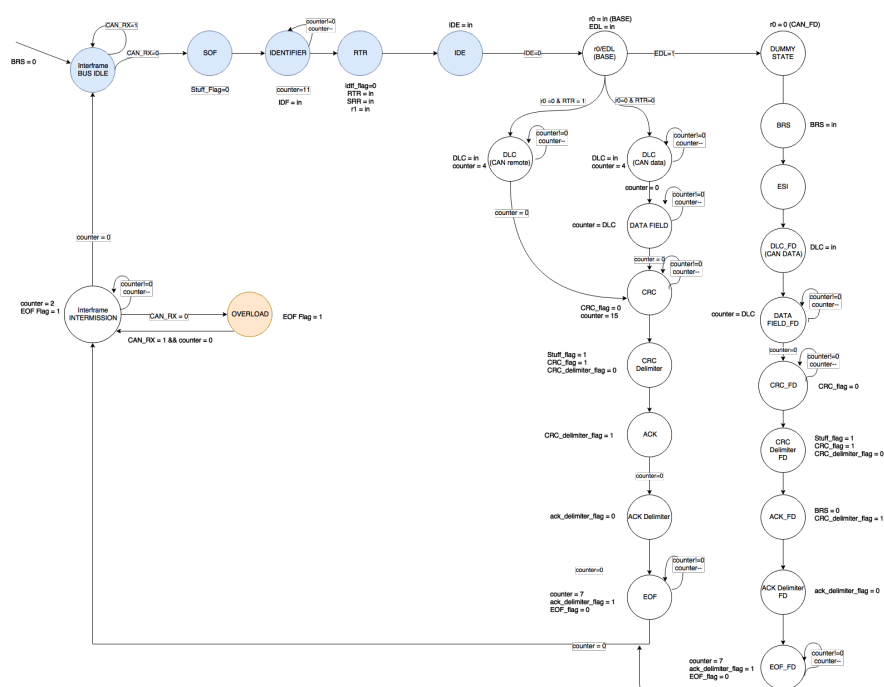


Figura 3: *FSM para o formato normal do CAN*

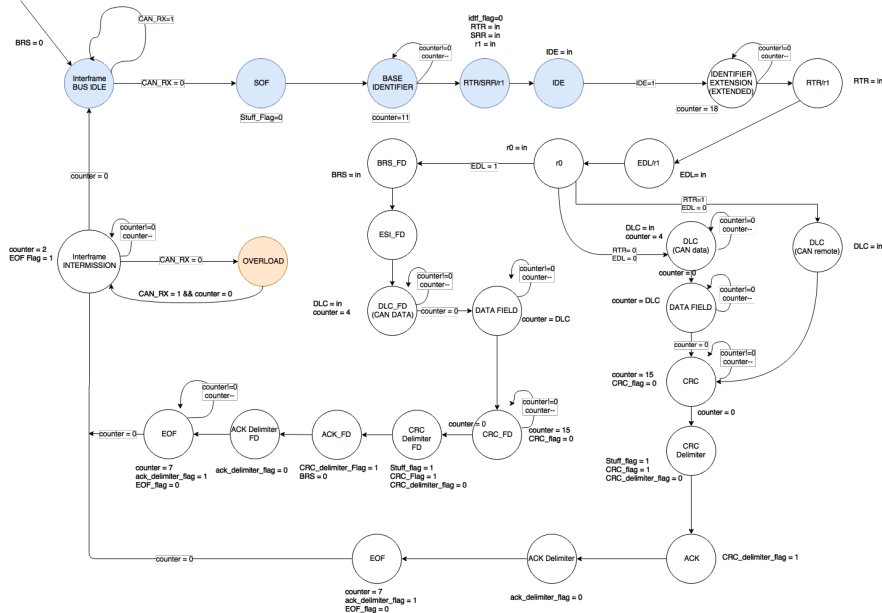


Figura 4: *FSM para o formato estendido do CAN*

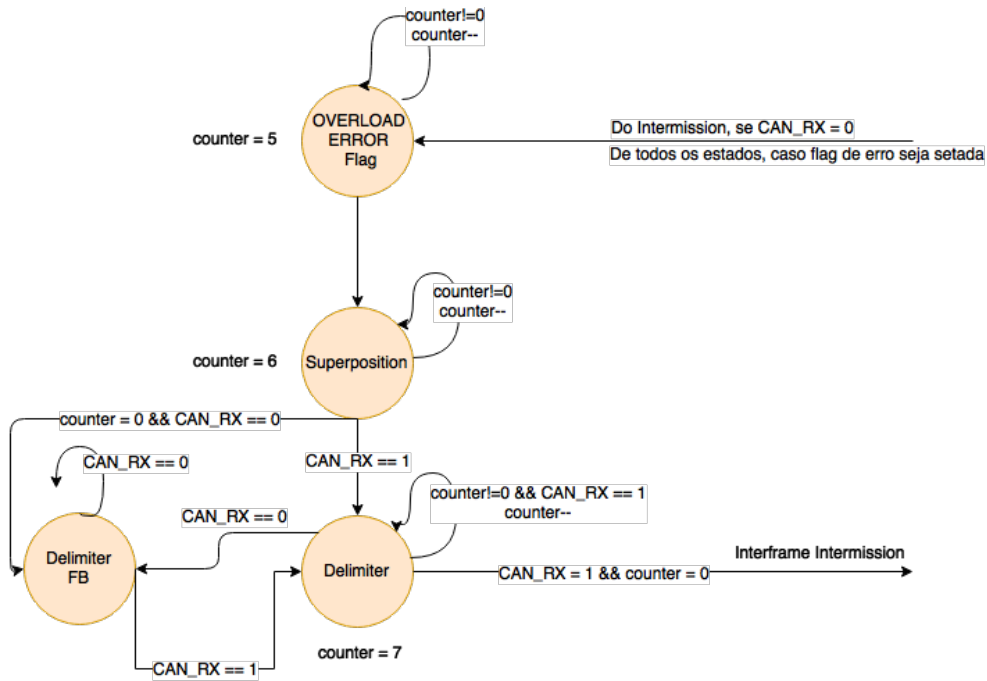


Figura 5: *FSM quando frames de Overloads e Erro precisam ser tratados*

4 Bit Timing

Cada bit no protocolo CAN é dividido em 4 partes: SYNC_SEG, PROP_SEG, PHASE_SEG1, PHASE_SEG2. O ponto onde o valor do bit é lido no barramento é chamado de *sample point*, como indicado na figura abaixo. O ponto do *sample point* é que determina qual o valor exato do bit e caso exista uma algum *shift* na fase e o valor não estiver sendo lido no *sample point*, uma sincronização é necessária.

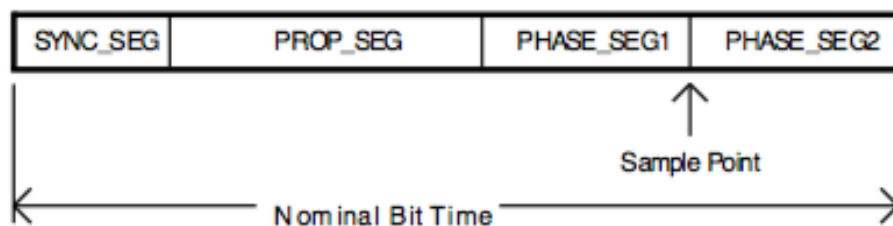


Figura 6: O Bit

A soma dos periodos destes segmentos é igual a um *Nominal Bit Time* e cada um destes segmentos é multiplo de uma unidade de *Time Quantum*, t_Q . A duração de um Time Quantum é igual ao periodo do clock do sistema CAN, que deriva, ou do clock do microcontrolador, ou do oscilador, que é escalonado de acordo com o *Baud Rate Prescaler*, como mostra a figura 7.

A principal função deste processo é garantir a sincronização entre os nós do sistema, pois como o bit do protocolo CAN é NRZ - Non-Return to Zero, isto é, caso venham vários bits dominantes nenhum deles será interrompido por um bit zero(a não ser em caso de stuff) podendo haver perda de sincronização entre os nós e, por consequência, perda de confiabilidade.

Existem duas formas de sincronização: Hard Synchronisation e Re-synchronisation.

Hard Synchronisation funciona como um reset do bit time. Sendo assim ao ser utilizada ela irá forçar o *edge* de variação de bit a acontecer no SYNC_SEG, onde é sempre o ponto ideal de sincronização. Este tipo de sincronização somente acontece quando uma mudança de recessivo para dominante ocorre durante BUS IDLE, ou seja, quando qualquer estação pode enviar uma mensagem para o barramento e ainda não há contenda.

