- Abbiamo già detto che nel ciclo di lettura dello spazio di I/O non è possibile settare l'indirizzo in simultanea ad /ior (che viene abbassato per indicare l'operazione di lettura).

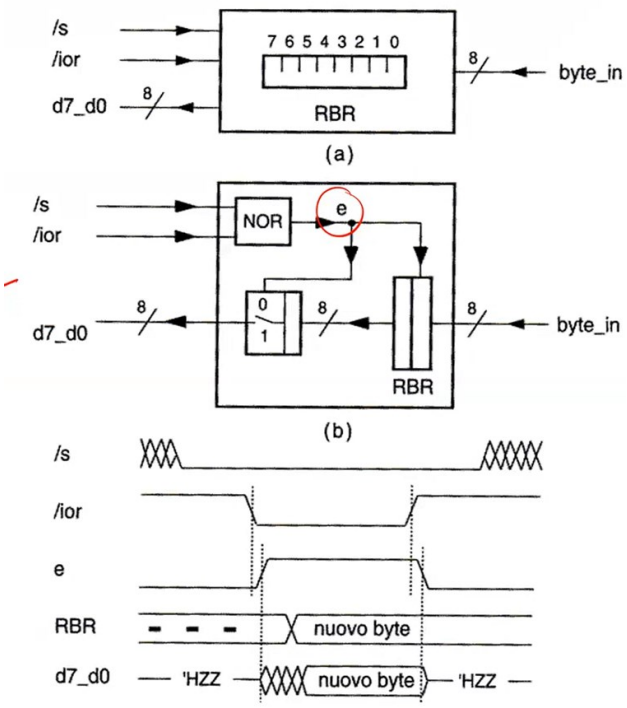
- **Perché non possiamo farlo?**
  - Sappiamo che /s è un'uscita combinatoria che può ballare.
  - Se questo piedino balla con /ior a zero può succedere che si indichi un comando di lettura del registro RBR. L'interfaccia si sincronizza col dispositivo a valle per indicare che il processore ha letto il registro e che quindi può essere sovrascritto. Leggere un registro, in alcuni casi, significa dire al relativo dispositivo di svolgere delle operazioni.
  - Segue che /ior deve essere abbassato solo ed esclusivamente dopo che gli indirizzi si sono stabilizzati.

Interfacce parallele

Interfaccia parallela di ingresso senza handshake

- L'interfaccia presenta un registro RBR, un /s, un /ior, i fili di dati d7_d0. Dal lato del dispositivo abbiamo otto fili con cui viene indicato il byte in ingresso.
- Non avendo operazioni in uscita non è necessario possedere un registro TBR e la variabile /iow.

- **Come è fatta l'interfaccia internamente?**
  - Abbiamo un registro RBR che salva i dati in ingresso.
  - Il clock deve essere fornito sulla base degli accessi fatti all'interfaccia. Quando il processore vuole accedere il registro campiona!
  - Il comando di memorizzazione non sarà dato col solito generatore di impulsi: esprimiamo un comando di memorizzazione quando il processore vuole accedere a questa interfaccia.
  - La cosa può essere ottenuta attraverso una porta NOR che ha in ingresso /s ed /ior: restituisco 1 soltanto se entrambe sono abbassate a zero. Se esprimo 1 le porta tristate viene abbassate e il dato in ingresso campionato.



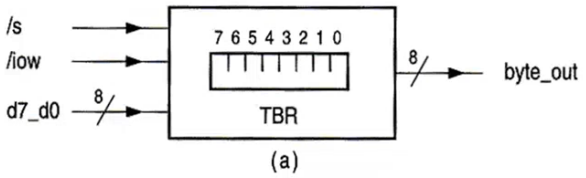
  - **Temporizzazione**:
    - Si stabilizzano gli indirizzi
    - Abbasso /ior a zero: con un leggero ritardo dovuto alla porta NOR le tri-state vanno in conduzione e il registro RBR campiona il nuovo byte.
    - Dopo un tempo di propagazione RBR presenterà il nuovo byte.
    - Non appena /ior ritorna ad 1 le tristate vanno in alta impedenza.

  - **Descrizione Verilog** (un po' sintesi-oriented, cit.):

```
module Interfaccia_Parallela_di_Ingresso(d7_d0,s_,ior_,byte_in);
 input s_,ior_;
 output[7:0] d7_d0;
 input[7:0] byte_in;
 reg[7:0] RBR;
 wire e; assign e=({s_,ior_}=='B00)?1:0; //e=~(s_|ior_)
assign d7_d0=(e==1)?RBR:'HZZ;
 always @(posedge e) #3 RBR<=byte_in;
endmodule
```

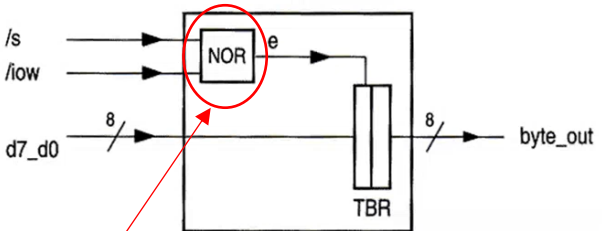
Interfaccia parallela di uscita senza handshake

- L'interfaccia presenta un solo registro TBR. Non ho bisogno di un indirizzo visto che la porta è 1, ho solo /s ed /iow. Dalla parte del dispositivo ho 8 fili in uscita.

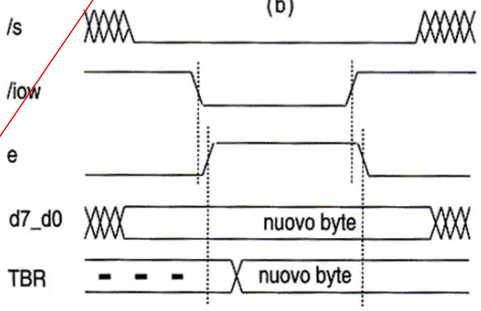


(a)

- All'interno dell'interfaccia ho una porta NOR che esprime il clock per il registro TBR.
- Non sono necessarie porte tri-state, ovviamente.
- Ricordiamo che il comando di memorizzazione è il fronte di discesa e non quello di salita.
- I dati devono essere pronti prima del fronte di discesa.
- All'arrivo del clock, con un po' di ritardo, i fili di uscita si adegueranno al nuovo byte.



(b)



- **Perché non posso fare un'interfaccia che memorizza sul fronte di salita?**
  - Per prima cosa avrei bisogno di una porta AND avente in ingresso /s negato ed /iow. Restituisco 1 solo con /s uguale a zero e /iow uguale ad 1 (cioè quando /iow viene alzato).
  - Questa cosa non funziona: l'uscita /s (uscita combinatoria) balla, questo significa dare continuamente comandi di memorizzazione al registro. Con la porta NOR siamo certi che daremo il comando di memorizzazione solo dopo la stabilizzazione di /s.

