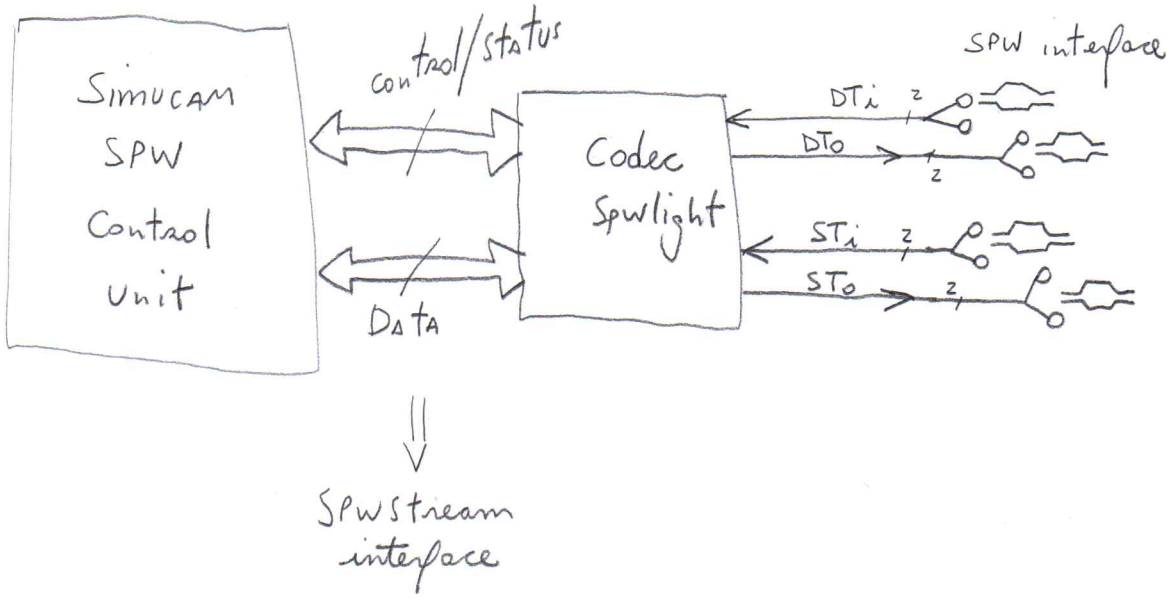
IMT - NSEE - SpW error injection - Exchange Level

- Alterações no codec spw light (v20130504);
- Inclusão de lógica de injeção de erros:
  - Desconexão;
  - Paridade;
  - Escape;
  - Credit;
  - Char sequence.
- Interface SpwStream;
- Implementação "generic" do tx e do Rx;

Visão macro - uma instância de canal spw:



- Validações via testbench => simulações em Modelsim.

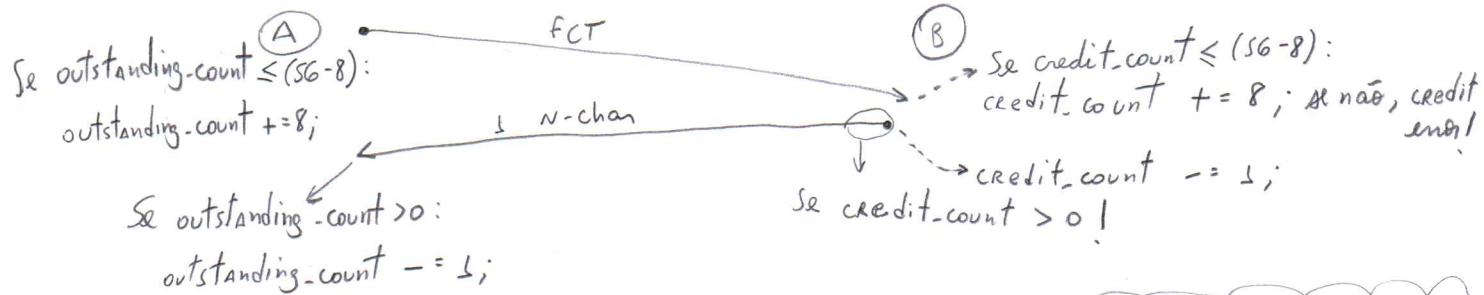
- Revisão - pontos importantes do protocolo SPW
  (ECSS-E-ST-50-12C - JUL/2008)

Tipos de chars:

- L-chars: FCT, ESC, NULL (ESC + FCT), time-code (ESC + data)
- N-chars: data, EOP, EEP (vão p/ packet level).