Já o Re-synchronisation ocorre quando uma mudança de bit de dominante para recessivo ocorre fora do SYNC_SEG. Sua diferença com relação ao Hard Synchronisation é que este tipo apenas ajusta a fase do bit de acordo com

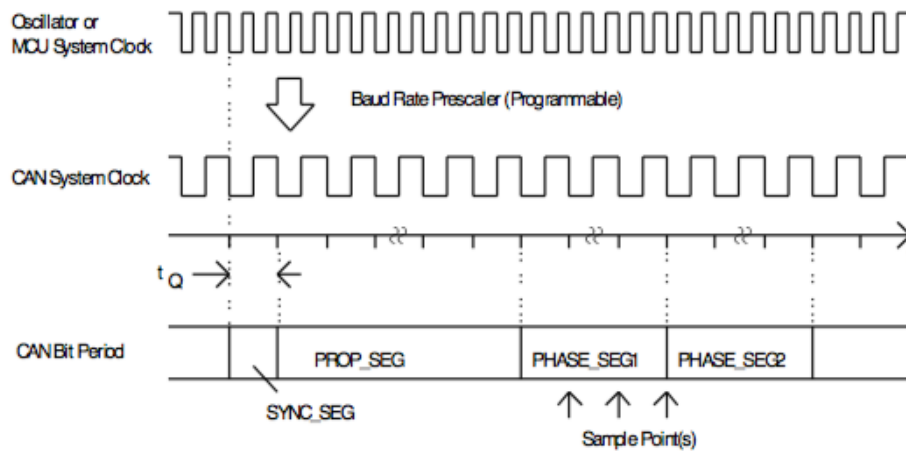


Figura 7: Relação entre o sistema entre o clock do sistema CAN e o período do seu bit

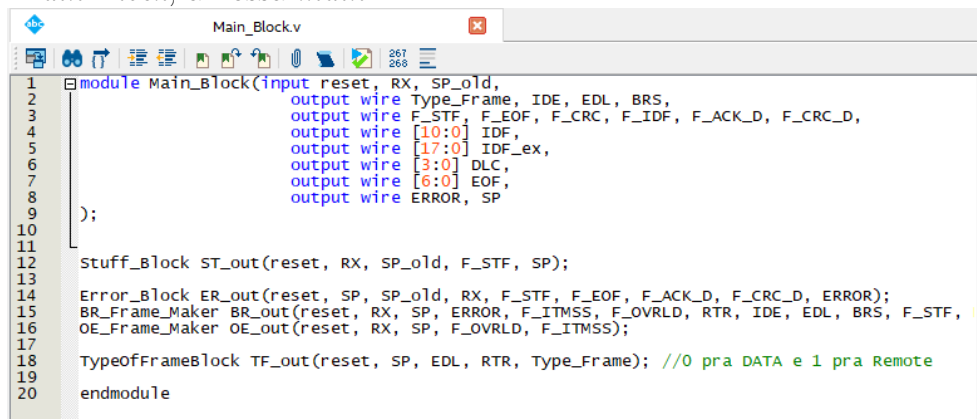
o valor deslocado fora do SYNC_SEG. Se a transição ocorre - de um bit recessivo para dominante - depois do SYNC_SEG e antes do sample-point isso será interpretado como um bit tardio, ou seja, o bit anterior consumiu mais tempo que o esperado do bit atual. O nó tentará sincronizar o bit aumentando a duração do PHASE_SEG1 pela quantidade de Time Quanta que foi consumida do bit atual, contanto que não exceda o re-synchronisation jump width. Assim o próximo *leading edge* acontecerá no SYNC_SEG.

Se a transição ocorrer depois do sample-point, porém antes do SYNC_SEG do próximo bit, isso será interpretado como um bit antecipado, ou seja, o tempo de bit consumiu menos que o esperado. O nó tentará sincronizar o bit diminuindo a duração do PHASE_SEG2 pelo numero de Time Quanta que o bit foi antecipado, isto é, quantos Time Quanta faltavam para o SYNC_SEG do próximo bit. Novamente o numero de Time Quanta que irá limitar essa diminuição é o synchronisation jump width.

Vale salientar que re-synchronisation jump width é programável e não pode exceder 4 Time Quanta, assim como não pode exceder o numero de Time Quanta do PHASE_SEG1, ou seja, é o mínimo entre o PHASE_SEG1 e 4. Seu valor mínimo é 1 Time Quantum.

5 Design

O principal bloco de nosso projeto é que engloba todas as outras entidades é o *Main Block*, a nossa *main*.



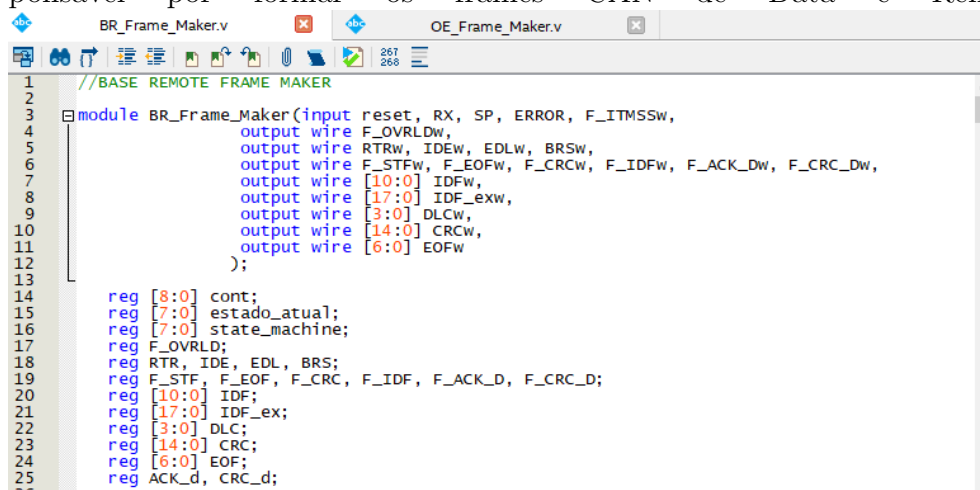
```
1 module Main_Block(input reset, RX, SP_old,
2                   output wire Type_Frame, IDE, EDL, BRS,
3                   output wire F_STF, F_EOF, F_CRC, F_IDF, F_ACK_D, F_CRC_D,
4                   output wire [10:0] IDF,
5                   output wire [17:0] IDF_ex,
6                   output wire [3:0] DLC,
7                   output wire [6:0] EOF,
8                   output wire ERROR, SP
9 );
10
11 stuff_Block ST_out(reset, RX, SP_old, F_STF, SP);
12
13 Error_Block ER_out(reset, SP, SP_old, RX, F_STF, F_EOF, F_ACK_D, F_CRC_D, ERROR);
14 BR_Frame_Maker BR_out(reset, RX, SP, ERROR, F_ITMSS, F_OVRD, RTR, IDE, EDL, BRS, F_STF,
15 OE_Frame_Maker OE_out(reset, RX, SP, F_OVRD, F_ITMSS);
16
17 TypeOfFrameBlock TF_out(reset, SP, EDL, RTR, Type_Frame); //0 pra DATA e 1 pra Remote
18
19 endmodule
```

Como visto no código acima, o *Main Block* envia a sua entrada *SP_old* (*Sample Point*) para o bloco de *stuff* (*Stuff_Block*) e recebe um novo *SP* do bloco de *stuff*. Esse novo *SP* não inclui os edges de bits de *stuff* no momento em que eles estão no *RX*, fazendo com que todas as entidades do projeto as ignorem também.

O bloco de erro (*Error_Block*) é o único que recebe também o *Sample Point* antigo, pois este é necessário para a verificação de erros de *stuff*.

