Struttura del Calcolatore

Versione per studenti

Giovanni Stea

Ultima modifica: 27/11/2025

Prerequisiti

Gli studenti hanno appreso:

- Teoria delle reti combinatorie e delle reti sequenziali
- Linguaggio Verilog e microprogrammazione
- Assembler

Version history

24/11/20: prima versione

1/12/20: aggiunta sezione 2.3 su conversione A/D e D/A, corrette imprecisioni rilevate durante le lezioni.

3/12/20: corrette imprecisioni rilevate durante le lezioni

5/12/20: migliorata la sezione 2.3.2 sulla conversione analogico/digitale

8/2/23: corretto bug nel codice Assembler, pag. 60. Aggiunte figure dei convertitori (courtesy of Luca Giannini)

25/7/23: corretto errore nel codice Assembler a pag. 60 (courtesy of Jun Hao Liu)

27/11/23: corretti errori minori nelle figure dei formati F6 ed F7

12/12/23: modifiche cosmetiche

20/05/24: corretto errore minore a pag. 17

28/11/24: aggiunte figure (courtesy of Gabriele Domenico Cambria)

26/06/25: aggiunta appendice con domande a risposta multipla.

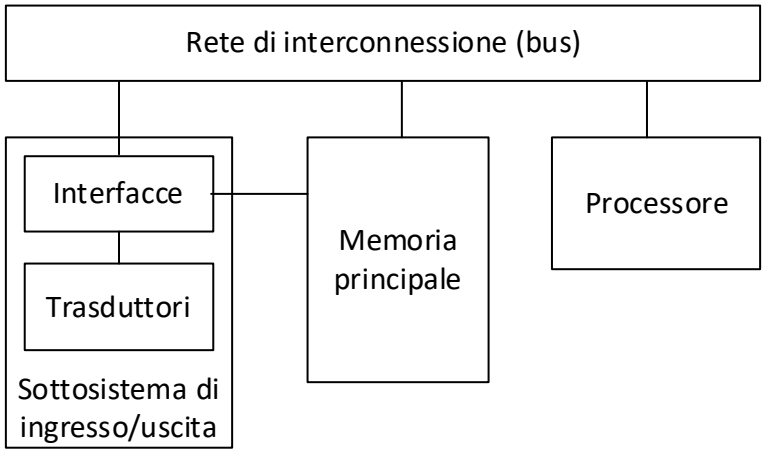
27/06/25: aggiunte figure mancanti nelle sezioni.

27/11/25: modificato il disegno a pag. 47 per chiarire che c'è una sola variabile logica tra trasmettitore e ricevitore seriale.

Sommario

1 Struttura del calcolatore

Scopo del prossimo blocco di lezioni è la descrizione in Verilog di un sistema-calcolatore completo di **processore, memoria, interfacce e dispositivi di ingresso/uscita**. Sarà un calcolatore abbastanza semplice da essere trattabile.



- **Il sottosistema di ingresso/uscita (I/O)** si occupa di gestire la codifica delle informazioni ed il loro scambio con il mondo esterno. A seconda del tipo di dispositivo che abbiamo, tali informazioni **saranno ricavate da** movimento di organi meccanici, impulsi elettrici, suoni, immagini, etc. (ingresso), oppure **diventeranno** movimento di organi meccanici, impulsi elettrici, suoni, immagini (uscita). All'interno di questo sottosistema distinguiamo **interfacce e dispositivi**. Questi ultimi effettuano la vera e propria codifica. Le prime, invece, **gestiscono** i vari dispositivi, cioè fanno in modo che il colloquio tra questi ed il processore possa avvenire con modalità standard. Nel seguito, vedremo in dettaglio alcuni dispositivi ed interfacce (di tipo didattico). Per adesso, ci limitiamo ad osservare che le interfacce conterranno un numero **(piccolo) di registri di interfaccia**, che il processore può leggere o scrivere (o, più raramente, leggere e scrivere).
- La **memoria principale** contiene in ogni istante **le istruzioni e i dati** che il processore elabora (alcuni dati possono risiedere nel sottosistema di I/O). Una parte di questa memoria è adibita a **memoria video**, e contiene una replica dell'immagine che viene mostrata sullo schermo. Per questo motivo, c'è un collegamento diretto tra la memoria ed un'interfaccia (video) nella figura.
- Il **processore** ciclicamente **preleva un'istruzione dalla memoria (fetch o chiamata) e la esegue**, fin quando non trova un'istruzione particolare (HLT) che lo blocca. Le istruzioni che esegue si trovano, di norma, **in sequenza**, cioè in locazioni contigue, nella memoria principale (**istruzioni operative**). A volte, nell'eseguire una particolare istruzione (**istruzioni di controllo**), il flusso sequenziale viene alterato, ed il prelievo di istruzioni riparte da una locazione diversa. Va osservato che il processore, per poter agire in questo modo, deve poter partire **al reset** in modo consistente. Ciò significa che

- Deve iniziare a leggere la memoria **da una locazione ben precisa;**
- In quella locazione ci deve essere **già scritto del codice, in maniera indelebile.**

Ciò si realizza facendo in modo che:

- Al reset, il processore abbia inizializzato l'**instruction pointer** (ed altro che vedremo)
- Parte della memoria sia implementata con tecnologia EPROM (Visto che la memoria RAM è **volatile**), e contenga cablato al suo interno un programma **bootstrap** che viene eseguito alla partenza del calcolatore.