Escape sequences (não vão p/ packet level)

Flow control:



credit_count : tx_credit ✓
outstanding_count : Rx_credit ✓

Prioridade:

Maior: Time-Code
FCT
N-char
Menor: Null

Erros (todos detectáveis pela máquina de Rx):

1. Desconexão: (s/ transição nas linhas D, S por mais de 850 ns); — após 1º bit Rx.
2. Paridade;
3. Escape: (se receber um ESC seguido de char ≠ FCT ou data (→ ESC/EOP/EEP)); — Em qq estado
4. Credit: Em duas situações (só em RUN):
   a) Rx data sem estar esperando: outstanding_count = ϕ. (NULL / Time-Code)
   b) Rx FCT, mas credit_count já é > (56-8)!

E: Detectável pela state machine de link:

5. Character sequence: (não gera flag de saída!) => só ocorre fora do estado RUN!
   - Premissa: Ignorar Rx de chars ≠ NULL antes de Rx Null;
   a) • ERR: FCT Rx nos estados: ErrorWait, Ready, Started;
   b) • ERR: N-char / Rx / Time-code fora do estado Run.

Após 1º NULL Rx.

Restrições de implementação:

- Injeção de erros nº 1, 2, 3, 4: só em RUN.

- Não dependemos do top level spwstream. Ao final da implementação, apenas replicar os ports;
- O componente spwerr será um subcomponente instanciado pelo spwlink.

• Testbenches - spw light codec

Estudo do tb-spwlink: Erros gerados através de "stimuli" direto no front-end Rx (pois não está em loopback nesse caso):

(Em RUN)
a) Disconnect -> Interrompe o Rx nulls por mais de 850ns. -> OK; no nosso caso, é fazer o mesmo através de um stop TX nulls, ou via link disable.

(Em RUN)
b) Paridade -> injeta bit de paridade errado no Rx. Depois de um NULL, manda um dado c/ paridade, e depois repete o mesmo dado, sem Acertar a paridade!

DADO: P Φ 10101010 (MSB)

c) Character Sequence -> Ao invés de seguir mandando no RX NULLs ou FCT no estado "connecting", mandou um EOP. Isso foi entendido pelo Rx como um dado (n-char), provocando um reset.

d) ESC -> Em RUN, estava mandando no Rx null's normalmente:

P P
... Φ 111 | 0 100 ... -> ESC + FCT = NULL

Até que "mandou" no RX:

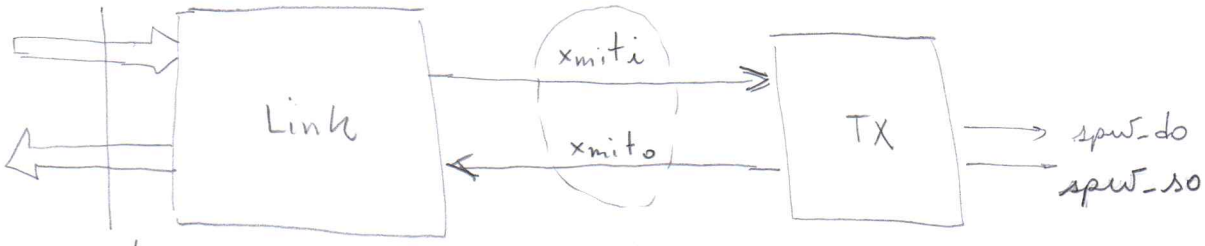
... 0 111 | 0 111 -> ESC + ESC -> Erro! Que ocorreria também com:
... Φ 111 | Φ 110 -> ESC + EEP
... Φ 111 | Φ 101 -> ESC + EOP