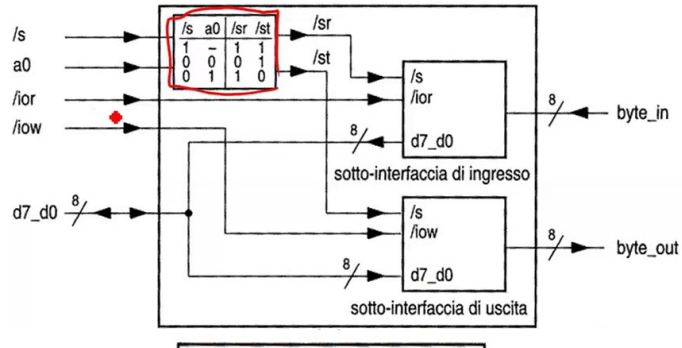
- **Descrizione in Verilog**:

```
module Interfaccia_Parallela_di_Uscita(d7_d0,s_,iow_,byte_out);
 input s_,iow_;
 input[7:0] d7_d0;
 output[7:0] byte_out;
 reg[7:0] TBR; assign byte_out=TBR;
 wire e; assign e=({s_,iow_}=='B00)?1:0; //e=~(s_|iow_)
 always @(posedge e) #3 TBR<=d7_d0;
endmodule
```

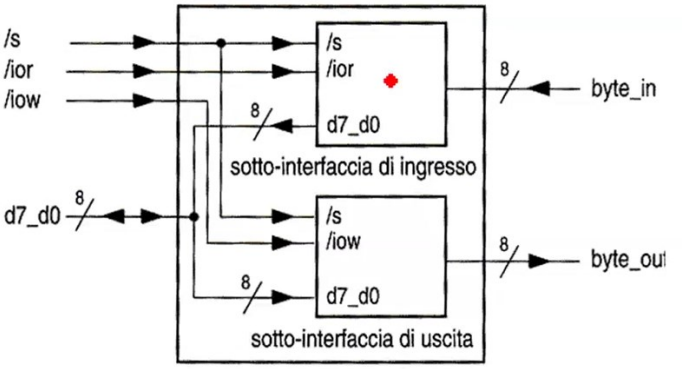
Montaggio di interfacce parallele di ingresso e uscita

- Vogliamo realizzare un'interfaccia sia di ingresso che di uscita.

- **Primo metodo**:
  - Poniamo la porta di ingresso all'indirizzo pari (0) e quella di uscita all'indirizzo dispari (1). Serve un minimo di logica combinatoria per produrre due select. I meccanismi utilizzati sono i soliti.

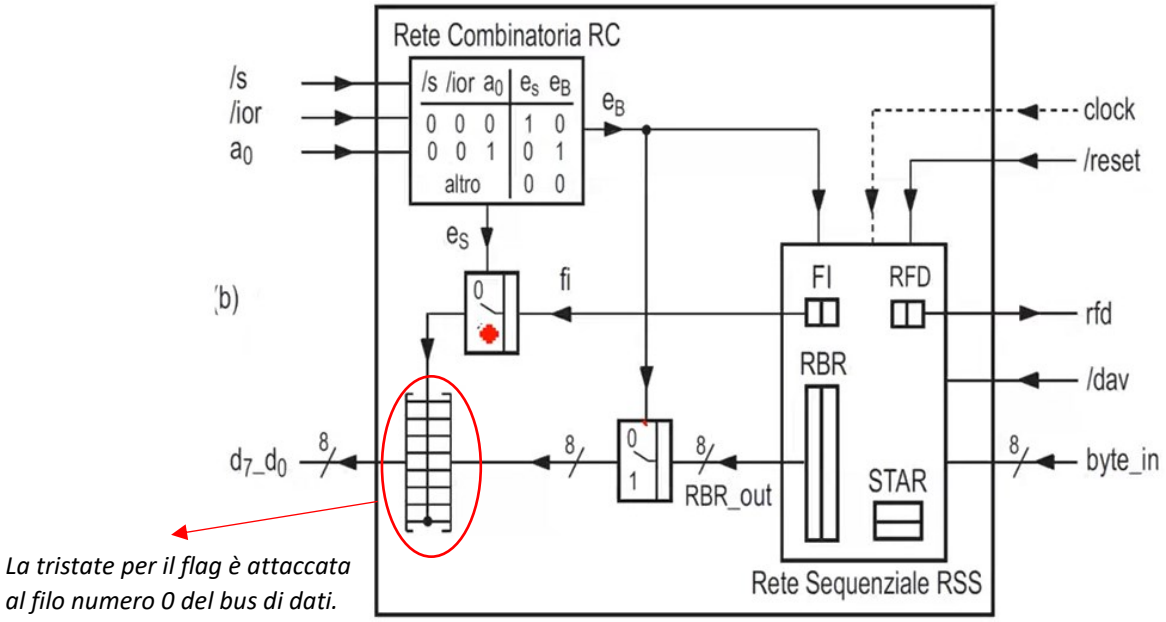


- **Secondo metodo**:
  - Le due porte sono allo stesso offset.
  - Il programmatore le riferisce con lo stesso indirizzo nello spazio di I/O, e l'accesso dipende dal tipo di accesso: è immediato dalle regole di pilotaggio che solo una delle due sotto-interfacce sarà abilitata a fare qualcosa.



Interfaccia parallela di ingresso con handshake

- L'interfaccia presenta due registri: il registro RBR, come buffer per il contenuto posto in ingresso dal dispositivo, e il registro RSR, che presenta il flag di ingresso pieno FI.
- L'interfaccia da la possibilità di svolgere operazioni di lettura su entrambi i registri.
- Nella RSS viene gestito un handshake: il dispositivo è il produttore, l'interfaccia il consumatore.



La tristate per il flag è attaccata al filo numero 0 del bus di dati.

- All'interno abbiamo una rete combinatoria che deve generare i segnali di abilitazione per le porte tri-state (e_B ed e_S), nel caso in cui il processore voglia svolgere operazioni di lettura.
- All'interno dell'interfaccia abbiamo anche una RSS che gestisce l'handshake col dispositivo e setta/resetta il flag FI. Per gestire l'handshake abbiamo bisogno del valore di e_B in ingresso nella RSS: quando e_b assume come valore 1 significa che il processore sta svolgendo un'operazione di lettura sul registro RBR.

Per quanto riguarda la disposizione delle porte tristate:

- Le tristate non sono mai in conduzione contemporaneamente:
  - Quando $e_s = 1, e_b = 0$ abbiamo la tristate relativa al flag FI in conduzione. Tutte le porte tristate relative al registro RBR sono in alta impedenza (vedere le cose a tre dimensioni…)
  - Quando $e_s = 0, e_b = 1$ le tristate relative al registro RBR sono in conduzione. La porta tristate relativa al flag FI si trova in alta impedenza.
- Quando $e_s = 0, e_b = 0$ non stiamo svolgendo operazioni di lettura, e tutte le porte tristate sono in alta impedenza.