Introdurremo un processore d'esempio, detto **sEP8 (8-bit simple Educational Processor)**. Tale processore è in grado di elaborare dati a 8 bit e lavora in aritmetica in base 2, rappresentando gli interi in C2. È in grado di indirizzare una memoria di **16Mbyte**.

- **La rete di interconnessione** (bus) mette in comunicazione tutti questi moduli, trasportando i segnali generati da uno verso l'altro.

Il calcolatore è una serie di RSS. Tali sono il processore e la maggior parte delle interfacce che introdurremo (la memoria RAM, abbiamo visto, è invece una RSA). Per questo, possiamo supporre che tutti i moduli di tipo RSS siano dotati di un piedino /***reset***, che fa sì che partano tutti assieme in modo coerente.

Lo scopo di questa parte di corso è arrivare ad una **descrizione in Verilog del processore**, come RSS (che poi potremmo sintetizzare in accordo al modello con scomposizione PO/PC). Ciò è particolarmente importante dal punto di vista concettuale, perché vi dà modo di osservare – dal punto di vista della struttura hardware – una particolare rete logica, che è in grado di eseguire del software (programmi scritti in linguaggio macchina).

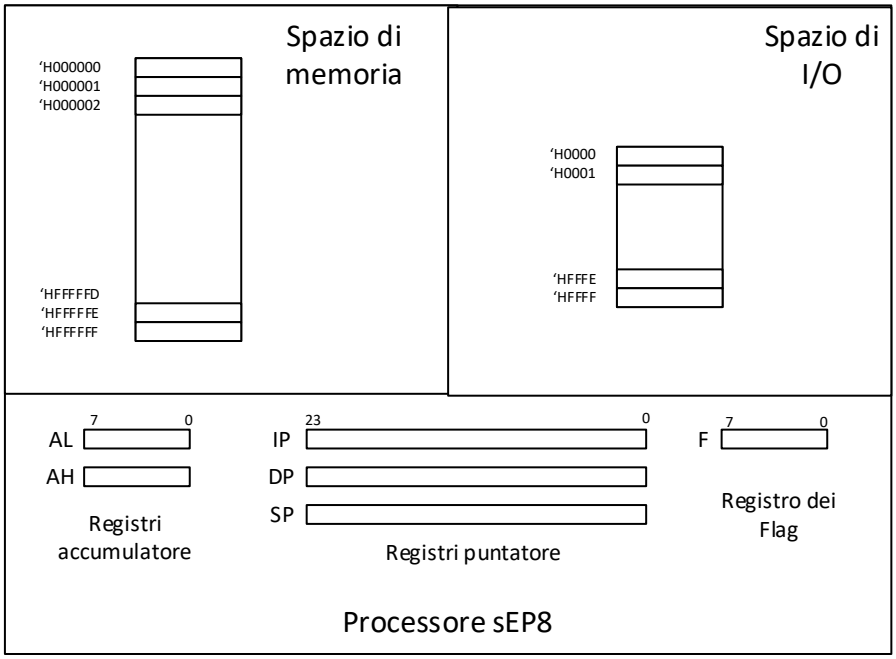
Per procedere alla descrizione in Verilog del processore, dobbiamo prima **darne una specifica**, così come facciamo con qualunque RSS. Dare la specifica comporta dire:

a) Con quali altre reti si interfaccia, e come;
b) Quale è il suo comportamento osservabile.

Le prossime ore sono dedicate a questo scopo.

1.1 Visione del calcolatore da parte del programmatore

Descriviamo adesso ciò che un programmatore vede del sistema calcolatore. Ciò che il programmatore vede è riassunto nella figura:



La memoria appare al programmatore come uno spazio lineare di 2^{24} locazioni da un byte ciascuna, per un totale di 16 Mbyte. Per indirizzarlo, è necessario specificare un indirizzo a 24 bit.

Lo **spazio di I/O**, cioè l'insieme dei **registri di interfaccia** che il processore può teoricamente indirizzare **appare al programmatore** come uno spazio **lineare da 2^{16}=64K locazioni o porte**. Per indirizzare una di queste porte, il processore dovrà specificarne l'offset all'interno dello spazio di I/O. Non necessariamente ad ogni locazione dello spazio di I/O corrisponderà un **registro di interfaccia**. Anzi, è molto probabile che la maggior parte di questo spazio di indirizzamento non abbia una controparte fisica. Le interfacce sono poche (qualche decina), e ciascuna di esse ha pochi registri (nell'ordine delle unità). In questo spazio, il processore potrà **leggere un byte** da una porta (ad esempio, prelevando un dato prodotto da un'interfaccia di ingresso, o per informarsi sullo **stato** di tale interfaccia), o **scrivere un byte** (per farlo uscire attraverso un'interfaccia di uscita o per **configurare** tale interfaccia nel modo voluto).

Il processore sEP8 ha **tre tipi di registri:**

- **Registri accumulatore**: quelli destinati a contenere operandi di elaborazioni. Sono due, AH ed AL, entrambi di 8 bit.
- **Registro dei flag**: sarà ad 8 bit, e di questi saranno significativi per noi **4 bit**: CF (0), ZF (1), SF (2), OF (3).
- **Registri puntatore**: sono tre, e devono poter contenere indirizzi di memoria. Per questo saranno a 24 bit.
  - IP (instruction pointer): contiene l'indirizzo della **prossima** istruzione da eseguire;
  - SP (stack pointer): contiene l'indirizzo del **top della pila;**

- DP (data pointer): contiene l'indirizzo di operandi, a seconda della modalità di indirizzamento (che vedremo più avanti).