e) Credit -> Em run, mandou 8 FCT's de uma vez no Rx. O spwlink incrementou de 8 cada FCT Rx o Tx_credit, mas ao ultrapassar 56 (7 FCT's no máx.), ocorreu erro de credit. ATENÇÃO: Podemos enviar 8 FCT's, mas sem incrementar nosso RX-credit!

TODO ok! Com isso, estudar a interface spwlink ↔ TX, para saber como atuar no TX via o link apenas.

TODO ok! Criar componente spwerr.vhd (COMPLETO!), declarar no spwpkg.vhd, e instanciar dentro do spwlink. ou TODO Fazer o tratamento de err inj. no link //

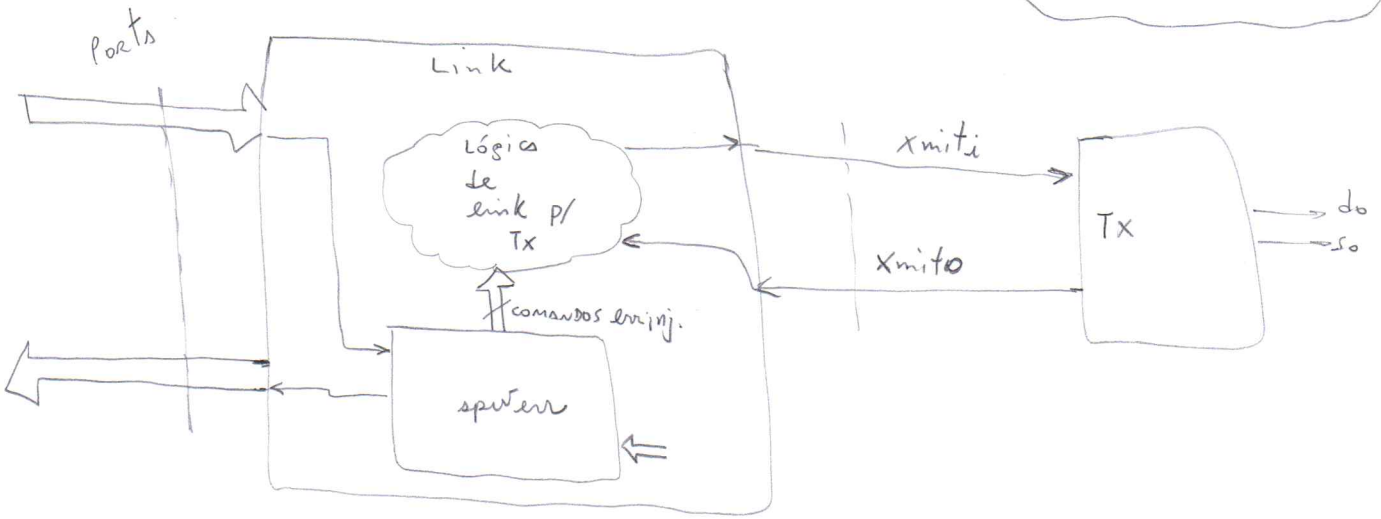
Interface Link ⟷ Tx :