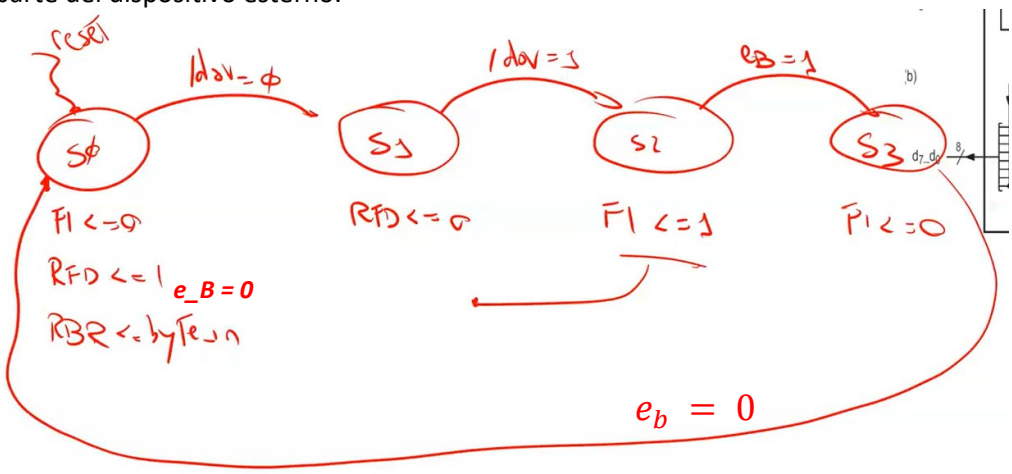
Sintetizziamo la RSS:

- Di quali registri ho bisogno? Quelli già detti, oltre al registro STAR per tenere conto della sequenza di stati.

- **Condizioni di reset**:
  - I valori per gestire l'handshake sono quelli già noti (rfd e /dav uguali ad 1)
  - Il flag di ingresso pieno va a 0 perché nessuno, al momento del reset, ha scritto qualcosa.
  - Il contenuto di RBR non è significativo e quindi non necessita di essere inizializzato.
  - Il registro STAR avrà come valore iniziale quello relativo al primo stato interno.

- **Descrizione delle operazioni negli stati** (immagine a pagina dopo):
  - **S0**: stato di riposo in cui attendiamo che il produttore (il dispositivo) fornisca un dato (cioè attendo che /dav venga abbassato). Ogni volta aggiorno il valore del registro RBR, in modo tale da averlo già pronto nello stato S1.
  - **S1**: Metto rfd a 0 per indicare che il consumatore (cioè l'interfaccia) ha ricevuto i dati e li sta elaborando. Attendo che /dav venga nuovamente alzato (gestione dell'handshake, necessario).
  - **S2**: metto il flag di ingresso pieno uguale ad 1. Rimango in questo stato finchè il processore non svolgerà un'operazione di lettura nel buffer RBR (cioè finchè non avrò $e_b = 1$).

- **S3**: il processore sta svolgendo un'operazione di lettura. Posso porre il flag di ingresso pieno a zero. Rimango in questo stato finchè l'operazione di lettura non sarà completata (il segnale è $e_b = 0$). A quel punto possiamo ritornare allo stato iniziale, ponendoci in attesa di nuovi input da parte del dispositivo esterno.



- **Descrizione in Verilog**:

```
module RSS(dav_,rfd,byte_in,fi,RBR_out,eB,clock,reset_);
 input clock,reset_; wire clock_RSS; assign #5 clock_RSS=clock;
 input dav_,eB;
 output rfd,fi;
 input[7:0] byte_in;
 output[7:0] RBR_out;

 reg RFD; assign rfd=RFD;
 reg FI; assign fi=FI;
 reg[7:0] RBR; assign RBR_out=RBR;

 reg[1:0] STAR; parameter S0=0,S1=1,S2=2,S3=3;
 always @(reset_==0) #1 begin RFD<=1; FI<=0; STAR<=S0; end
 always @(posedge clock_RSS) if (reset_==1) #3
   casex(STAR)
   //Handshake con il dispositivo esterno con immissione del
   //nuovo byte in RBR
   S0: begin RFD<=1; RBR<=byte_in; STAR<=(dav_==1)?S0:S1; end
   S1: begin RFD<=0; STAR<=(dav_==0)?S1:S2; end

   //Messa a 1 del contenuto di FI ed attesa che il processore
   //inizi la fase di esecuzione dell'istruzione IN RBR_offset,AL;
   //messa a 0 del contenuto di FI e passaggio allo stato interno
   //successivo non appena tale fase ha inizio
   S2: begin FI<=(eB==0)?1:0; STAR<=(eB==0)?S2:S3; end
```

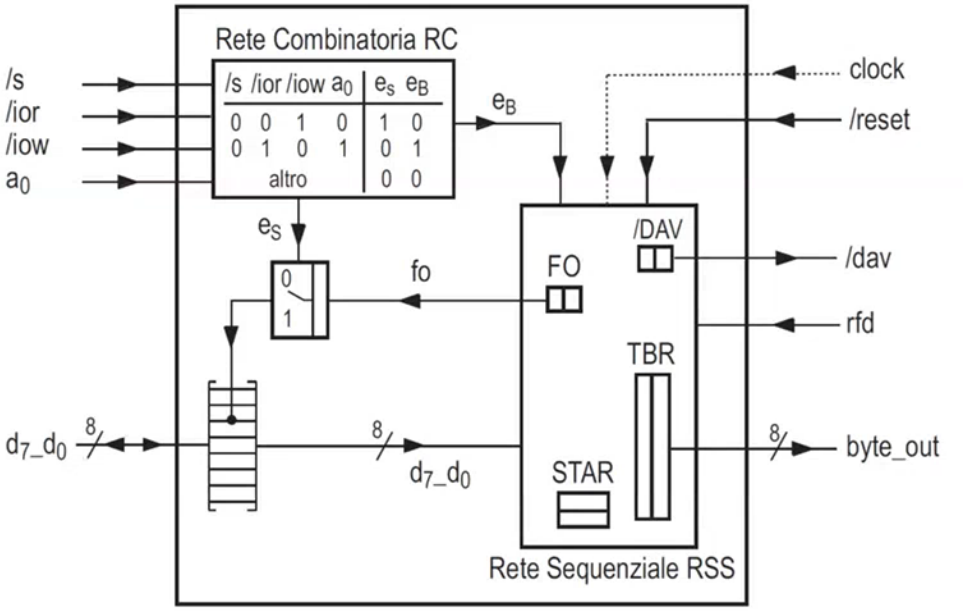
Attenzione al clock. Stea nell'immagine aggiorna il flag con un clock di ritardo rispetto alla variazione di e_B. Se uno vuole fare il precisino può impostare l'aggiornamento del registro FI in questo modo e in S2.

```
   //Ritorno allo stato interno iniziale quando il processore
   //termina la fase di esecuzione dell'istruzione IN RBR_offset,AL
   S3: begin STAR<=(eB==1)?S3:S0; end
 endcase
endmodule
```

  - Attenzione al filo del clock interno, che è il filo del clock del bus leggermente ritardato. Questa cosa non è rilevante, ma utile per capire se abbiamo lavorato correttamente.

Interfaccia parallela di uscita con handshake

- L'interfaccia presenta due registri: il registro TSR, che presenta il flag di uscita vuoto FO (in posizione 5), e il registro TBR, come buffer per il contenuto che sarà letto dal dispositivo.
- Contrariamente a prima abbiamo sia il comando di lettura /ior che quello di scrittura /iow. Con la rete vogliamo svolgere:
  - Operazioni di lettura sul flag FO, per verificare se il dispositivo ha indotto la modifica del flag leggendo il contenuto.
  - Operazioni di scrittura sul registro TBR, per trasmettere al dispositivo un nuovo byte.
- Abbiamo un meccanismo di handshake, ma con ruoli invertiti rispetto a prima: l'interfaccia è il produttore, il dispositivo il consumatore.