5.1 BR Frame Maker

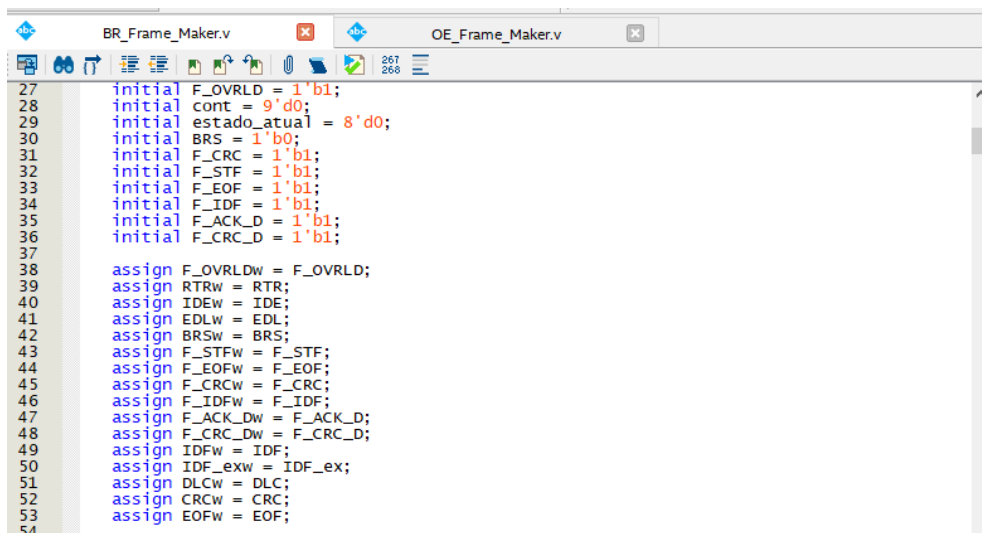
O *BR Frame Maker* é nosso *Frame Maker* principal, responsável por formar os frames *CAN* de *Data* e *Remote*.



```
1 //BASE REMOTE FRAME MAKER
2
3 module BR_Frame_Maker(input reset, RX, SP, ERROR, F_ITMSSw,
4                       output wire F_OVRDw,
5                       output wire RTRw, IDEw, EDLw, BRSw,
6                       output wire F_STFW, F_EOFw, F_CRCw, F_IDFW, F_ACK_Dw, F_CRC_Dw,
7                       output wire [10:0] IDFw,
8                       output wire [17:0] IDF_exw,
9                       output wire [3:0] DLCw,
10                      output wire [14:0] CRCw,
11                      output wire [6:0] EOFw
12 );
13
14 reg [8:0] cont;
15 reg [7:0] estado_atual;
16 reg [7:0] state_machine;
17 reg F_OVRD;
18 reg RTR, IDE, EDL, BRS;
19 reg F_STF, F_EOF, F_CRC, F_IDF, F_ACK_D, F_CRC_D;
20 reg [10:0] IDF;
21 reg [17:0] IDF_ex;
22 reg [3:0] DLC;
23 reg [14:0] CRC;
24 reg [6:0] EOF;
25 reg ACK_d, CRC_d;
26
```

Na imagem acima temos a declaração de registradores, entradas e saídas necessárias. Entre elas:

- **cont:** Contador do bloco.
- **estado_atual:** Guarda o estado da FSM.
- **F_...:** Registradores que começam com F_ são flags usadas a posteriori no bloco.
- **IDF e IDF_ex:** Guarda os identificadores.



```
27 initial F_OVRD = 1'b1;
28 initial cont = 9'd0;
29 initial estado_atual = 8'd0;
30 initial BRS = 1'b0;
31 initial F_CRC = 1'b1;
32 initial F_STF = 1'b1;
33 initial F_EOF = 1'b1;
34 initial F_IDF = 1'b1;
35 initial F_ACK_D = 1'b1;
36 initial F_CRC_D = 1'b1;
37
38 assign F_OVRDw = F_OVRD;
39 assign RTRw = RTR;
40 assign IDEw = IDE;
41 assign EDLw = EDL;
42 assign BRSw = BRS;
43 assign F_STFw = F_STF;
44 assign F_EOFw = F_EOF;
45 assign F_CRCw = F_CRC;
46 assign F_IDFw = F_IDF;
47 assign F_ACK_Dw = F_ACK_D;
48 assign F_CRC_Dw = F_CRC_D;
49 assign IDFW = IDF;
50 assign IDF_exw = IDF_ex;
51 assign DLCw = DLC;
52 assign CRCw = CRC;
53 assign EOFw = EOF;
54
```

Na imagem abaixo, temos o estado “sts1”, o Bus Idle. Neste estado esperamos que o RX seja igual a ‘0’ para mudarmos para o próximo estado. Vale salientar que, se estamos recebendo um bit ‘0’ neste estado, já estamos então no SOF, e iremos agora para o IDENTIFIER.

```

55
56 parameter sts1 = 0, sts2 = 1, sts3 = 2, sts4 = 3, sts5 = 4, sts_B6 = 5, sts_E6 = 6, sts_
57 sts_BCR7 = 8, sts_BCD8 = 9, sts_BF7 = 10, sts_BF8 = 11, sts_BF9 = 12, sts_BF10
58 sts_BF11 = 14, sts_E7 = 15, sts_E8 = 16, sts_E9 = 17, sts_ECD10 = 18, sts_ECR10
59 sts_ECD11 = 20, sts_EF10 = 21, sts_EF11 = 22, sts_EF12 = 23, sts_EF13 = 24, sts
60 sts_II = 26, sts_III = 27, sts_IV = 28, sts_V = 29, sts_INT = 30, sts_OVL = 31;
61
62
63 always @(posedge SP) begin
64
65     cont <= cont + 9'd1;
66
67     case(estado_atual)
68     sts1: begin //BUS IDLE
69         if (RX == 1'b1)begin
70             $display("Estou no Bus Idle");
71         end
72         BRS <= 1'b0;
73         F_EOF <= 1'b1;
74         F_IDF <= 1'b1;
75         F_ACK_D <= 1'b1;
76         F_CRC_D <= 1'b1;
77         if (RX == 1'b0)begin //Já estou no bit de SOF
78             $display("\n\n");
79             $display("Estou no SOF");
80             F_STF <= 1'b0;
81             estado_atual <= sts3;
82             cont <= 9'd0;
83         end
84     end
85     sts2: begin //IDENTIFIER

```

No IDENTIFIER vamos transferindo bit a bit os valores assumidos por RX para o registrador IDF.

No RTR, estamos lendo o bit RTR do CAN Base Format, r1 do CAN FD Base Format e SRR do CAN Extended. Na implementação do projeto escolhemos colocar todos esses dados em uma variável chamada RTR. Isto foi feito pensando da seguinte forma:

1. O RTR indica se o frame é de dados ou remoto (0 para DATA e 1 para REMOTE). O r1 é obrigatoriamente '0', o que indica a inexistência de frames remotos no CAN FD. Veja só, o r1 funciona como um RTR do FD, onde o bit estaria sempre indicando um frame de Data.
2. O SRR não tem uso neste projeto, então ele é guardado na variável RTR até que se perceba o tipo do cabeçalho do Frame (Can Base ou Extended). Se o cabeçalho for estendido, não usamos o conteúdo da variável RTR até reescrevermos ele com o valor correto do RTR, em um estado posterior da FSM.

Deve ficar claro que o r1 e o SRR não são de forma alguma o RTR do frame (CAN FD não possui RTR), mas apenas estão usando uma mesma variável chamada RTR, por conveniência. O uso de variáveis diferentes, mais apropriadas e que evitem confusão, é recomendado na maioria dos casos. No IDE, definimos o tipo de cabeçalho, se Base ou Extended, mudando para dois diferentes estados, dependendo do valor lido, como mostrado abaixo:

```

85 sts3: begin //IDENTIFIER
86   IDF [10-cont] <= RX;
87   $display("Estou no Identifier %d", cont+1);
88   // $display("contador: %d", cont);
89   if (cont >= 9'd10) begin
90     // $display("contador: %d", cont);
91     estado_atual <= sts4;
92     cont <= 9'd0;
93   end
94 end
95 sts4: begin //RTR
96   $display("Estou no RTR");
97   estado_atual <= sts5;
98   cont <= 9'd0;
99   RTR <= RX;
100 end
101 sts5: begin //IDE
102   IDE <= RX;
103   $display("Estou no IDE");
104   if (RX == 1'b0) begin
105     estado_atual <= sts_B6;
106   end
107   if (RX == 1'b1) begin
108     estado_atual <= sts_E6;
109   end
110   cont <= 9'd0;
111 end

```

Agora a máquina de estados se divide. Estados começados por “sts_B” são para o CAN Base, estados começados por “sts_E” são para o CAN Extended.

Para o can BASE temos o estado de EDL, que junto com o valor do RTR pego anteriormente, define se o frame é CAN BASE FD, CAN BASE Remote ou CAN Base DATA. O EDL diferencia entre CAN e CAN FD. O RTR diferencia entre o Data e Remote. É importante lembrar que não existe EDL no CAN BASE Format (apenas no FD), mas usamos o nome EDL na variável por conveniência.

Para o CAN FD temos o estado que lê a segunda parte do identificador.

```

109 end
110   cont <= 9'd0;
111 end
112 sts_B6: begin //EDL Base
113   EDL <= RX;
114   F_IDF <= 1'b0;
115   $display("Estou no EDL Base");
116   if (RX == 1'b0 && RTR == 1'b0) begin
117     estado_atual <= sts_BCD7;
118   end
119   if (RX == 1'b0 && RTR == 1'b1) begin
120     estado_atual <= sts_BCR7;
121   end
122   if (RX == 1'b1) begin
123     estado_atual <= sts_BF7;
124   end
125   cont <= 9'd0;
126 end
127 sts_E6: begin //IDENTIFIER EXTENSION EXTENDED
128   $display("Estou no Idf extended %d", cont+1);
129   IDF_ex [17-cont] <= RX;
130   if (cont >= 9'd17) begin
131     estado_atual <= sts_E7;
132     cont <= 9'd0;
133   end
134 end
135

```

5.1.1 Estados do CAN BASE

Os estados do CAN base seguem em frente, lendo o DLC. Ao fim do DLC a flag de stuff (F_STF) é desativada e a flag de CRC ativada.

```

BR_Frame_Maker.v  OE_Frame_Maker.v
267 268

135 //-----Estados de CAN BASE (BC)-----//
136
137
138
139 sts_BCD7: begin //DLC CAN Data
140   F_IDF <= 1'b1;
141   $display("Estou no DLC do CAN Base Data %d", cont+1);
142   // $display("Contador: %d", cont);
143   DLC [3-cont] <= RX;
144   if (cont >= 9'd3) begin
145     // $display("Vou mudar pro Data Field");
146     // $display("Contador: %d", cont);
147     estado_atual <= sts_BCD8;
148     cont <= 9'd0;
149   end
150 end
151 sts_BCR7: begin //DLC CAN Remote
152   F_IDF <= 1'b1;
153   $display("Estou no DLC do CAN Base Remote %d", cont+1);
154   // $display("Contador: %d", cont);
155   DLC [3-cont] <= RX;
156   if (cont >= 9'd3) begin
157     estado_atual <= sts_I;
158     cont <= 9'd0;
159     F_CRC <= 1'b0;
160     F_STF <= 1'b1;
161   end
end

```

Após isso é lido o DATA Field.