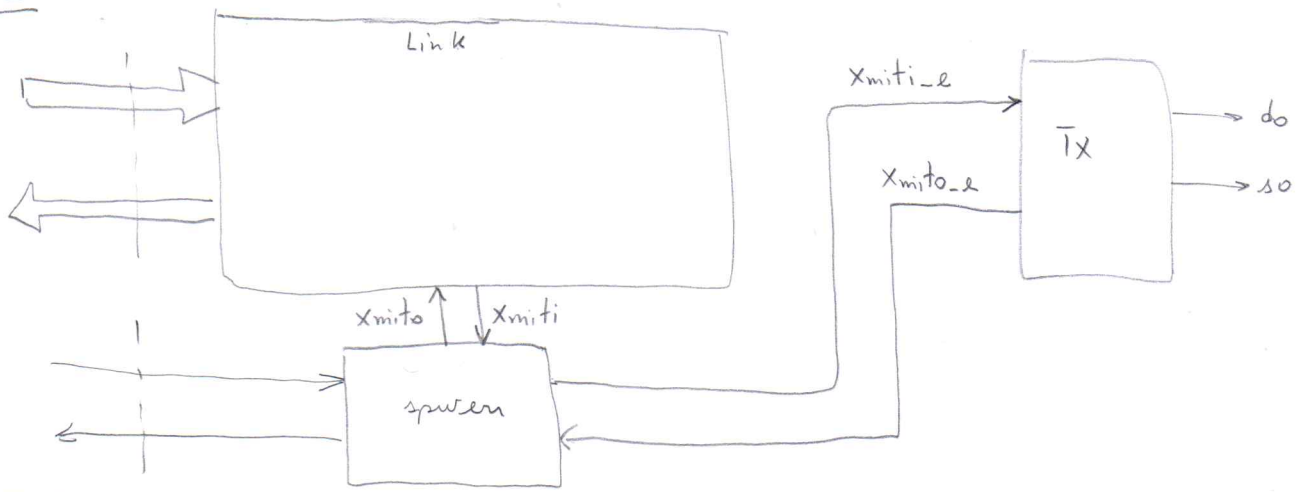
Ports
p/ outros
componentes
e/ou TOP LEVEL
(spwstream)

Componente spwerr - opção 1 :

Avaliando prós e contras, a escolha é a opção 1



Opção 2 :



Sinais de interesse na interface Link ⟷ TX:

a) type: spw_xmit_in_type : link ⟶ Tx :

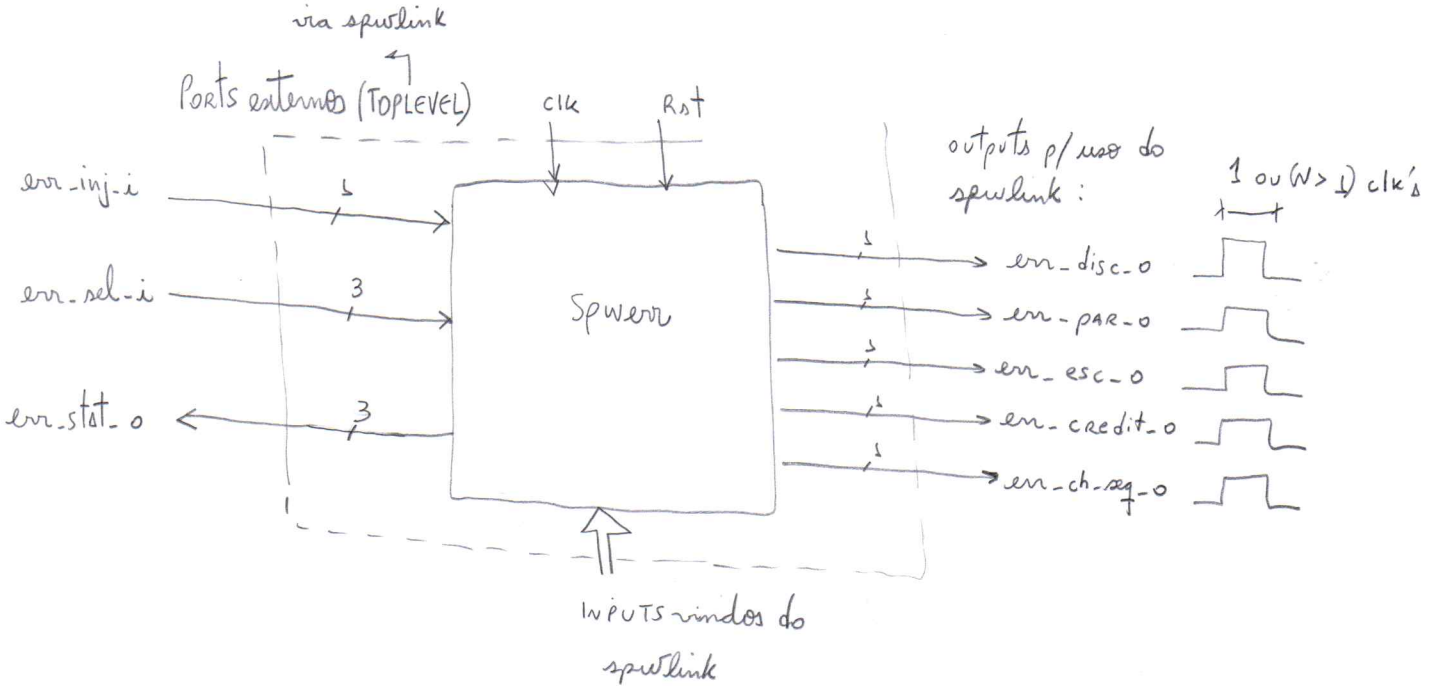
bits {
- txen : 1 : tx habilitado
- stnull : 1 : apenas NULL podem ser enviados
- stfct : 1 : apenas NULL e/ou FCT podem ser enviados
- fct_in : 1 : Requisita tx de um FCT (manter em '1' até fctack = '1').
- txwrite : 1 : Requisita tx de um N-char (manter em '1' até txack = '1').
- txflag : '0' = data char / '1' = EOP / EEP