- Abbiamo una rete combinatoria che produce l'ingresso di abilitazione per una porta tristate, e l'uscita e_b. e_s è uguale a 1 se vogliamo svolgere una lettura del flag FO, zero altrimenti. e_b non è più utilizzato per una porta tristate, ma rimane per permettere alla RSS di capire se il processore ha eseguito un'operazione di lettura sul registro TBR.

Per quanto riguarda la disposizione delle porte tristate:

- A differenza di prima abbiamo una sola porta tristate: essa è in conduzione con $e_s = 1$, cioè se vogliamo svolgere un'operazione di lettura sul registro TSR.
  - Se la porta è in conduzione non avremo valori in ingresso coi fili di dati, ma avremo l'utilizzo del filo in posizione 5 come unico filo di uscita.

Descrizione:

```
module RSS(dav_,rfd,byte_out,fo,d7_d0,eB,clock,reset_);
 input clock,reset_; wire clock_RSS; assign #5 clock_RSS=clock;
 input rfd,eB;
 output dav_,fo;
 output[7:0] byte_out;
 input[7:0] d7_d0;

 reg DAV_; assign dav_=DAV_;
 reg FO; assign fo=FO;
 reg[7:0] TBR; assign byte_out=TBR;
 reg[1:0] STAR; parameter S0=0,S1=1,S2=2,S3=3;

 always @(reset_==0) #1 begin DAV_<=1; FO<=1; STAR<=S0; end
 always @(posedge clock_RSS) if (reset_==1) #3
   casex(STAR)
     //Messa a 1 del contenuto di FO ed attesa che il processore
     //inizi la fase di esecuzione dell'istruzione OUT AL,TBR_offset;
     //messa a 0 del contenuto di FO, immissione finale in TBR del byte
```

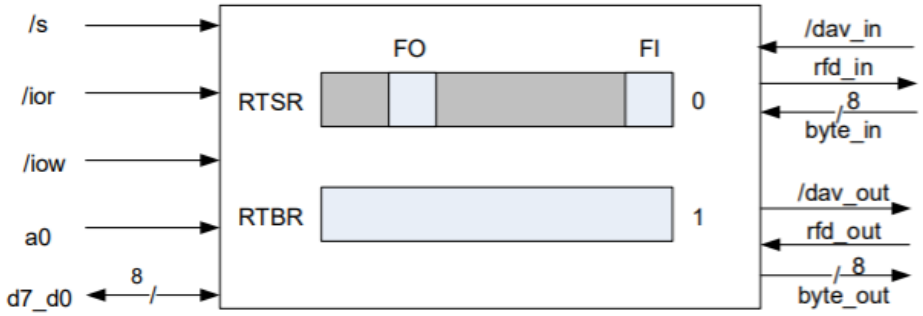
```
      //inviato dal processore tramite d7_d0 e passaggio allo stato
      //interno successivo, il tutto non appena tale fase ha inizio
      S0: begin FO<=(eB==0)?1:0; TBR<=d7_d0; STAR<=(eB==0)?S0:S1; end

      //Attesa che il processore termini la fase di esecuzione della
      //istruzione OUT AL,TBR_offset
      S1: begin STAR<=(eB==1)?S1:S2; end

      //Handshake con il dispositivo esterno con invio del byte contenuto
      //in TBR e conseguente ritorno allo stato interno iniziale
      S2: begin DAV_<=0; STAR<=(rfd==1)?S2:S3; end

      S3: begin DAV_<=1; STAR<=(rfd==0)?S3:S0; end
   endcase
endmodule
```

- **S0**: Sono in uno stato di attesa. Attendo che il processore esegua operazioni di scrittura sul TBR (me ne accorgo con $e_b = 1$). Aggiorno fin da subito il buffer per avere il contenuto già pronto in S1.
- **S1**: Attendo che l'operazione di scrittura venga conclusa (me ne accorgo con $e_b = 0$).
- **S2**: Pongo il flag di uscita vuota a zero poco prima di passare in S3. Abbasso /dav. Attendo che rfd venga posto a zero prima di passare in S3 (gestione dell'handshake, ricordare le regole).
- **S3**: Alzo /dav, e ritorno in S0 non appena avrò rfd a uno. A quel punto sono certo che il dispositivo esterno abbia concluso l'elaborazione del dato.

Interfaccia parallela di ingresso-uscita

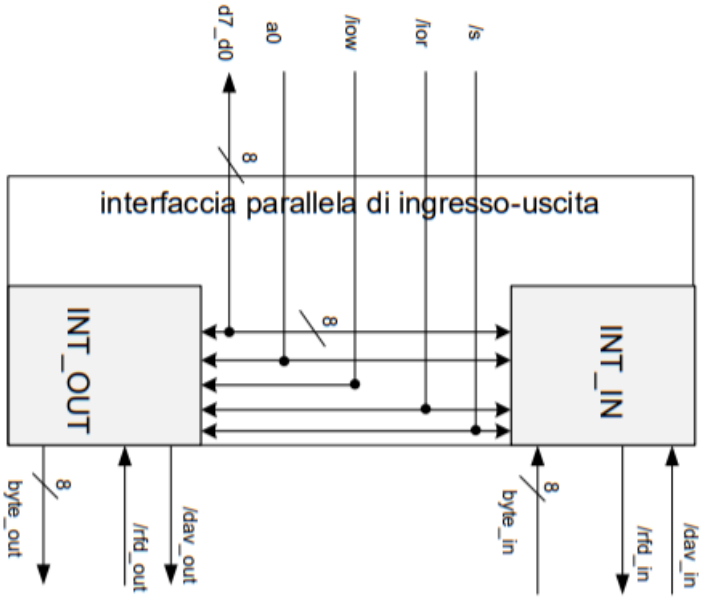
Le due interfacce con handshake appena introdotte possono essere connesse in un'unica interfaccia parallela con handshake d ingresso-uscita. Si consideri che:

- I registri RBR e TBR sono mappati su un unico indirizzo interno (1). Agiremo su un registro o su un altro in base all'operazione indicata con gli attivi bassi.



- Possiamo svolgere:
  - Operazioni di lettura sul registro RTSR (unione dei due registri già presentati coi flag). Il flag di uscita vuota è in posizione 5, mentre il flag di ingresso pieno in posizione 0.
  - Operazioni di lettura su RBR.
  - Operazioni di scrittura su TBR.

Osservazione: ci sono problemi se /s=0, /ior=0, a_0=0? No perché vanno in conduzione, in entrambi i moduli, le tristate relative ai flag, e questi vengono posizionati in posti differenti (posizione 5 e 0).