No CAN Base FD passamos por um Dummy State (onde o r0 está sendo transmitido, porém não é útil para nosso projeto). O BRS é lido e o ESI é ignorado, pois também não é útil para nosso projeto.

```

BR_Frame_Maker.v  OE_Frame_Maker.v
267 268

162 sts_BCD8: begin //DATA FIELD
163   $display("Estou no Data Field %d", cont+1);
164   if (cont >= (DLC*8)-1) begin //Verificar esta comparação
165     estado_atual <= sts_I;
166     cont <= 9'd0;
167     F_CRC <= 1'b0;
168     F_STF <= 1'b1;
169   end
170 end
171
172 //-----Estados de CAN BASE FD (BF)-----//
173
174
175 sts_BF7: begin //Dummy STATE
176   F_IDF <= 1'b1;
177   $display("Estou no Dummy State do Can Base FD");
178   estado_atual <= sts_BF8;
179   cont <= 9'd0;
180 end
181 sts_BF8: begin //BRS
182   BRS <= RX;
183   $display("Estou no BRS");
184   estado_atual <= sts_BF9;
185   cont <= 9'd0;
186 end
187 sts_BF9: begin //ESI
188   $display("Estou no ESI");
189   estado_atual <= sts_BF10;
190   cont <= 9'd0;
191 end

```

O DLC do FD é lido. Ao fim do DLC a flag de stuff (F_STF) é desativada e a flag de CRC ativada. Após isso é lido o DATA Field.

```

BR_Frame_Maker.v  OE_Frame_Maker.v
192 sts_BF10: begin //DLC_FD
193   $display("Estou no DLC do CAN Base FD %d", cont+1);
194   DLC [3-cont] <= RX;
195   if (cont >= 9'd3)begin
196     estado_atual <= sts_BF11;
197     cont <= 9'd0;
198   end
199 end
200 sts_BF11: begin //DATA_FIELD_FD
201   $display("Estou no Data Field %d", cont+1);
202   if (cont >= (DLC*8)-1)begin //Verificar esta comparação
203     estado_atual <= sts_I;
204     cont <= 9'd0;
205     F_CRC <= 1'b0;
206     F_STF <= 1'b1;
207   end
208 end
209
210 //-----Estados de CAN EXTENDED (E)-----//
211
212 sts_E7: begin //RTR EXTENDED
213   RTR <= RX;
214   F_IDF <= 1'b0;
215   $display("Estou no RTR do CAN Extended");
216   $display("RX: %d", RX);
217   estado_atual <= sts_E8;
218   cont <= 9'd0;
219 end

```

5.1.2 Estados do CAN Extended:

Os estados do CAN Extended passam pelo RTR, onde o valor correto é sobrescrito no registrador RTR (que até então continha o valor do SRR).

No estado EDL, apenas guardamos o valor de RX no registrador EDL. É importante lembrar que não existe EDL no CAN Extended Format (apenas no FD), mas usamos o nome EDL na variável por conveniência!

No estado r0, usamos os valores de RTR e EDL pegos anteriormente e definimos se o frame é CAN Extended FD, CAN Extended Remote ou CAN Extended DATA. O EDL diferencia entre CAN e CAN FD. O RTR diferencia entre o Data e Remote.

```

BR_Frame_Maker.v  OE_Frame_Maker.v
218 cont <= 9'd0;
219 end
220 sts_E8: begin //EDL
221   EDL <= RX;
222   F_IDF <= 1'b1;
223   $display("Estou no EDL");
224   $display("RX: %d", RX);
225   estado_atual <= sts_E9;
226   cont <= 9'd0;
227 end
228 sts_E9: begin //r0
229   $display("Estou no r0");
230   $display("EDL: %d", EDL);
231   if (EDL == 1'b1)begin
232     estado_atual <= sts_EF10;
233   end
234   if (EDL == 1'b0 && RTR == 1'b0)begin
235     estado_atual <= sts_ECD10;
236   end
237   if (EDL == 1'b0 && RTR == 1'b1)begin
238     estado_atual <= sts_ECR10;
239   end
240   cont <= 9'd0;
241 end

```

O CAN Extended comum segue pelos estados de DLC e DATA Field. Ao fim do DLC a flag de stuff (F_STF) é desativada e a flag de CRC ativada.

```

242 //-----Estados de CAN EXTENDED COMUM (EC)-----//
243
244 sts_ECD10: begin //DLC EXTENDED DATA
245     $display("Estou no DLC do CAN Extended DATA %d", cont+1);
246     DLC [3-cont] <= RX;
247     if (cont >= 9'd3)begin
248         estado_atual <= sts_ECD11;
249         cont <= 9'd0;
250     end
251 end
252
253 sts_ECR10: begin //DLC EXTENDED REMOTE
254     $display("Estou no DLC do CAN Extended Remote %d", cont+1);
255     DLC [3-cont] <= RX;
256     if (cont >= 9'd3)begin
257         estado_atual <= sts_I;
258         cont <= 9'd0;
259         F_CRC <= 1'b0;
260         F_STF <= 1'b1;
261     end
262 end
263
264 sts_ECD11: begin //DATA FIELD
265     $display("Estou no Data Field %d", cont+1);
266     if (cont >= (DLC*8)-1)begin //Verificar esta comparação
267         estado_atual <= sts_I;
268         cont <= 9'd0;
269         F_CRC <= 1'b0;
270         F_STF <= 1'b1;
271     end
272 end

```

No CAN Extended FD, lemos o BRS, ignoramos o ESI e em seguida lemos o DLC e Data Field. Ao fim do DLC a flag de stuff (F_STF) é desativada e a flag de CRC ativada.

```

273 //-----Estados de CAN EXTENDED FD (EF)-----//
274
275 sts_EF10: begin //BRS FD
276     BRS <= RX;
277     $display("Estou no BRS do CAN Extended FD");
278     estado_atual <= sts_EF11;
279     cont <= 9'd0;
280 end
281
282 sts_EF11: begin //ESI_FD
283     $display("Estou no ESI do CAN Extended FD");
284     estado_atual <= sts_EF12;
285     cont <= 9'd0;
286 end
287
288 sts_EF12: begin //DLC_FD
289     $display("Estou no DLC do CAN Extended FD %d", cont+1);
290     DLC [3-cont] <= RX;
291     if (cont >= 9'd3)begin
292         estado_atual <= sts_EF13;
293         cont <= 9'd0;
294     end
295 end
296
297 sts_EF13: begin //DATA_FIELD_FD
298     $display("Estou no Data Field %d", cont+1);
299     if (cont >= (DLC*8)-1)begin
300         estado_atual <= sts_I;
301         cont <= 9'd0;
302         F_CRC <= 1'b0;
303         F_STF <= 1'b1;
304     end
305 end

```

5.1.3 Estados Finais:

Após estes estados, todos os frames convergem para os estados seguintes, que encerram os frames. São eles:

No “sts.I”, temos o CRC.