Affinché il processore parta in stato consistente al reset, è necessario inizializzare **IP e F**. Mentre F verrà inizializzato a **zero**, IP verrà inizializzato a 'HFF0000. Ciò significa che **a partire da quella locazione si deve trovare il programma di bootstrap**, e che la porzione di memoria che parte da quella locazione deve essere non volatile (e.g., EPROM).

1.2 Descrizione del linguaggio macchina del processore sEP8

Il linguaggio machina di un processore è, di fatto, il suo comportamento osservabile. Noi esseri umani programmiamo però in **Assembler**, **e non in linguaggio macchina**. Quindi conviene iniziare la descrizione del comportamento del processore sEP8 spiegando come un programmatore Assembler dovrebbe scrivere le sue istruzioni, e successivamente discutere come queste si possano codificare in linguaggio macchina.

Per un programmatore Assembler, il formato delle istruzioni del processore sEP8 sarà il seguente:

```
OPCODE source, destination
```

In cui `OPCODE` è il codice operativo dell'istruzione, mentre *source* e *destination* individuano, secondo le modalità di indirizzamento consentite dal linguaggio macchina, i due operandi sorgente e destinatario. In alcune istruzioni il campo *source* può mancare. In due istruzioni (`NOP` e `HLT`) mancano entrambi. Le **modalità di indirizzamento** sono quelle che conosciamo.

Per le **istruzioni operative**:

- **indirizzamento di registro**: uno o entrambi gli operandi sono nomi di registro:

  ```
  OPCODE AL, AH
  OPCODE DP
  ```

- **indirizzamento immediato**: l'operando *sorgente* è specificato direttamente nell'istruzione come costante:

  ```
  OPCODE $0x10, AL
  ```

- **indirizzamento di memoria**: valido per il *sorgente* o per il *destinatario* (mai per entrambi contemporaneamente). Sono possibili due indirizzamenti di memoria**:**
  - **diretto:** l'indirizzo è specificato direttamente nell'istruzione.

    ```
    OPCODE 0x1010, AL
    ```

  - **indiretto**: la locazione di memoria ha indirizzo contenuto nel registro `DP`.

    ```
    OPCODE (DP), AL
    ```

- **indirizzamento delle porte di I/O:** le porte di I/O si indirizzano in modo **diretto**, specificando l'offset della porta dentro l'istruzione stessa:

  ```
  IN 0x1010, AL
  ```

```
OUT AL, 0x9F10
```

Le **istruzioni di controllo** sono invece quelle che alterano il flusso dell'esecuzione del programma, che normalmente procederebbe in sequenza. Le istruzioni di controllo sono **salti, condizionati e non**, **chiamate di sottoprogramma ed istruzioni di ritorno da sottoprogramma.**

Le istruzioni di controllo sono, quindi:

```
JMP  indirizzo
Jcon indirizzo
CALL indirizzo
RET
```

Le prime tre istruzioni devono specificare l'indirizzo a cui si salta, che va a **sostituire il contenuto di `IP`**. Ci ricordiamo che le istruzioni di `CALL` e `RET` interagiscono con la **pila**:

- la `CALL` salva in pila il contenuto di `IP` (3 byte), cioè l'indirizzo della istruzione successiva alla `CALL` medesima (indirizzo di ritorno);
- la `RET` preleva dalla pila un indirizzo (3 byte), e lo sostituisce ad `IP`.

Come si può vedere, l'Assembler dell'sEP8 è scarno, ma abbastanza vicino a quello dei processori Intel visto all'inizio del corso. Abbiamo visto che un processore deve tradurre un'istruzione Assembler:

```
OPCODE source, destination
```

In una sequenza di zeri e uni con una certa sintassi. Questa sintassi costituisce il **linguaggio macchina** di quel processore, e deve essere **compatta** e facile da interpretare (per un processore, non necessariamente per noi).

Per gli esseri umani è dirimente il "tipo" di operazione (ad esempio, `MOV`). Una `MOV` è una copia di informazione, e che gli operandi siano registri o locazioni di memoria non fa una grande differenza. Pertanto, il linguaggio Assembler, che è concepito per essere capito dagli umani, specifica come prima informazione il "tipo" dell'operazione, e successivamente gli operandi. Per un processore, invece, è dirimente **dove si trovino gli operandi**. Facciamo un esempio:

```
MOV AH, AL
MOV $0x10, AL
MOV (DP), AL
```

- Nel primo caso, il processore gli operandi li ha già, perché sono contenuti nei registri.
- Nel secondo caso, invece, il processore deve **leggere in memoria** l'operando sorgente, che è contenuto nell'istruzione medesima (l'istruzione si trova in memoria, ovviamente).
- Nel terzo caso, infine, il processore dovrà **ancora leggere in memoria**, per procurarsi l'operando sorgente, ma l'indirizzo a cui deve leggere è contenuto in `DP`.

Una volta che il processore si è procurato l'operando sorgente, la **fase di esecuzione** delle tre operazioni sarà identica (i.e., metti qualcosa dentro `AL`). La **fase di fetch**, nella quale il processore si procura gli operandi, dovrà invece essere differente.

Ciascuna istruzione macchina è lunga **almeno un byte**. Il **primo** byte di ogni istruzione codifica:

a) Il **tipo** di operazione (quello che noi umani troviamo in **opcode** in Assembler), che è rilevante in **fase di esecuzione**;
b) Il modo in cui si devono recuperare gli operandi, detto **formato** dell'istruzione, che è invece rilevante in **fase di fetch**.