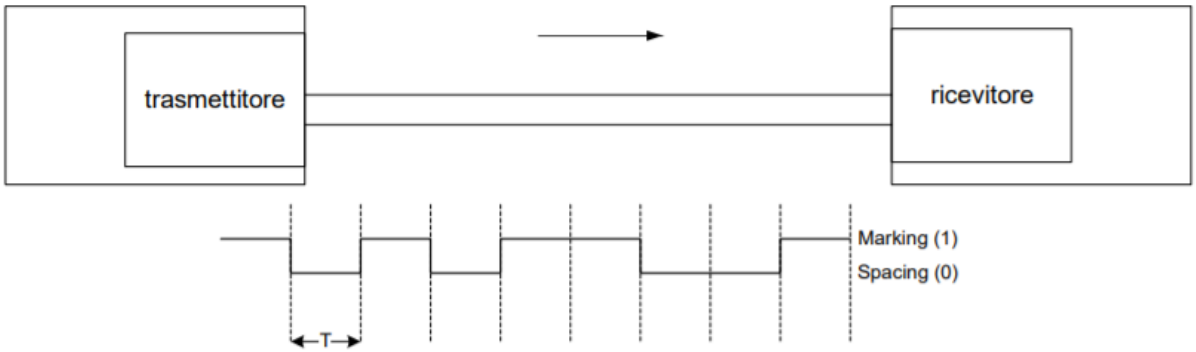
Interfacce seriali

- Un'interfaccia seriale è un'interfaccia dove la trasmissione dei singoli bit avviene in modo seriale. Con "in modo seriale" si intende la trasmissione di un bit alla volta, dal meno significativo, al più significativo. L'interfaccia:
  - riceve byte dal bus e trasmette all'esterno sequenze di bit (serializzazione);
  - riceve dall'esterno sequenze di bit e presenta byte al processore componendo quelle sequenze di bit in un registro (de-serializzazione).
- Le interfacce viste fino ad ora sono, teoricamente, seriali, in quanto permettono di trasmettere una serie di byte. La differenza sta nel fatto che qua serializziamo il singolo byte.

- Un PC, di norma (oggi sempre meno), presenta almeno una porta seriale. La cosa è utilizzata per modem e un tempo pure per i mouse (pensiamo ai mouse di una volta, quelli con la pallina, che presentano un connettore non-USB).
- Perché esistono le interfacce seriali: un tempo i calcolatori consistevano in grossi elaboratori (cervello) che venivano connessi a terminali stupidi (mani). La connessione tra terminale stupido ed elaboratore intelligente avveniva, appunto, mediante linee seriali.

Comunicazione seriale

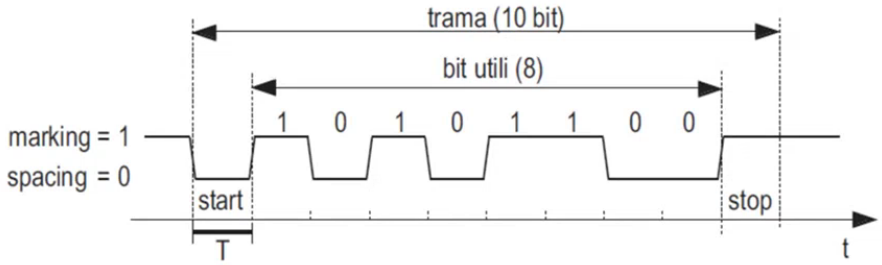
- Distinguiamo due tipi di comunicazione:
  - ***full-duplex***: il mezzo trasmissivo sostiene trasmissione in entrambe le direzioni contemporaneamente
  - ***half-duplex***: il mezzo trasmissivo indica la trasmissione da un solo lato.
- Nello spiegare la comunicazione seriale ci limiteremo, per questioni di semplificazione, a osservare la comunicazione *half-duplex*.



- Il mezzo trasmissivo consiste in un insieme di due linee:
  1. Una linea di riferimento, detta linea di massa
  2. Una linea che porta una tensione riferita a massa. Su questa linea sono leciti due valori:
     - *marking*, cioè 1 logico;
     - *spacing*, cioè 0 logico.
- La trasmissione di un bit consiste nel tenere la seconda linea in uno stato di *marking* o *spacing* per un determinato tempo T, detto **tempo di bit**.
- Un insieme di bit scambiati costituisce una trama, detta in inglese frame.

- **Problema fondamentale**: abbiamo detto che la comunicazione si ha mediante due fili, allora come possiamo sincronizzare trasmettitore e ricevitore? Chiaramente non potremo condividere un clock (significa avere un filo in più), né aggiungere delle linee dedicate all'*handshake* (due fili in più).
- **Soluzione al problema fondamentale**:
  - Le due parti devono essere concordi sul tempo di bit e sul numero di bit che caratterizza ogni trama (tipicamente si ha da 5 a 8 bit per trama);
  - Le trame devono essere rese riconoscibili, cioè abbiamo bisogno di elementi che permettono di capire quando una trama inizia e quando finisce. Abbiamo stabilito che:
    - La linea sta, normalmente, in uno stato di marking;
    - Per iniziare una nuova trama poniamo la linea in uno stato di spacing. Questo significa che ogni trama presenterà un bit iniziale, non informativo del contenuto della trama, detto bit di start.

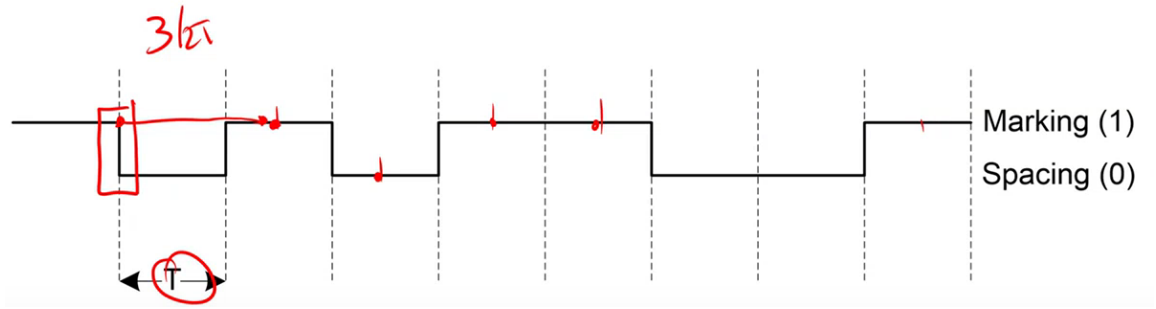
- Per concludere la trama poniamo la linea in uno stato di marking. Abbiamo quindi un bit finale, non informativo del contenuto della trama, detto bit di stop.



Esempio di trama. Il trasmettitore fa il primo passo ponendo il bit di start. Sia il trasmettitore che il ricevitore conoscono il tempo di bit.