No “sts.II” temos o CRC Delimiter, onde o BRS volta pra ‘0’, independente do valor lido anteriormente, pois neste momento os frames CAN e

CAN FD devem voltar ao mesmo datarate. No “sts_III” temos o ACK e no “sts_IV” o ACK Delimiter. Por fim temos o estado “sts_V”, o EOF.

```

303 //-----Tratando estados finais-----//
304
305 sts_I: begin //CRC
306     $display("Estou no CRC %d", cont+1);
307     $display("Contador: %d", cont);
308     F_CRC_D <= 1'b1;
309     if (cont >= 9'd14)begin
310         estado_atual <= sts_II;
311         cont <= 9'd0;
312         F_CRC_D <= 1'b0;
313     end
314 end
315
316 sts_II: begin //CRC Delimiter
317     F_CRC_D <= 1'b1;
318     BRS <= 1'b0;
319     CRC_d <= RX;
320     $display("Estou no CRC Delimiter");
321     estado_atual <= sts_III;
322     cont <= 9'd0;
323 end
324
325 sts_III: begin //ACK
326     F_ACK_D <= 1'b0;
327     $display("Estou no ACK");
328     estado_atual <= sts_IV;
329     cont <= 9'd0;
330 end
331
332 sts_IV: begin //ACK Delimiter
333     F_ACK_D <= 1'b1;
334     F_EOF <= 1'b0;
335     ACK_d <= RX;
336     $display("Estou no ACK Delimiter");
337     estado_atual <= sts_V;
338     cont <= 9'd0;
339 end
340
341 sts_V: begin //EOF
342     F_EOF <= 1'b1;
343     $display("Estou no EOF %d", cont+1);
344     EOF[6-cont] <= RX;
345     if (cont >= 9'd6)begin
346         estado_atual <= sts_INT;
347         cont <= 9'd0;
348     end
349 end

```

5.1.4 As Flags:

As flags F_CRC, F_EOF, F_ACK_D, F_CRC_D apenas avisam aos outros blocos do nosso projeto que o próximo estado é o estado que aquela flag representa, ou seja, estado de CRC, End of Frame, ACK Delimiter e CRC Delimiter, respectivamente. Estas flags são de extrema importância no nosso decodificador CAN.

5.1.5 Tratando Intermisson, Error e Overload:

Após o EOF, a máquina se dirige ao estado de INTERMISSION, onde uma série de comparações é feita. Primeiro, devemos verificar o valor do RX para sabermos se realmente estamos no Intermisson, pois caso este valor seja igual a '0', estamos na verdade em um Overload. Se o RX for igual a '0' em qualquer dos dois bits de Intermisson, mudamos para o estado de Overload e

ativamos uma flag (F_OVRD) que avisa ao OE Frame Maker que este deve começar a ler um bit de Overload. Se não temos bits '0', seguimos novamente (após o segundo bit) para o BUS Idle, reiniciando a FSM.

No estado de Overload, esperamos por um aviso do bloco OE Frame Maker por meio de uma flag (F_ITMSS), de que devemos voltar a execução normal, ou seja, que já estamos no primeiro bit de Intermission. Por já estarmos no Intermission, temos que verificar se o RX é igual a '0', retornando para o atual estado de Overload e ativando novamente a flag (F_OVRD), caso esta situação seja verdade. Se não temos bits '0', seguimos para o estado de Intermission com valor de contador já setado em 1.

```

347 //-----Tratando INTERMISSION, ERROR e OVERLOAD-----//
348
349 sts_INT:begin //INTERMISSION
350 if (RX == 1'b1)begin
351 $display("Estou no Intermission");
352 end
353 if (RX == 1'b0)begin //Nesse caso, já estamos no primeiro
354 estado_atual <= sts_OVL; //bit de Overload
355 cont <= 9'd0;
356 F_OVRD <= 1'b0;
357 $display("\n\n");
358 $display("Estou no Primeiro Bit de Overload");
359 end
360 if (RX == 1'b1 && cont == 9'd1)begin //Vou para o BUS IDLE
361 estado_atual <= sts1;
362 cont <= 9'd0;
363 end
364 end
365 sts_OVL:begin //OVERLOAD AND ERROR
366 F_OVRD <= 1'b1;
367 F_CRC <= 1'b1;
368 F_STF <= 1'b1;
369 BRS <= 1'b0;
370 F_EOF <= 1'b1;
371 F_IDF <= 1'b1;
372 F_ACK_D <= 1'b1;
373 F_CRC_D <= 1'b1;
374 if (F_ITMSS == 1'b0)begin //Primeiro bit do INTERMISSON
375 if (RX == 1'b1)begin
376 end
377 if (RX == 1'b0)begin //Nesse caso, já estamos no primeiro
378 estado_atual <= sts_OVL; //bit de Overload
379 cont <= 9'd0;
380 F_OVRD <= 1'b0;
381 end
382 else begin
383 estado_atual <= sts_INT; //INTERMISSION
384 cont <= 9'd1;
385 end
386 end
387 end
388 end

```

A qualquer momento, a entrada ERROR do bloco pode ser ativada, nos encaminhando diretamente para o estado de Overload e ativando a flag de overload. Isso acontece após um aviso do bloco de erro, que tem ERROR como saída. Um frame de erro começa a ser lido no bloco OE Frame Maker.

```

if(reset == 1)begin
    $display("\n\n");
    $display("Estou resetando");
    cont <= 9'd0;
    estado_atual <= sts1;
    F_OVRD <= 1'b1;
    F_STF <= 1'b1;
    F_CRC <= 1'b1;
    BRS <= 1'b0;
    F_EOF <= 1'b1;
    F_IDF <= 1'b1;
    F_ACK_D <= 1'b1;
    F_CRC_D <= 1'b1;
    IDF <= 11'd0;
    IDF_ex <= 18'd0;
    DLC <= 4'd0;
    EOF <= 7'd0;
end
else begin
    if(ERROR == 0 && estado_atual != sts_OVL)begin //Estou no primeiro bit de ERRO
        cont <= 9'd0;
        estado_atual <= sts_OVL;
        F_OVRD <= 1'b0;
        F_STF <= 1'b1;
        F_CRC <= 1'b1;
        BRS <= 1'b0;
        F_EOF <= 1'b1;
        F_IDF <= 1'b1;
        F_ACK_D <= 1'b1;
        F_CRC_D <= 1'b1;
    end
end
end
endmodule

```

5.2 Overload and Error Frame Maker

O Overload Frame Maker é o bloco responsável por ler e formar os frames de erro e overload. Começamos por um estado “sts1”, que chamamos de “Waiting for overload or error”. Este estado aguarda que a flag de overload (F_OVRD) recebida do BR Frame Maker seja ativada, indicando que um frame de erro ou overload deve começar a ser lido. Sendo este caso, ocorre uma mudança de estado para o “sts2”.

```

1  //OVERLOAD ERROR FRAME MAKER
2
3  module OE_Frame_Maker(input reset, RX, SP, F_OVRDLW,
4                        output wire F_ITMSSW
5                        );
6
7      reg [8:0] cont;
8      reg [7:0] estado_atual;
9      reg F_ITMSS;
10
11      parameter sts1 = 0, sts2 = 1, sts3 = 2, sts4 = 3, sts5 = 4;
12
13      initial cont = 9'd0;
14      initial estado_atual = 8'd0;
15      initial F_ITMSS = 1'b1;
16
17      assign F_ITMSSW = F_ITMSS;
18
19      always @ (posedge SP) begin
20          if(reset == 1)begin
21              cont <= 9'd0;
22              estado_atual <= sts1;
23              F_ITMSS <= 1'b1;
24          end
25          else begin
26              cont <= cont + 9'd1;
27          end
28
29          case(estado_atual)
30              sts1: begin //WAITING FOR OVERLOAD OR ERROR
31                  F_ITMSS <= 1'b1;
32                  if(F_OVRDLW == 1'b1)begin
33                      //display("Waiting for Error");
34                  end
35                  if(F_OVRDLW == 1'b0)begin //Já estamos no segundo bit de overload
36                      $display("Estou no Bit de Flag 2");
37                      cont <= 9'd0;
38                      estado_atual <= sts2;
39                  end
40              end

```

No estado “sts2” (figura abaixo), o “Overload error flag third bit”, consumimos os bits de flag do frame e seguimos em diante para o estado “sts3”. É possível notar que o estado começa no terceiro bit de flag, pois o primeiro bit foi consumido ainda no BR Frame Maker, e o segundo bit foi consumido no estado “Waiting for overload or error”.

No estado “sts3” temos o superposition. O superposition é composto de apenas bits 0; caso um bit 1 surja, estamos então no estado de Delimiter. O estado atual muda então para o estado “sts5”.

O estado “sts5” é o estado de delimiter. Como o primeiro bit de delimiter é lido ainda no estado de superposition, começamos a ler nesse estado o segundo bit de delimiter. O delimiter é composto de 8 bits 1. Caso um bit 0 seja lido, é preciso resetar o delimiter.

O estado “sts4” é um estado tampão, pois ao ser resetado, é preciso que o delimiter comece do primeiro bit (o estado delimiter lê a partir do segundo bit de delimiter). Este estado é chamado então de Delimiter first bit.