Per questo motivo, le istruzioni del linguaggio macchina vanno divise non tanto per tipo di operazione, ma per **formato** della medesima. Infatti, la fase di fetch è la prima che si deve affrontare, ed è quella in cui si recuperano gli operandi. La fase di esecuzione comincia soltanto quando il processore:

- Ha capito quale operazione deve effettuare
- Si è procurato gli operandi su cui effettuarla.

I formati possibili per il nostro processore sono **otto**, il che vuol dire che nel primo byte:

- I primi tre bit codificano il **formato**
- I restanti cinque bit codificano il codice operativo (32 possibili **opcode**).

Analizziamo i formati in dettaglio, per avere una prima idea del perché sono fatti in questo modo. La suddivisione sarà più chiara quando avremo visto la descrizione del processore.

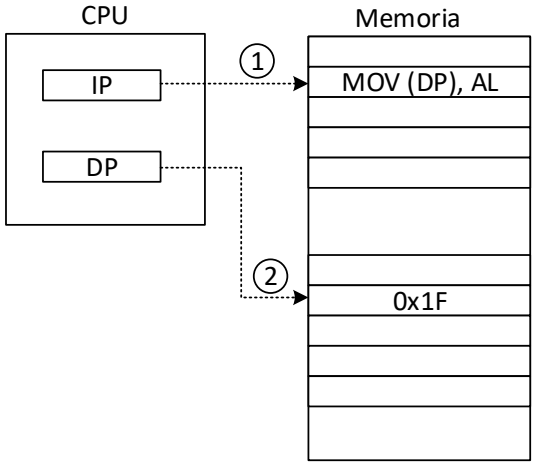
- **Formato F0 (000)**: in questa categoria rientrano tutte le istruzioni per le quali il processore non deve compiere nessuna azione per procurarsi gli operandi, in quanto:
  a) gli operandi sono registri, oppure;
  b) le istruzioni non hanno operandi (`HLT`, `NOP`, `RET`).

  Le istruzioni di questo formato saranno quindi costituite **da un unico byte**. La fase di fetch di un'istruzione di formato F0 si concretizza nella lettura di quest'unico byte, all'indirizzo puntato da `IP`.

```
HLT         |00000000|
NOP         |00000001|
MOV AL,AH   |00000010|
MOV AH,AL   |00000011|
INC DP      |00000100|
SHL AL      |00000101|
SHR AL      |00000110|
NOT AL      |00000111|
SHL AH      |00001000|
SHR AH      |00001001|
NOT AH      |00001010|
PUSH AL     |00001011|
POP AL      |00001100|
PUSH AH     |00001101|
POP AH      |00001110|
PUSH DP     |00001111|
POP DP      |00010000|
RET         |00010001|
```

- **Formato F2 (010)**: raggruppa tutte le istruzioni in cui l'operando sorgente si trova in memoria, indirizzato in modo indiretto tramite il registro puntatore `DP`. Anche in questo caso tutta l'informazione relativa all'istruzione può stare su un singolo byte, ma la fase di fetch di queste istruzioni è ben diversa. Infatti, l'operando sorgente si trova **in memoria**, e va prelevato dalla

memoria. Pensate all'istruzione `MOV (DP), AL`. Per poterla eseguire - cioè, per poter sostituire il contenuto di `AL` - devo prima leggere dalla memoria il valore da scriverci dentro. Questa lettura fa parte della fase di fetch. Pertanto, per tutte le istruzioni di questo formato, la fase di fetch deve prevedere un accesso in lettura in memoria all'indirizzo puntato da `DP`. Questa lettura deve essere di un byte (gli operandi di queste istruzioni sono a 8 bit).

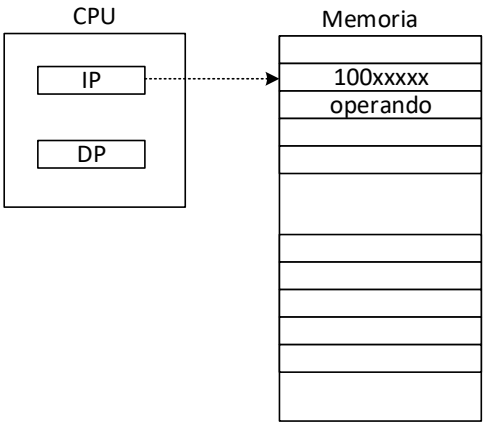



```
MOV (DP),AL     |01000000|
CMP (DP),AL     |01000001|
ADD (DP),AL     |01000010|
SUB (DP),AL     |01000011|
AND (DP),AL     |01000100|
OR  (DP),AL     |01000101|
MOV (DP),AH     |01000110|
CMP (DP),AH     |01000111|
ADD (DP),AH     |01001000|
SUB (DP),AH     |01001001|
AND (DP),AH     |01001010|
OR  (DP),AH     |01001011|
```

- **Formato F3 (011)**: raggruppa le istruzioni in cui l'operando **destinatario** è indirizzato in modo indiretto, usando il registro puntatore `DP`. Anche in questo caso tutta l'informazione relativa all'istruzione può stare su un singolo byte.

```
MOV AL,(DP)  |01100000|
MOV AH,(DP)  |01100001|
```