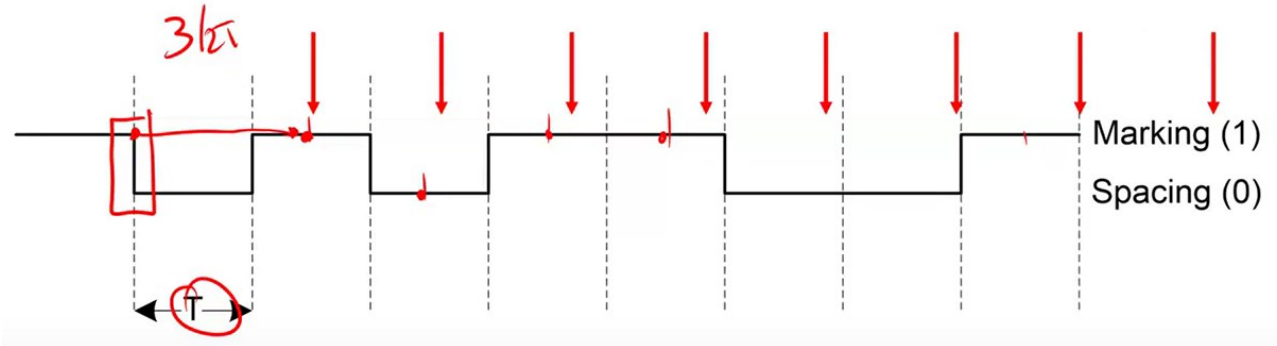
- Il bit di start ed il bit di stop, ribadiamo, sono bit di controllo:
  - Una trama di n bit utili è lunga almeno $n + 2$ bit (include il bit di start e il bit di stop);
  - La velocità di trasmissione netta è pari a $\frac{n}{n+2}$ della velocità della linea. La conseguenza per un ingegnere, che punta all'efficienza, è di fare trame con un numero di bit elevato.
- **Ma allora perché abbiamo detto che le trame sono costituite, in media, da 5/8 bit?** Il clock di trasmettitore e ricevitore non sono identici: possono essere simili, ma non uguali. Se le trame sono lunghe gli errori si accumulano e i bit della trama si disallineano.

- **Cosa avviene nella teoria?**



  - Devo individuare il fronte di discesa nel modo più preciso possibile.
  - Attendo 3/2 del tempo di bit.
  - Effettuo il primo campionamento e attendo, da adesso, un tempo di bit ogni volta.

- **Cosa avviene in realtà?**



  - Supponiamo che il clock del ricevitore sia un po' più lento: a causa di fenomeni fisici e della discrepanza dei clock si accumulano ritardi e nel giro di poco esco dal bit, quindi campiono in modo sbagliato.

- **Quale compromesso posso adottare?** Se il clock del ricevitore misura un tempo $T + \Delta T$ (sommo tempo di bit e ritardo), per poter decodificare n bit è necessario che

$$n \cdot \Delta T \leq \frac{T}{2}$$

cioè che la somma dei ritardi non superi la metà del tempo di bit. Se ciò avviene si esce dal bit e si campiona in modo sbagliato. Segue che

$$\frac{\Delta T}{T} \leq \frac{1}{2 \cdot n}$$

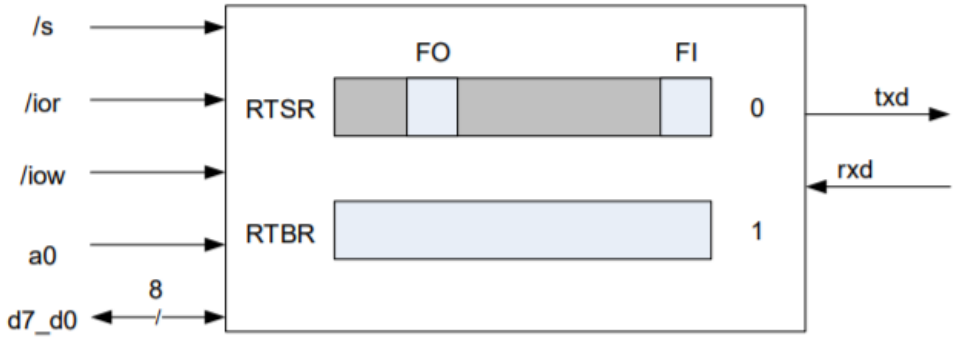
cioè l'errore relativo tollerabile è inversamente proporzionale al numero di bit che devono essere trasmessi tra due segnali di sincronizzazione.

- **Standard EIA-RS2E2C**: standard sviluppato negli anni 60 per gestire la comunicazione seriale. Si occupa di varie questioni: temporizzazione, funzione dei segnali, connettori, formato delle trame, protocollo di comunicazione, **voltaggi elettrici**…
- Quest'ultima cosa è quella che ci interessa maggiormente: si osservi che le tensioni indicate dallo standard sono diverse da quelle standard utilizzate nelle reti logiche.
    - 0 logico: tensione positiva [+3; +25]V
    - 1 logico: tensione negativa [-25; -3]V

  Si è deciso di adottare queste tensioni per ragione di simmetria: si vuole fare in modo che il valor medio della tensione non sia significativamente diverso da zero (in caso contrario si accumula corrente).

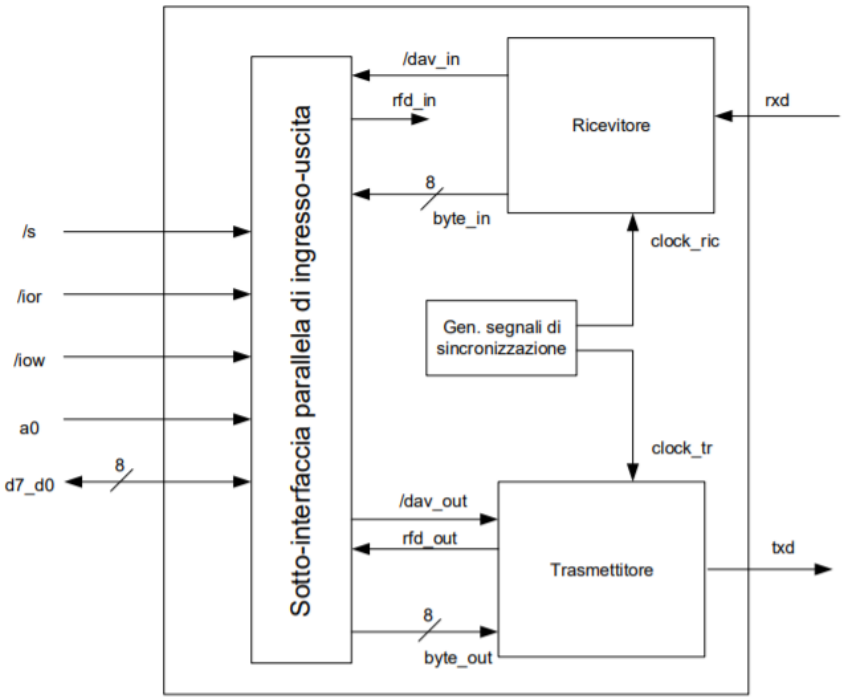
Visione funzionale e struttura interna dell'interfaccia

- La struttura è molto simile all'interfaccia parallela con handshake ingresso/uscita:



Abbiamo un registro di stato RTSR, in cui il bit 5 ed il bit 0 sono rispettivamente il flag di uscita vuota FO e di ingresso pieno FI, e il registro RTBR ad 8 bit che serve per contenere i dati da trasmettere a quelli ricevuti.-