vector 8 bits {
- txdata : Dado / 00000000 = EOP / 00000001 = EEP.

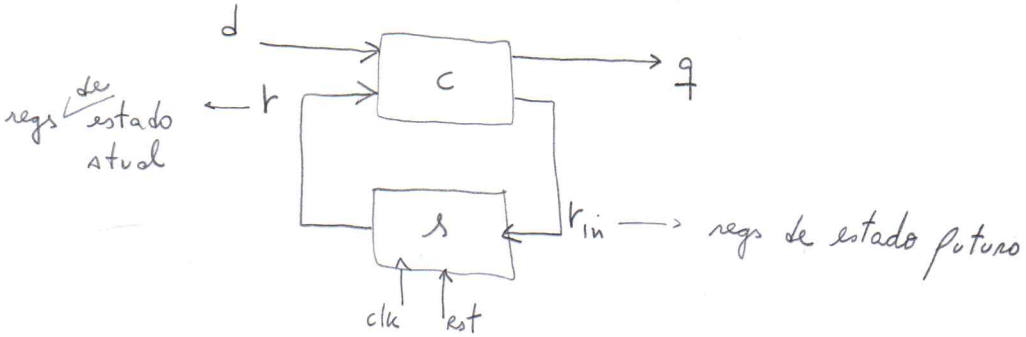
b) type: spw_xmit_out_type : link ⟵ tx :

bits {
- fctack : 1 : Confirma tx de um Fct. Cai quando fctin ↓.
- txack : 1 : Confirma tx de um N-char. Cai quando txwrite ↓ (ver também dependência com fctin).

Spwerr - diagrama de blocos



- O componente spwerr será uma state machine, com o conceito de arquitetura com dois processos: sequencial e combinatório:



- Types (records) de dados:

vão também p/ o toplevel spwstream

- err_usr_i → entradas de usuário: { err_inj_i, err_sel_i }
- err_usr_o → saídas de usuário: { err_stat_o }
- err_link_i → entradas vindas do spwlink: a definir (de tudo que está disponível como input do spwlink)
- err_link_o → saídas para o spwlink: { err_disc_o, err_par_o, err_esc_o, err_credit_o, err_ch_seq_o }

Spwsn - descritivo de ports :

- err_inj_i : o ↯ dispara a injeção do erro (err_sel_i), se as condições p/ cada erro forem válidas. Só é tratado se err_stat_o = Stby !