Ao fim da execução, a flag de intermission (F_ITMSS) é setada, avisando ao BR Frame Maker que deve voltar a execução normal.

```

41 sts2: begin //OVERLOAD_ERROR_Flag THIRD Bit
42 $display("Estou no Bit de Flag %d", cont+3);
43 if(cont >= 9'd3)begin
44 cont <= 9'd0;
45 estado_atual <= sts3;
46 end
47 end
48 sts3: begin //Superposition
49 if(RX == 1'b0)begin
50 $display("Estou no Superposition %d", cont+1);
51 end
52 if(cont >= 9'd5 && RX == 1'b0)begin
53 cont <= 9'd0;
54 estado_atual <= sts4;
55 end
56 if(RX == 1'b1)begin
57 $display("Estou no Delimiter 1");
58 cont <= 9'd0;
59 estado_atual <= sts5;
60 end
61 end
62 sts4: begin //Delimiter First Bit
63 $display("Estou no Delimiter 1");
64 if(RX == 1'b1)begin
65 cont <= 9'd0;
66 estado_atual <= sts5;
67 end
68 end
69 sts5: begin //Delimiter
70 $display("Estou no Delimiter %d", cont+2);
71 if(RX == 1'b0)begin
72 $display("Vou resetar o Delimiter");
73 cont <= 9'd0;
74 estado_atual <= sts4;
75 end
76 if(RX == 1'b1 && cont >= 9'd6)begin
77 cont <= 9'd0;
78 estado_atual <= sts1;
79 F_ITMSS <= 1'b0;
80 end
81 end
82 endcase

```

5.3 Type Frame

Este bloco define o tipo de frame que esta sendo lido pelo decoder: dados e remoto. A identificação se da por dois campos no frame no nosso decoder.

Caso o bit 13 - no formato base - ou o bit 33 - no formato estendido - seja dominante, o bloco *Type of Frame* retornará um bit dominante que representa um frame de dados; caso contrario retornará um bit recessivo que representa um frame remoto.

Adicionamos também que se caso o bit 15 - no formato base - ou o bit 34 - no formato estendido - estaremos decodificando um frame no formato CAN FD, ou seja, só existem frames de dados. Sendo assim a saída do bloco *Type of Frame* será dominante.

5.4 Stuff

É o bloco responsável por alterar o SP recebido do bit timing, verificando a repetição de bits e ignorando o posedge e negedge do antigo sinal de SP no novo sinal de SP (no momento em que o bit de stuff acontece - primeiro bit diferente após uma repetição de 5 bits iguais).

```

1 module Stuff_Block(
2     input reset, RX, SP_old, F_STF,
3     output wire SP
4 );
5
6 reg [8:0] cont;
7 reg previous_bit, SPw;
8 reg [7:0] estado_atual;
9
10 initial previous_bit = ~RX;
11 initial SPw = 1'b0;
12 initial estado_atual = 8'd0;
13
14 parameter sts0 = 0, sts1 = 1, sts2 = 2;
15 assign SP = SPw;
16
17 always@(posedge SP_old) begin
18     if(reset == 1) begin
19         previous_bit = ~RX;
20         cont <= 9'd0;
21         estado_atual <= 8'd0;
22     end
23     $display("Contador no Stuff: %d", cont);
24     case(estado_atual)
25     sts0:begin
26         if(F_STF == 1'b0) begin
27             previous_bit <= RX;
28             estado_atual <= 8'd1;
29             cont <= 9'd0;
30         end
31     end

```

Isto é feito de forma simples. No estado “sts0” o bloco espera até que a flag de stuff (F_STF) seja ativada. Caso esteja, este já é o primeiro bit que precisamos verificar, mas não há um outro para que seja possível fazer uma comparação. O bit é então guardado em uma variável chamada “previous_bit”.

```

31     end
32     sts1:begin
33         if(F_STF == 1'b0) begin
34             if(previous_bit == RX) begin
35                 $display("previous_bit == RX");
36                 if(cont >= 9'd3) begin
37                     $display("cont >= 9'd3");
38                     cont <= 9'd0;
39                     estado_atual <= 8'd2;
40                 end
41                 else begin
42                     $display("cont < 9'd3");
43                     cont <= cont + 9'd1;
44                 end
45             end
46             else begin
47                 $display("previous_bit != RX");
48                 cont <= 9'd0;
49             end
50         end
51         else begin
52             cont <= 9'd0;
53             estado_atual <= 8'd0;
54         end
55         previous_bit <= RX;
56     end
57     sts2:begin
58         estado_atual <= 8'd0;
59         cont <= 9'd0;
60     end
61 endcase
62 end

```

No estado “sts1” ocorre uma série de comparações.

No estado “sts1” ocorre uma série de comparações. Simplificando:
 Verifica o estado da FLAG de stuff
 Se Flag estiver ativada
 Compara o bit anterior com o atual
 Se os bits forem iguais
 Se contador chegou a 3
 Muda para estado “sts2”
 Caso contrário
 Incrementa Contador
 Caso Contrário
 Zera Contador e se mantém no estado
 Caso contrário
 Zera contador e muda para estado “sts0”
 Ao final, o “previous_bit” é atualizado com o valor de RX.

```

63 always@(*) begin
64   case(estado_atual)
65   sts0:begin
66     SPw = SP_old;
67   end
68   sts1:begin
69     SPw = SP_old;
70   end
71   sts2:begin
72   end
73   endcase
74 end
75 endmodule
76

```

No estado “sts2” apenas mudamos para o “sts0”, reiniciando o bloco. Combinacionalmente, nos estados “sts0” e “sts1”, fazemos com que o novo SP repita os posedges e negedges do antigo SP (SP_old). No estado “sts2” fazemos o novo SP ignorar o valor do SP_old.

5.5 ERROR Block

Este bloco é responsável por detectar os erros passivos que foram determinados na especificação. Com exceção do erro de CRC que por questões de prazos e projeto decidiu-se por não ser implementado. Este bloco consegue detectar Erros de Bit Stuffing, EOF e erros de forma no CRC e ACK Delimiter. Cada um desses erros vai ser detalhadamente explicado nas suas respectivas sessões.

A saída dos blocos de erro se liga diretamente a uma porta AND, e isto se deve ao fato do protocolo CAN considerar 0 lógico como aplicação de DDP entre o CAN High e CAN Low e o valor lógico 1 como a falta de DDP.

Cada um destes blocos recebe uma flag respectiva ao tipo de erro que tem que ser monitorado. Por exemplo, durante os campos em que pode ocorrer stuff a flag F_STF que permite a detecção de stuff estará com um valor dominante.

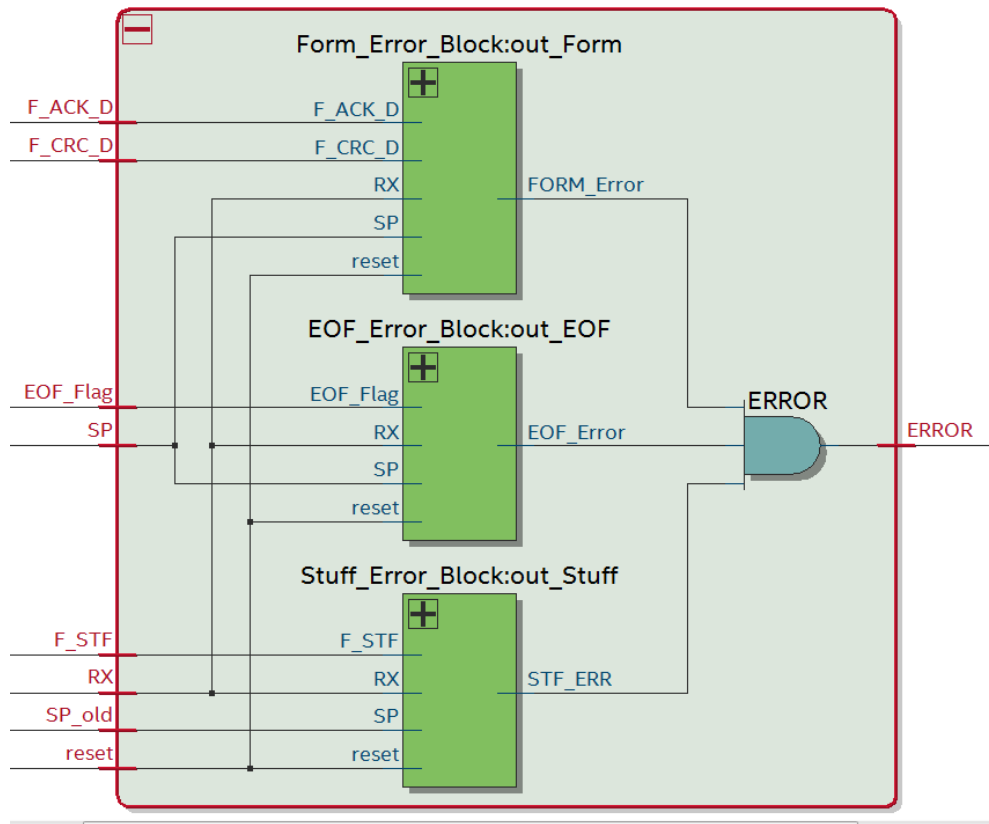


Figura 8: *Error Block*

5.5.1 Bit Stuffing Error Block

Como já explicado anteriormente um stuff ocorre quando cinco bits de mesma magnitude - sendo eles dominantes ou recessivos - ocupam o barramento de forma intermitente. Quando este comportamento é percebido, devido a característica NRZ - Non Return to Zero do protocolo CAN um edge é forçado para que ocorra a sincronização, neste caso Re-Synchronization. Este comportamento é seguido pelo frame entre o SOF - Start of Frame até o CRC Sequence. Caso em algum momento um sexto bit seja detectado nesta região, o bloco BitStuffError deixa como saída um bit dominante que irá informar ao Frame Maker a ocorrência do erro.