- Spogliamo ulteriormente l'interfaccia

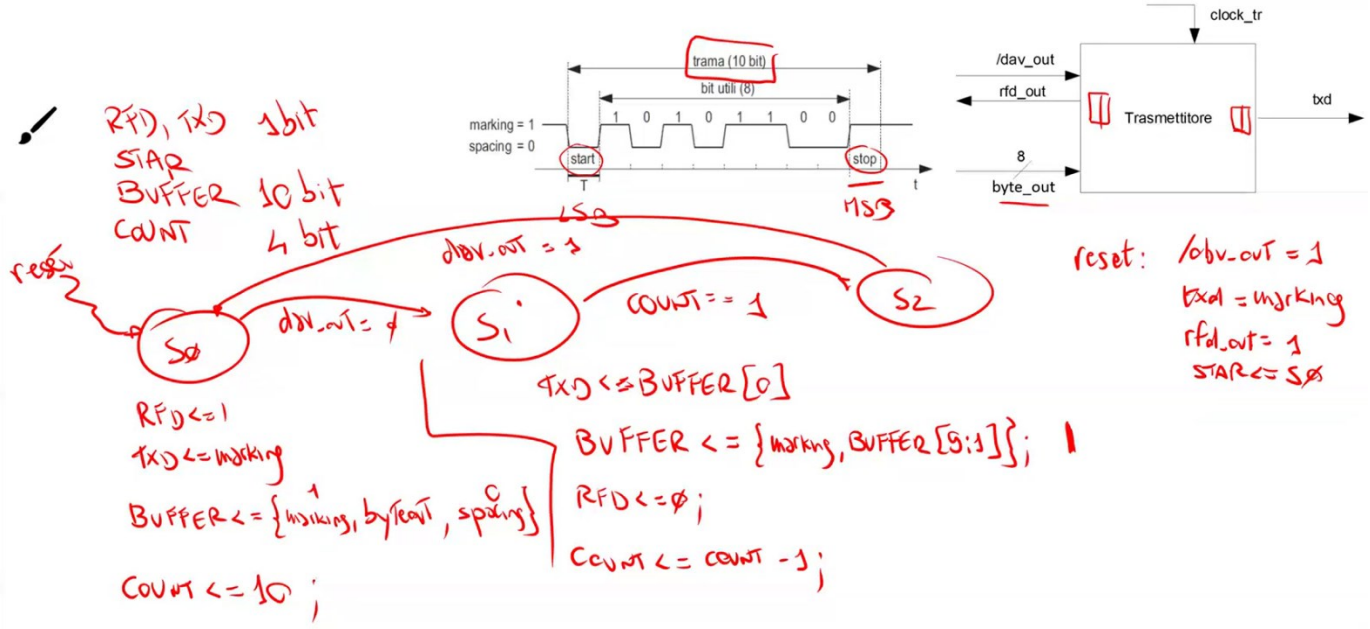


Individuiamo una RSS Ricevitore e una RSS Trasmettitore. Si osservi che:

- Il singolo ricevitore o il singolo trasmettitore non costituiscono da soli l'interfaccia seriale. È necessaria una sotto-interfaccia parallela di ingresso-uscita.
- Il ricevitore è un produttore (dal punto di vista della sotto-interfaccia): riceve dal dispositivo esterno, ma produce un byte per la sotto-interfaccia.
- Il trasmettitore è un consumatore (dal punto di vista della sotto-interfaccia): riceve un byte dalla sotto-interfaccia, ma produce una trama per il dispositivo esterno.

- Inoltre:
  - Le due RSS presentano clock diverso (necessario, come capiremo poco più avanti).
  - La variabile rfd_in che non viene letta dal ricevitore. Questo perché il ricevitore non ha modi per comunicare col dispositivo esterno e indicare che il processore ha utilizzato i dati ed è pronto per riceverne altri.

Descrizione del trasmettitore



- L'interfaccia:
  - Accetta un nuovo byte dalla sotto-interfaccia parallela di uscita, con la quale ha un *handshake*;
  - Trasmette tutti i bit di quel byte sul mezzo trasmissivo tramite il filo *txd*.

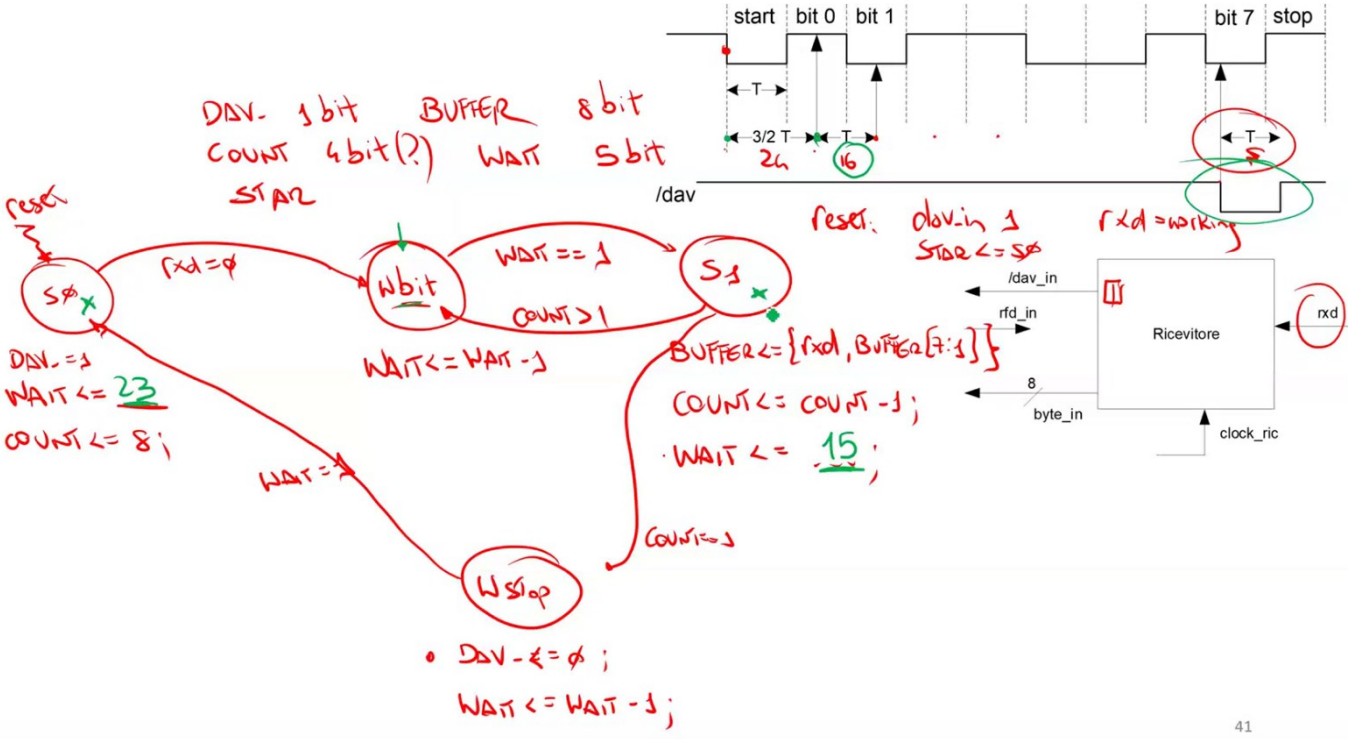
- **Di quali registri abbiamo bisogno?**
  - Registro di stato STAR
  - Registri a supporto delle uscite: RFD, TXD (ciascuno di 1bit)
  - Registro BUFFER dove memorizzare l'intera sequenza di bit (10 bit, incluso il bit di start e il bit di fine)
  - Registro COUNT, quattro bit (10) necessari per contare il numero di bit da trasmettere.
- **Ipotesi al reset**:
  - /dav_out e rfd_out presentano i valori tipici dell'handshake al momento del reset (entrambi 1)
  - txd è uguale ad uno, quindi è in uno stato di marking
  - STAR è uguale ad S0, come al solito si pone come valore iniziale quello del primo valore interno.
- **Ipotesi sul clock**: un clock equivale al tempo di bit T.

```
module Trasmettitore (dav_out_, rfd_out, byte_out, txd, clock, reset_);
  input clock, reset_;
  input dav_out_;
  input [7:0] byte_out;
  output rfd_out, txd;

  reg [3:0] COUNT;
  reg [9:0] BUFFER;
  reg RFD, TXD; assign rfd_out=RFD; assign txd=TXD;

  reg [1:0] STAR; parameter S0=0, S1=1, S2=2;
  parameter mark=1'B1, start_bit=1'B0, stop_bit=1'B1;

  always @(reset_==0) #1 begin RFD<=1; TXD<=mark; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex (STAR)
      S0: begin RFD<=1; COUNT<=10; TXD<=mark;
                BUFFER<={stop_bit,byte_out,start_bit};
```

```
                STAR<=(dav_out==1)?S0:S1; end
      S1: begin RFD<=0; TXD<=BUFFER[0]; BUFFER<={mark, BUFFER[9:1]};
                COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
      S2: begin STAR<=(dav_out==0)?S2:S0; end
    endcase
endmodule
```