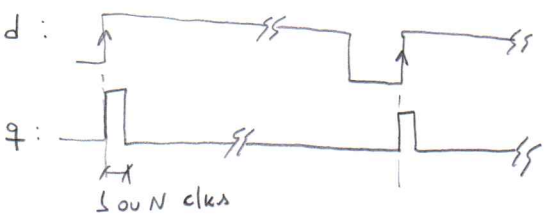
- err_sel_i : seleção de erro :
  - ΦΦΦ - Disconnect
  - ΦΦ1 - Parity
  - Φ1Φ - ~~Escape : ESC + EOP~~ Reserved
  - Φ11 - ~~Escape : ESC + EEP~~ Reserved
  - 1ΦΦ - Escape : ESC + ESC
  - 1Φ1 - Credit
  - 11Φ - Char sequence
  - 111 - Reserved.

  } implementações futuras, se necessário. → Aumentar ports de saída p/ spwlink.

- err_stat_o : status da máquina spwsn :
  - ΦΦΦ : Stby.
  - ΦΦ1 : err_inj accepted / in execution.
  - Φ1Φ : err_sel invalid.
  - Φ11 : inconsistent err_inj request.
  - 1ΦΦ : err_inj ended ok.
  - 1Φ1 ~ 111 : Reserved

  ↓ retorna p/ Stby com a queda de err_inj_i.

Exemplo de código vhdl p/ FSM - dois processos :



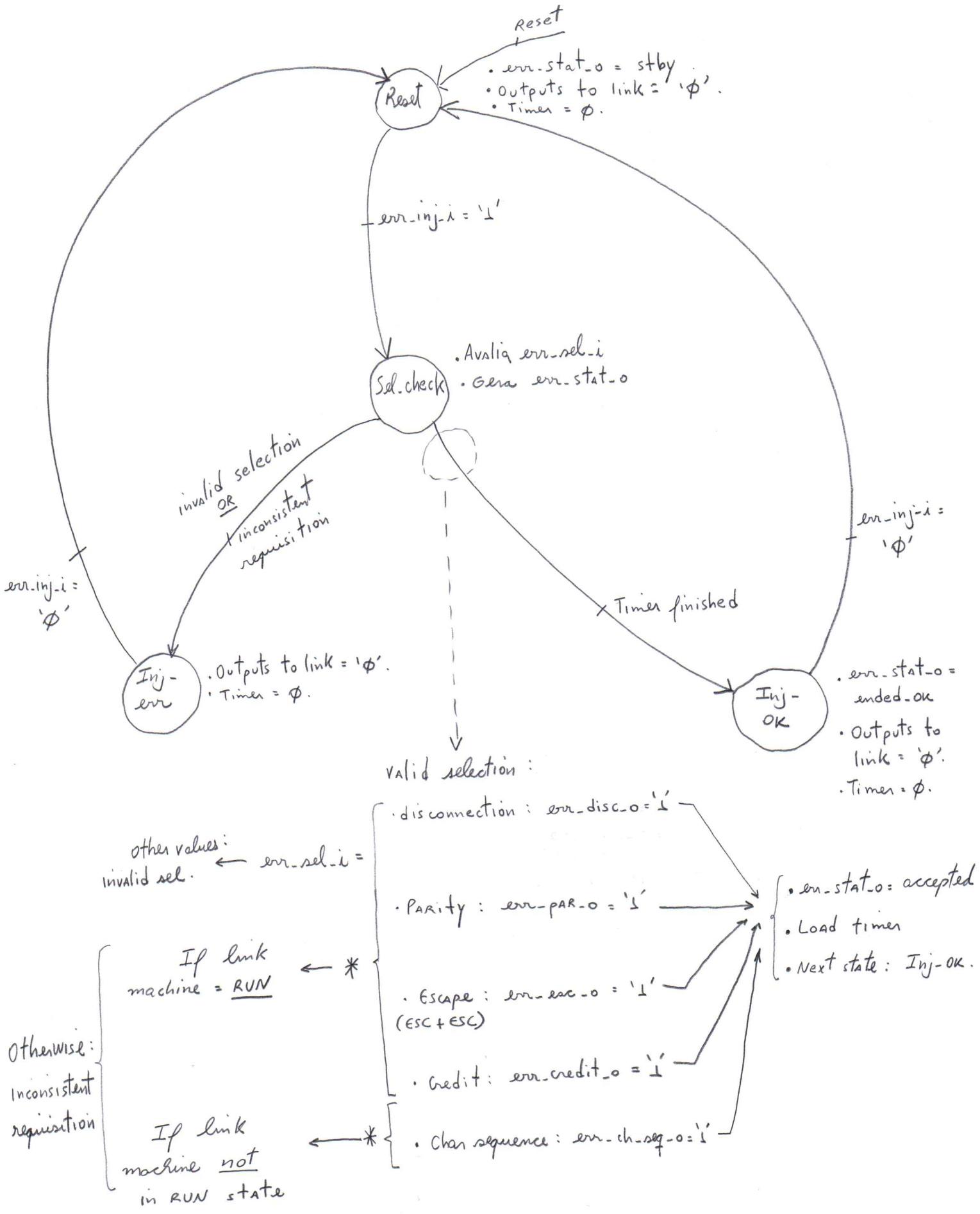
Regs de estado : reg_type r, r_in (Atual ↓, Futuro ↓)

```
Process comb (d, r)
    var v : reg_type;
begin
    v := r;
    Se v.state = st_A ==> Se d:0 --> q<=Φ; else se d='1' --> q<='1';
                                                            v.state = st_B;
    Se v.State = st_B ==> q<=Φ; Se d='Φ' --> v.state=st_A;
    ⋮
    r_in <= v;
end
```