5.5.2 EOF Error Block

EOF - End of Frame é caracterizado por uma sequência de sete bits recessivos. Este campo indicará que chegamos ao final do frame e assim não ocorrendo nenhum erro até o ultimo bit, toda a operação de transmissão no

barramento foi bem sucedida. É dito até o ultimo bit pois o último bit de cada frame é tratado como *don't care*, isto é, caso ocorra um bit dominante no último bit de cada frame um erro de EOF não será detectado.

Um Erro de EOF é detectado quando sua flag, EOF_Flag, está ativada e um bit dominante é detectado neste campo. Caso ocorra, um bit dominante será liberado na saída deste bloco para que o Frame Maker note o erro e o Frame Maker passe a tratar o erro.

5.5.3 Form Error Block

Form Error Block é a unidade do bloco de erro que detecta erros de forma. Apesar do EOF também ser considerado um erro de forma, por decisão de projeto considerando uma melhor separação modular de códigos, os erros de forma no ACK Delimiter e no CRC Delimiter ficaram separados do erro de forma do EOF. No caso para erros de forma no ACK e CRC Delimiter, o erro é detectado quando a flag esta ativada e um bit dominante é notado, pois ACKs e CRCs Delimiters devem ser obrigatoriamente recessivos.

6 Test Bench e Waveforms

```
1  module Main_Block_TB;
2
3      reg reset, RX, SP_old;
4      wire Type_Frame, IDE, EDL, BRS;
5      wire F_STF, F_EOF, F_CRC, F_IDF, F_ACK_D, F_CRC_D;
6      wire [10:0] IDF;
7      wire [17:0] IDF_ex;
8      wire [3:0] DLC;
9      wire [6:0] EOF;
10     wire ERROR, SP;
11
12     Main_Block block (reset, RX, SP_old, Type_Frame, IDE, EDL, BRS, F_STF, F_EOF, F_CRC, F_I
13
14     initial
15         SP_old = 0;
16
17     always
18         #3 SP_old = ~SP_old;
19
```

No testbench, temos o SP atualizando seu valor após um intervalo regular (#3). O primeiro frame é um frame Base do tipo Data. Os comentários no código abaixo mostram os momentos em que surgem bits de stuff.

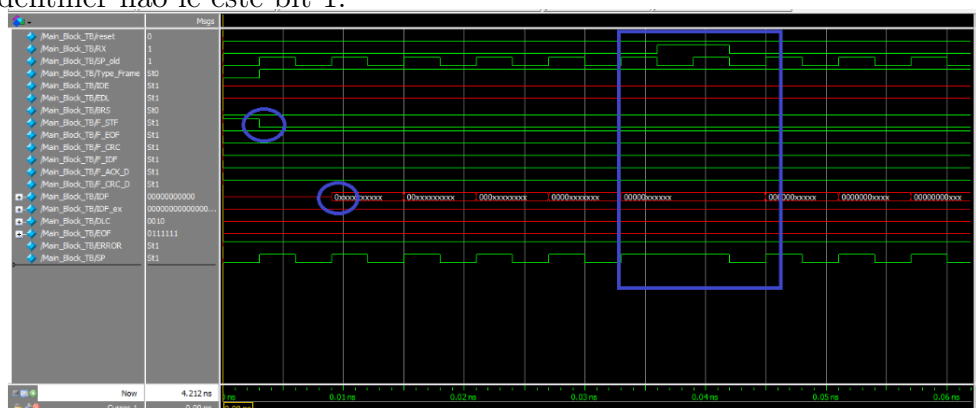
```

19
20 initial begin
21
22     RX = 0 ; reset = 0; # 6; //SOF CAN Base Data
23     RX = 0 ; reset = 0; # 6; //Identifier Begins
24     RX = 0 ; reset = 0; # 6;
25     RX = 0 ; reset = 0; # 6;
26     RX = 0 ; reset = 0; # 6;
27     RX = 0 ; reset = 0; # 6;
28     /* STUFF */ RX = 1 ; reset = 0; # 6;
29     RX = 0 ; reset = 0; # 6;
30     RX = 0 ; reset = 0; # 6;
31     RX = 0 ; reset = 0; # 6;
32     RX = 0 ; reset = 0; # 6;
33     RX = 0 ; reset = 0; # 6;
34     /* STUFF */ RX = 1 ; reset = 0; # 6;
35     RX = 1 ; reset = 0; # 6; //Identifier Ends
36     RX = 0 ; reset = 0; # 6;
37     RX = 0 ; reset = 0; # 6;
38     RX = 0 ; reset = 0; # 6;
39     RX = 0 ; reset = 0; # 6;
40     RX = 0 ; reset = 0; # 6;
41     /* STUFF */ RX = 1 ; reset = 0; # 6;
42     RX = 1 ; reset = 0; # 6;
43     RX = 0 ; reset = 0; # 6;
44     RX = 0 ; reset = 0; # 6; //DATA Begins
45     RX = 0 ; reset = 0; # 6;
46     RX = 0 ; reset = 0; # 6;
47     RX = 1 ; reset = 0; # 6;
48     RX = 1 ; reset = 0; # 6;
49     RX = 0 ; reset = 0; # 6;
50     RX = 1 ; reset = 0; # 6;
51     RX = 0 ; reset = 0; # 6;
52     RX = 1 ; reset = 0; # 6;
53     RX = 0 ; reset = 0; # 6;
54     RX = 1 ; reset = 0; # 6;
55     RX = 1 ; reset = 0; # 6;
56     RX = 1 ; reset = 0; # 6;
57     RX = 1 ; reset = 0; # 6;
58     RX = 0 ; reset = 0; # 6; //Data Ends
59     RX = 0 ; reset = 0; # 6; //CRC Begins
60     RX = 1 ; reset = 0; # 6;

```

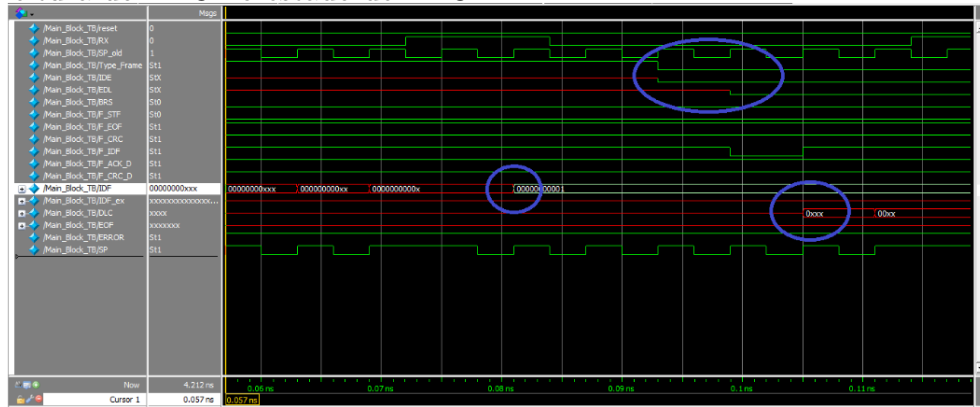
No começo do frame, podemos observar no waveform a ativação da flag de stuff (primeiro círculo). Logo após, notamos o começo da leitura do identificador. Seguindo em diante, visto que o identificador é composto por 10 bits '0' e 1 bit '1', temos um bit de stuff.

No quadrado marcado no waveform, é possível notar que o sample point ignora um período do sample point antigo exatamente no momento que um bit 1 aparece no RX, um bit de stuff. Também é possível perceber isto porque o identifier não lê este bit 1.

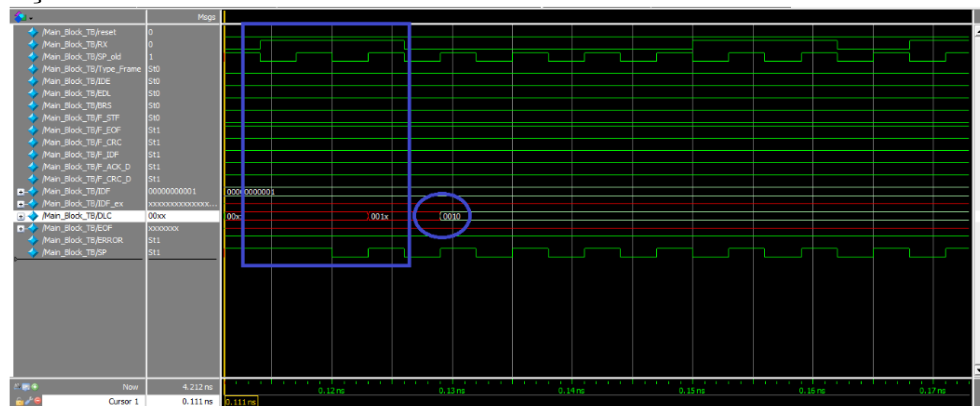


Em seguida, pela imagem abaixo, é possível perceber o fim da leitura do identifier (000000000001 = 1 em decimal). Em seguida o type_frame é atualizado para 0 (segundo círculo). 0 indica Data, 1 indica Remote. Em um ciclo o IDE é atualizado para 0 a partir do RX, o que indica um frame

com cabeçalho base. No ciclo seguinte, o EDL é atualizado para 0, indicando que se trata de CAN, e não CAN FD. No terceiro círculo notamos o início da leitura do DLC no estado de DLC.