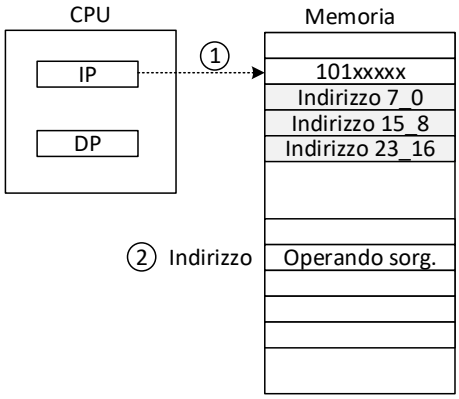
- **Formato F4 (100)**: raggruppa le istruzioni in cui l'operando sorgente è indirizzato in modo **immediato**, e sta su 8 bit. Ciò significa che l'istruzione è lunga **due byte**, e che il secondo byte dell'istruzione contiene l'operando sorgente. Pertanto, la fase di fetch dovrà leggere due byte in memoria, ad indirizzi consecutivi puntati dal registro `IP`.




```
MOV $operando,AL |10000000|operando|
CMP $operando,AL |10000001|operando|
ADD $operando,AL |10000010|operando|
SUB $operando,AL |10000011|operando|
AND $operando,AL |10000100|operando|
OR  $operando,AL |10000101|operando|
MOV $operando,AH |10000110|operando|
CMP $operando,AH |10000111|operando|
ADD $operando,AH |10001000|operando|
[…]
```

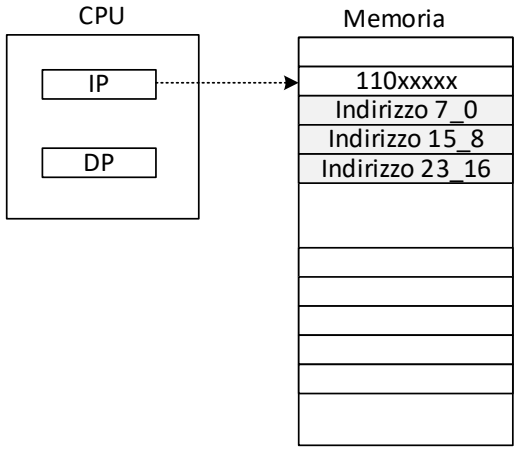
- **Formato F5 (101)**: raggruppa tutte le istruzioni in cui l'operando sorgente è indirizzato in modo **diretto**. Pertanto, le istruzioni saranno lunghe **4 byte**: uno di opcode e tre di indirizzo di memoria (ricordare che lo spazio di memoria è a 24 bit). La fase di fetch dovrà quindi:

- leggere in memoria 4 byte, a indirizzi consecutivi puntati dal registro `IP`
- una volta procuratasi l'indirizzo dell'operando sorgente, andare in memoria a leggere l'operando sorgente stesso



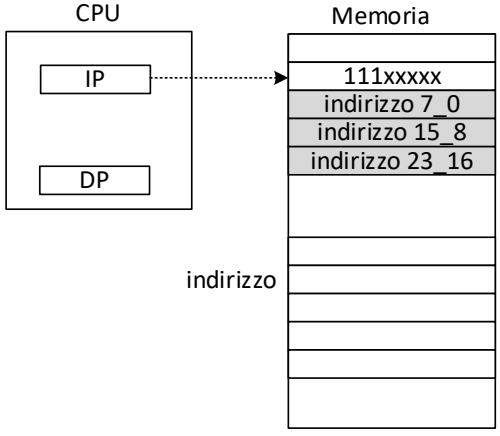

```
MOV indirizzo,AL |10100000| indirizzo |
CMP indirizzo,AL |10100001| indirizzo |
ADD indirizzo,AL |10100010| indirizzo |
SUB indirizzo,AL |10100011| indirizzo |
AND indirizzo,AL |10100100| indirizzo |
OR  indirizzo,AL |10100101| indirizzo |
MOV indirizzo,AH |10100110| indirizzo |
CMP indirizzo,AH |10100111| indirizzo |
[…]
```

- **Formato F6 (110)**: raggruppa tutte le istruzioni in cui l'operando destinatario è in memoria, indirizzato in modo **diretto**. Pertanto, il processore dovrà leggere 4 byte in memoria, per procurarsi l'indirizzo del destinatario, a locazioni consecutive puntate da `IP`.




```
MOV AL,indirizzo |11000000| indirizzo |
MOV AH,indirizzo |11000001| indirizzo |
```

- **Formato F7 (111)**: raggruppa tutte le istruzioni di controllo (`CALL`, `JMP`, `Jcon`) in cui ho un indirizzo di salto, specificato in modo diretto nell'istruzione stessa, su 3 byte. Pertanto, in fase di fetch vanno letti 4 byte consecutivi, puntati dall'indirizzo `IP`.