(assíncrono)

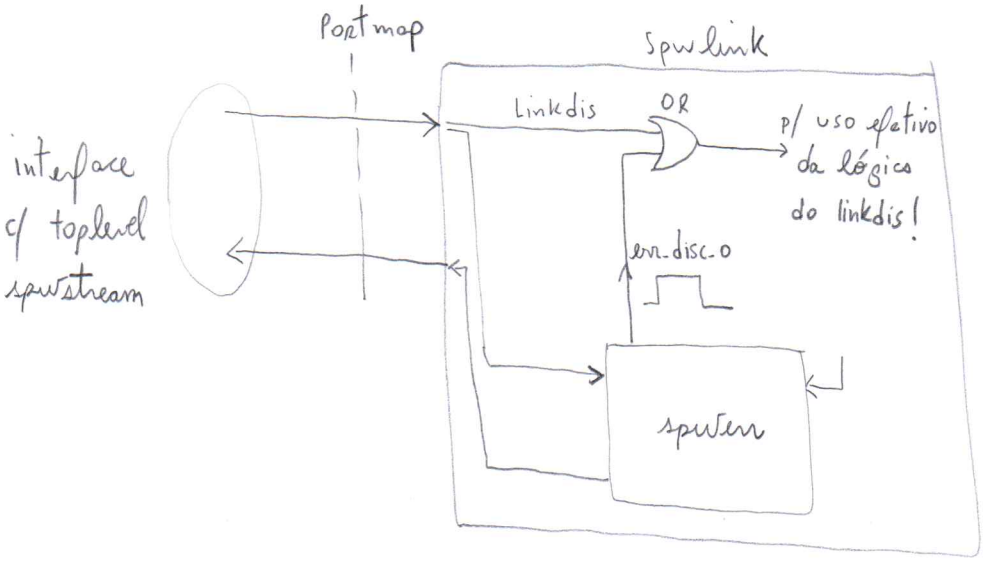
```
Process seq (rst, clk)
begin
    Se rst='1' --> <reseta r, etc>;
    Elseif (rising_edge(clk)) --> r <= r_in;
end  endif;
```

SpWerr - diagrama de estados



Lógica :

ou! \ a) Disconnect error : feita somente via link.