- **Spiegazione degli stati**:
  - **S0**: pongo RFD uguale ad 1, così come TXD in marking. Per ragioni di semplificazione e riduzione di stati (abbiamo già visto questo in passato) aggiorniamo il valore dei registri BUFFER e COUNT direttamente in S0. Attenzione al valore di BUFFER (includo nei bit anche il bit di star e quello di fine). Passo allo stato successivo quando /dav_out viene abbassato (cioè quando si hanno nuovi dati da trasmettere)
  - **S1**: gestisco la trasmissione dei bit mediante una serie di shift destri. Purtroppo la lettura in array del tipo BUFFER[COUNT] non siamo in grado di sintetizzarla. Esco quando avrò COUNT uguale ad 1, cioè quando avrò restituito tutti i bit attraverso il filo txd (a quel punto il buffer avrà tutti bit uguali ad 1).
  - **S2**: non posso tornare subito in S0, visto che devo finire di gestire la temporizzazione dell'handshake. Attendo che /dav_out ritorni ad 1 e solo a quel punto abbiamo finito.

Descrizione del ricevitore



- Il ricevitore:
  - Non è collegato al filo rfd_in. Come già anticipato il ricevitore non è in grado di controllare il flusso dei dati in ingresso: segue che non ha nessun senso gestire un handshake completo. Se la sottointerfaccia parallela non è in grado di accettare un nuovo dato è problema suo: il dato verrà sovrascritto da una nuova trama di bit.
  - Il ricevitore capta l'inizio di una trama col fronte di discesa da marking a spacing. Campiona il primo bit dopo 3/2 del tempo di bit da quando vede l'inizio della trama. Successivamente campiona ciascun successivo bit dopo un tempo di bit.
- **Ipotesi sul clock**:
  - Il clock del ricevitore deve essere diverso da quello del trasmettitore: questo perché dobbiamo riuscire a campionare i bit (per quanto possibile) a metà del tempo di bit. Non è possibile fare ciò con lo stesso clock del trasmettitore.
  - **Come imposto il clock?** Il periodo di clock è la minima finestra temporale su cui guardo gli eventi, il ricevitore deve individuare un certo evento nella maniera più veloce possibile: segue che più il clock è veloce, più saremo precisi. Supporremo che il clock abbia frequenza **16x**, cioè che sia sedici volte più veloce del clock del trasmettitore.

- **Di quali registri abbiamo bisogno?**
  - Registro DAV_ a 1 bit, che supporta l'uscita /dav_in
  - Il registro BUFFER dove la parte di trama ricevuta fino a questo momento. Mi bastano solo 8 bit.
  - Il registro COUNT, con cui tengo conto del numero di bit letti.
  - Il registro WAIT, per contare i clock. Utilizzo questo registro per attendere:
    - 16 cicli di clock tra un bit utile e il successivo, e alla fine (quando finisco nel bit di stop). In questo ultimo caso potrei attendere solo 8 clock, ma considerando le discrepanze fisiche nei due clock non mi conviene.
    - 24 cicli di clock dopo il fronte di discesa del bit di start

- **Ipotesi al reset**:
  - /dav_in è a 1 (come nell'handshake)
  - byte_in non è significativo poiché è protetto dall'handshake
  - rxd è in marking
- **Spiegazione degli stati**:
  - **S0**: in questo stato ho /dav a 1. Passo allo stato successivo quando rxd va in spacing, e rimango in quello stato per un numero di clock pari a 23 (cioè 3/2 del tempo di bit). Pongo WAIT uguale a 23 e non a 24 poiché perdo già un clock in S0 prima di passare allo stato Wbit.
  - **Wbit**: se mi trovo in questo stato significa che devo attendere prima di fare il prossimo campionamento. Abbiamo definito, in S0 e in S1, un valore di WAIT. WAIT viene decrementato e mi sposto (o ritorno) in S1 quando ho WAIT uguale ad 1.
  - **S1**: imposto il valore di BUFFER. Devo fare in modo che i primi valori inseriti si trovino sui posti meno significativi: faccio ciò inserendo ogni nuovo bit (preso dal filo rxd) come MSD (shifto sbarazzandomi della LSD). Decremento COUNT, ed ogni volta ritorno in Wbit (tranne quando avrò COUNT == 1). Memorizzo in WAIT 15, e non 16, per la stessa motivazione vista in S0. Quando COUNT sarà uguale ad 1 mi sposto in Wstop.
  - **Wstop**: porto /dav a 0, abbiamo finito. Prima di ritornare in S0 devo attendere 16 clock (il solito tempo di bit). A tale scopo recupero il valore del registro WAIT, impostato in S1. Decremento WAIT, e ritorno in S0 quando avrò WAIT == 1.

```
module Ricevitore (dav_in_, clock, rxd, reset_, byte_in);
  input clock, rxd, reset_;
  output dav_in_;
  output [7:0] byte_in;

  reg DAV_; assign dav_in_=DAV_;
  reg [3:0] COUNT;
  reg [4:0] WAIT;
  reg [7:0] BUFFER; assign byte_in=BUFFER;
  reg [1:0] STAR; parameter S0=0, S1=1, Wbit=2, Wstop=3;
  parameter start_bit=1'B0;

  always @(reset_==0) #1 begin DAV_<=1; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
  casex (STAR)
   S0: begin DAV_<=1; COUNT<=8; WAIT<=23;
             STAR<=(rxd==start_bit)?Wbit:S0; end
   Wbit: begin WAIT<=WAIT-1; STAR<=(WAIT==1)?S1:Wbit; end
   S1: begin COUNT<=COUNT-1; WAIT<=15; BUFFER<={rxd,BUFFER[7:1]};
            STAR<=(COUNT==1)?WStop:Wbit; end
   WStop: begin DAV_<=0; WAIT<=WAIT-1; STAR<=(WAIT==1)?S0:WStop; end
  endcase
endmodule
```