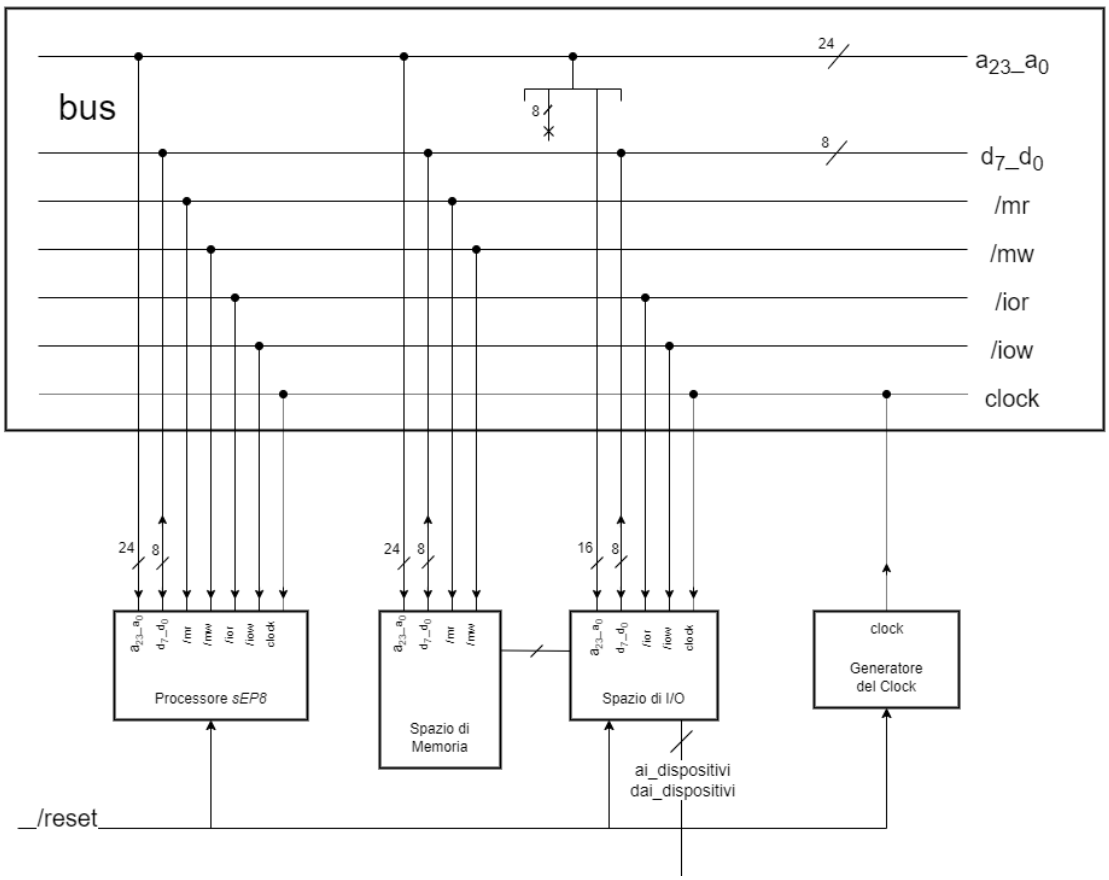



```
JMP  indirizzo  |11100000| indirizzo |
JE   indirizzo  |11100001| indirizzo |
JNE  indirizzo  |11100010| indirizzo |
JA   indirizzo  |11100011| indirizzo |
JAE  indirizzo  |11100100| indirizzo |
JB   indirizzo  |11100101| indirizzo |
[…]
CALL indirizzo  |11110011| indirizzo |
```

- **Formato F1 (001)**: raggruppa tutte le istruzioni che mancano. Queste sono le istruzioni relative allo spazio di I/O, per le quali è necessario prelevare in memoria l'indirizzo (a 16 bit) della porta di I/O sorgente/destinatario, e le istruzioni `MOV` che hanno come operando uno dei registri a 24 bit `DP` o `SP`. Per queste, è necessario quantomeno leggere altri 3 byte dopo il codice operativo, puntati dal registro `IP`. Siccome i passi da eseguire sono diversi a seconda dell'istruzione, verranno gestiti successivamente nelle fasi di esecuzione individuali (ancorché questa modalità sia poco pulita dal punto di vista concettuale). Per le istruzioni di questo formato, la fase di fetch si limita quindi al prelievo del codice operativo (lettura di un byte in memoria).

1.3 Architettura del calcolatore

Dopo aver visto ciò che **vede un programmatore**, vediamo adesso la **struttura interna** dell'architettura del calcolatore. Cominciamo con lo specificare cosa c'è sulla **rete di interconnessione**, e quindi quali sono i collegamenti di ciascun modulo.



1) **fili di indirizzo**: ne servono **24** in tutto. Sono **uscite** per il processore, il quale imposterà gli indirizzi delle locazioni di memoria o delle porte di I/O dove vuole leggere e scrivere, ed **ingressi** per il resto del mondo. Visto che lo spazio di I/O consta di sole 64k porte, alcuni di questi dovranno essere buttati via.

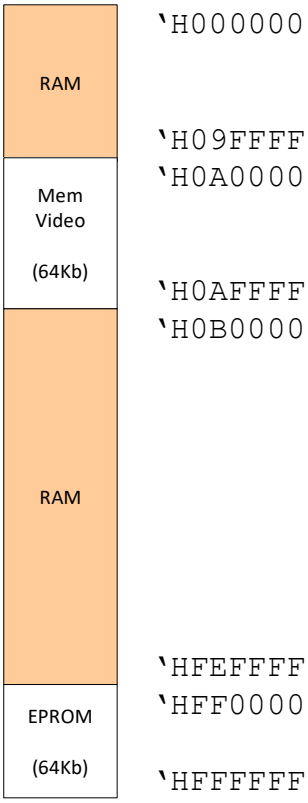
2) **fili di dati**: abbiamo detto che il processore legge e scrive **byte**. Quindi gli ci vogliono 8 fili di dati. Tali fili dovranno essere pilotati alternativamente dal processore e dagli altri dispositivi. Tutti, quindi, li dovranno **forchettate** in maniera opportuna. Quando scriveremo la descrizione del processore (e vedremo le interfacce) faremo attenzione a che non si verifichino mai cortocircuiti sui fili di dati.
3) **Fili di controllo**: tutti attivi bassi, /mr, /mw (per leggere e scrivere in memoria), /ior, /iow (per leggere e scrivere nello spazio di I/O). Uscite per il processore, ingressi per gli altri. I fili di accesso alla memoria verranno utilizzati coerentemente con la temporizzazione vista per i cicli di lettura e scrittura delle memorie RAM vista a suo tempo. I due fili per l'accesso allo spazio di I/O verranno utilizzati in maniera molto simile (non identica, vedremo più avanti).
4) **Segnale di clock**. Se tutti hanno un clock, ci deve pur essere qualcuno che lo genera.
5) **Fili di interconnessione tra interfacce e dispositivi**: ci sono anche quelli, quando sarà necessario li introdurremo.
6) **Fili di comunicazione tra la memoria video e l'adattatore grafico**: ci sono, non daremo ulteriori dettagli su quest'aspetto.