No spwlink :

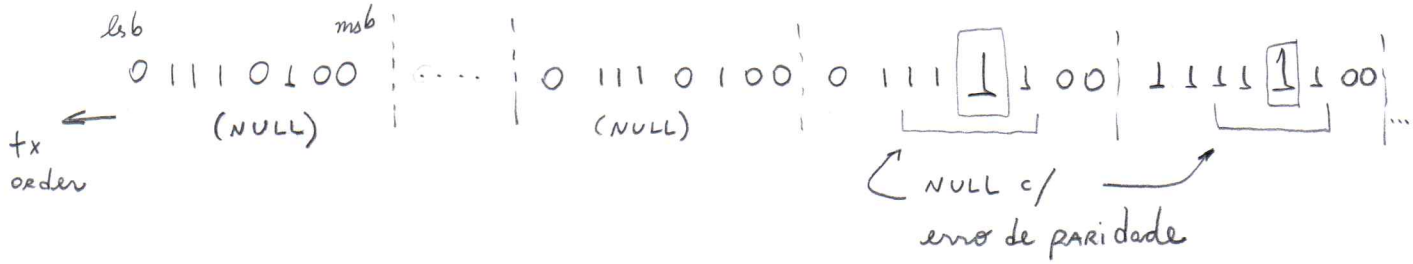
- Para os demais erros → conceito parecido : interceptar os signals pertinentes que vão p/ o tx, inserindo lógicas de err injection:
  - Sinais que vão p/ o tx : carrega em var v (mesmo type) ;
  - if/Elsif ... Else < requisição de erro n > : ← todos na lista de sensibilidade!
    - Adeque o sinal de v.<xxxxxx> conforme o err solicitado.
  - Saídas P/ o TX <= v !

Observar que, sem erros requisitados, é bypass dos signals.

- E, além disso, deixa uma FSM implementada, pois as lógicas de injeção de erro via tx podem precisar de estados.

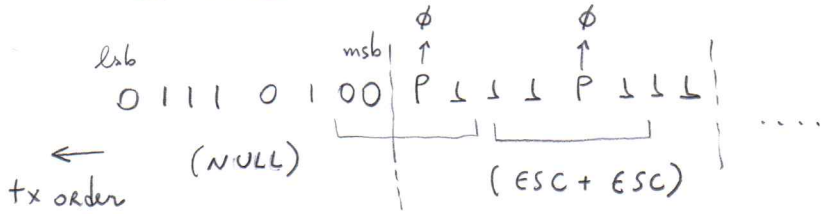
a) Paridade (só em RUN ; 8 ou 10 tx clks)

. Necessita de intervenção direta no tx. Pode ser feita no bloco de envio de NULL, ou N-char DATA. Escolhemos o NULL:



b) Escape (ESC + ESC) ; (só em RUN ; 8 tx clks)

. Necessita intervenção direta no tx. Feito no bloco de envio de NULL:



c) Char sequence : (em STARTED/CONNECTING) ; (4 tx clks).

. Gerado principalmente no link, com pequena intervenção no tx.

. Enviado um EOP (N-char) nos estados STARTED/CONNECTING (tx enabled)

• Não decrementa tx_credit !

• EOP = [P 1 ϕ 1]

• Nos estados ≠ RUN, o tx_clk = 10 MHz. Como a injecção de erro é baseada na duração do respectivo pulso (err_xxxx_0), deve-se garantir a parametrização correcta dessa duração. Ex:

clk = 200 MHz ; tx_clk = 10 MHz => 20:1 => 4 tx_clk's ≡ 80 clks.

d) Credit (em RUN ; 32 tx clks).

. Gerado principalmente no link, com pequena intervenção no tx.

. Envio de 8 x FCT's seguidos.

. Sem incrementar Rx_credit local!