Na imagem abaixo podemos notar um segundo bit de stuff sendo ignorado no meio do DLC, e em seguida, no círculo, o final da leitura do DLC e o começo da leitura dos dados.



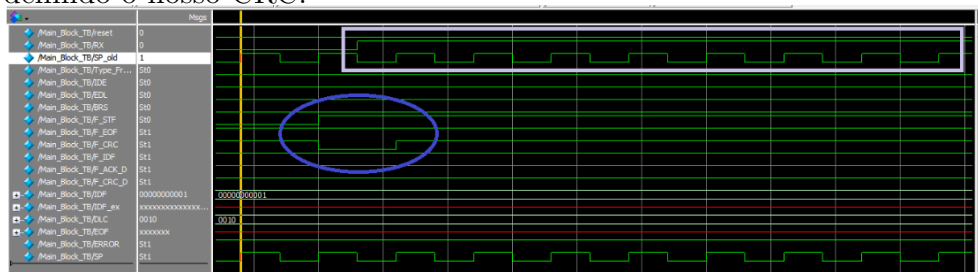
Seguindo no testbench, temos o fim dos dados e o começo do CRC, até o fim do frame, o intermission, bus idle e o começo de um segundo frame.

```

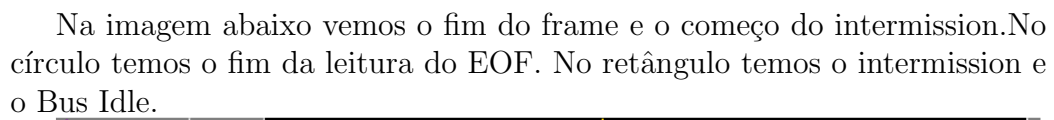
59 RX = 0 ; reset = 0; # 6; //Data Ends
60 RX = 1 ; reset = 0; # 6; //CRC Begins
61 RX = 1 ; reset = 0; # 6;
62 RX = 1 ; reset = 0; # 6;
63 RX = 1 ; reset = 0; # 6;
64 RX = 1 ; reset = 0; # 6;
65 RX = 1 ; reset = 0; # 6;
66 RX = 1 ; reset = 0; # 6;
67 RX = 1 ; reset = 0; # 6;
68 RX = 1 ; reset = 0; # 6;
69 RX = 1 ; reset = 0; # 6;
70 RX = 1 ; reset = 0; # 6;
71 RX = 1 ; reset = 0; # 6;
72 RX = 1 ; reset = 0; # 6;
73 RX = 1 ; reset = 0; # 6;
74 RX = 1 ; reset = 0; # 6; //CRC Ends
75 RX = 1 ; reset = 0; # 6;
76 RX = 1 ; reset = 0; # 6;
77 RX = 1 ; reset = 0; # 6;
78 RX = 1 ; reset = 0; # 6;
79 RX = 1 ; reset = 0; # 6;
80 RX = 1 ; reset = 0; # 6;
81 RX = 1 ; reset = 0; # 6;
82 RX = 1 ; reset = 0; # 6;
83 RX = 1 ; reset = 0; # 6;
84 RX = 1 ; reset = 0; # 6; //Frame Ends
85
86 RX = 1 ; reset = 0; # 6; //INTERMISSION Begins
87 RX = 1 ; reset = 0; # 6; //INTERMISSION ENDS
88
89 RX = 1 ; reset = 0; # 6; //BUS IDLE
90 RX = 1 ; reset = 0; # 6; //BUS IDLE
91
92 RX = 0 ; reset = 0; # 6; //SOF CAN Base Data
93 RX = 0 ; reset = 0; # 6; //Identifier Begins
94 RX = 0 ; reset = 0; # 6;
95 RX = 0 ; reset = 0; # 6;
96 RX = 0 ; reset = 0; # 6;
97 RX = 0 ; reset = 0; # 6;
98 /* STUFF */ RX = 1 ; reset = 0; # 6;
99 RX = 0 ; reset = 0; # 6;
100 RX = 0 ; reset = 0; # 6;

```

Na imagem abaixo podemos ver a flag de stuff sendo desativada após o campo de dados e a flag de CRC sendo ativada, indicando que o próximo estado é de CRC (círculo). No retângulo vemos uma sucessão de bits 1, como foi definido o nosso CRC.



Na imagem abaixo temos a flag de CRC delimiter sendo ativada, indicando que o próximo estado é o CRC delimiter (primeiro círculo). Em seguida temos a flag de ACK delimiter sendo ativada, indicando que o próximo estado é o ACK delimiter (segundo círculo). Após isso temos a flag de EOF sendo ativada, indicando que o próximo estado é o End of Frame (Terceiro círculo). Por fim, temos um retângulo indicando a leitura do EOF.



7 RTLs

Abaixo serão mostrados os RTLs que puderam ser gerados pelo Quartus 17.0. É possível notar que os RTLs gerados pelo Quartus se aproximaram bastante da nossa projeção durante a fase de prototipação que pudemos gerar o diagrama de bloco da sessão 2.

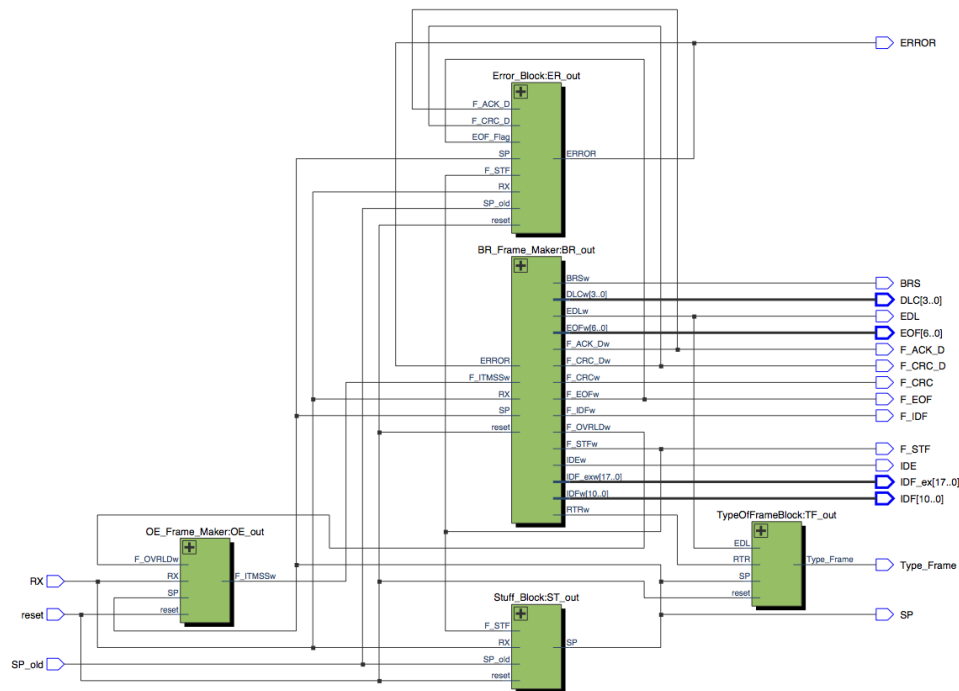


Figura 9: *Main Block*

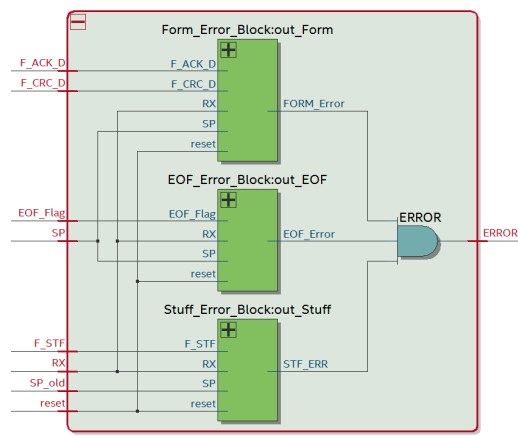


Figura 10: *Error Block*

8 Referências

1. NXP - CAN Bit Timing Requirements
2. CAN 2.0 Specification
3. CAN with Flexible Data-Rate Specification
4. Wikipédia: CAN BUS - https://en.wikipedia.org/wiki/CAN_bus
5. Understanding and Using the Controller Area Network - DI NATALE, MARCO