Il processore e la maggior parte delle interfacce sono RSS. Pertanto, avranno anche i loro **piedini per il reset** (che come al solito non disegniamo). Supporremo d'ora in avanti, che il reset arrivi **contemporaneamente** a tutti i moduli che ne hanno necessità.
Se vogliamo descrivere il sistema di cui sopra in Verilog, la descrizione è assolutamente banale (sono soltanto interconnessioni).

```
module Calcolatore(ai_dispositivi,dai_dispositivi);
  input [...:0] dai_dispositivi;
  output [...:0] ai_dispositivi;

  //bus
  wire [7:0] d7_d0;
  wire [23:0] a23_a0;
  wire mr_,mw_,ior_,iow_;
  wire clock, reset_;
  wire [15:0] a15_a0; assign a15_a0=a23_a0[15:0];

  //Collegamenti tra memoria video e adattatore grafico
  wire [...:0] a_mem_video;
  wire [...:0] da_mem_video;

  //Moduli costituenti il calcolatore
  Processore P(d7_d0,a23_a0,mr_,mw_,ior_,iow_,clock,reset_);
  Spazio_di_Memoria SdM(d7_d0,a23_a0,mr_,mw_,da_mem_video,a_mem_video);
  Spazio_di_IO SdIO(d7_d0,a15_a0,ior_,iow_,a_mem_video,da_mem_video,
                    ai_dispositivi,dai_dispositivi,clock,reset_);
  Generatore_del_Clock GC(clock);
endmodule
```

```
module Processore(…); … endmodule
module Spazio_di_Memoria(…); … endmodule
module Spazio_di_IO(…); … endmodule
module Generatore_del_Clock(…); … endmodule
module Gruppo RC con trigger di Schmitt(…); …
endmodule
```

1.3.1 Spazio di memoria



Lo spazio di memoria fisica, grande 16Mbyte, è implementato in parte con tecnologia RAM, in parte EPROM (la parte che contiene il programma di bootstrap), ed in parte come Memoria Video (di tipo ancora diverso, che non vediamo). Supponiamo (come esempio) di voler montare **64k di EPROM e 64k di memoria video,** con le seguenti specifiche:

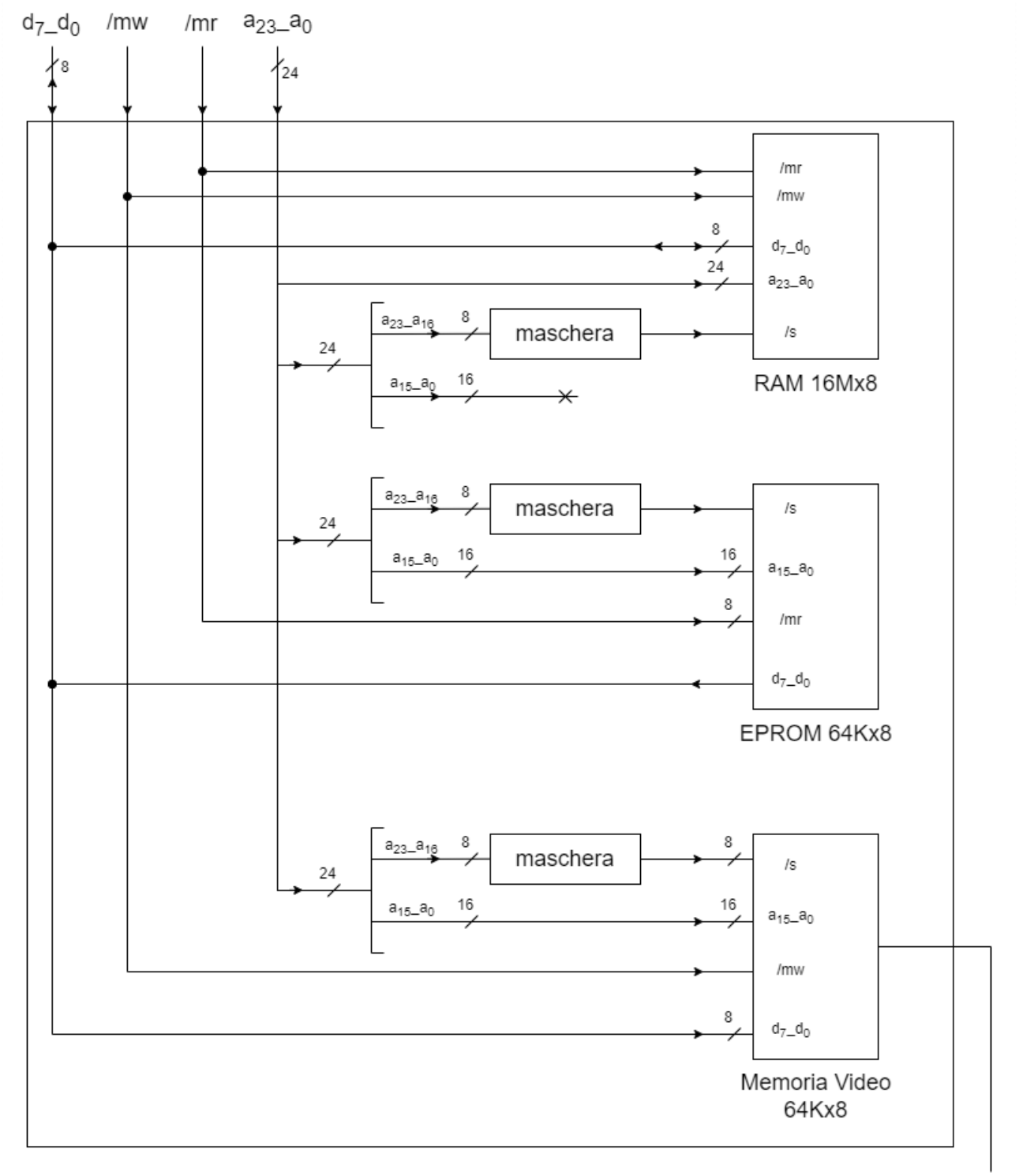
- la **EPROM** deve essere montata in modo tale che essa copra le locazioni tra **'HFF0000 e 'HFFFFFF**.
- La **memoria video** copre gli indirizzi fisici 'H0A0000–'H0AFFFF siano di memoria video.
- Il resto della memoria è memoria RAM volatile. In particolare, gli intervalli di indirizzi implementati con tecnologia RAM sono:
    - 'H000000-'H09FFFF
    - 'H0B0000-'HFEFFFF

Per poter soddisfare queste specifiche, devo realizzare un montaggio "in serie" di diversi moduli di memoria. Tali moduli dovranno essere **selezionati** a seconda dell'intervallo di indirizzi portati dal bus. Devo, quindi, generare il segnale di **select** per i tre moduli in modo tale che rispondano agli indirizzi richiesti. Ci vuole un minimo di **logica combinatoria**. Dei 24 fili che costituiscono gli indirizzi fisici, gli 8 più alti individuano il blocco di memoria che sto selezionando.

La semplice logica combinatoria che genera il segnale di abilitazione (/s) per un modulo a partire dagli indirizzi prende il nome di **maschera**. Si faccia caso ai seguenti punti:

- sul bus **non c'è nessun filo di select.** Il piedino /s di select è un ingresso dei chip di memoria, viene prodotto dagli indirizzi del bus (in genere quelli più significativi) e serve a poter implementare uno spazio di memoria unico usando chip diversi. Non ha alcun senso mettere un segnale di select nel bus (a chi dovrebbe andare?)

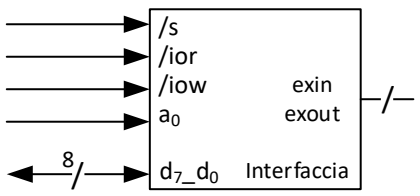
- Il chip di RAM copre anche gli indirizzi coperti dalla EPROM e dalla memoria video. Quando però il processore imposta uno di quegli indirizzi, la maschera che produce il select del chip di RAM non lo abilita, e quindi la RAM non risponde.



1.3.2 Spazio di I/O

Lo spazio di I/O (**parte di esso**, per essere precisi) è realizzato fisicamente tramite **interfacce**, che fungono da **raccordo tra il bus e i dispositivi** di I/O. Un'interfaccia, pertanto, ha dei collegamenti

sia “lato bus” che “lato dispositivo”. Per quanto riguarda i collegamenti dal lato del bus, saranno del tutto identici a quelli di una **piccola memoria RAM**, di poche locazioni (due, in questo esempio). Le locazioni che si trovano nelle interfacce prendono il nome di **porte** di ingresso e uscita.



/s	*/ior*	*/iow*	*Azione*
1	-	-	Nessuna azione
0	1	1	Nessuna azione
0	0	1	Azioni consistenti con un ciclo di lettura
0	1	0	Azioni consistenti con un ciclo di scrittura
0	0	0	Azioni non definite

L’interfaccia della figura ha due porte. Il processore può accedere ad una o all’altra, specificando un valore sul filo `a0`. Quando il processore vuole accedere in lettura, usa il piedino `/ior`. Quando vuole accedere in scrittura, usa il piedino `/iow`. Il piedino di select dell’interfaccia verrà generato dai **fili del bus indirizzi** (tutti meno quello connesso ad `a0`), tramite una **maschera**. La logica che viene inserita nella maschera determina a quale coppia di indirizzi risponde quest’interfaccia.

Saranno **simili, ma non identiche, anche le <u>temporizzazioni</u> per i cicli di lettura e scrittura** (sostituendo /ior ↔ /mr, /iow ↔ /mw. Alcune piccole differenze vanno comunque tenute in conto:

- in una RAM si può leggere e scrivere qualunque locazione. Spesso in un’interfaccia alcune porte **supportano soltanto lettura** (istruzione `IN`) **o soltanto scrittura** (`OUT`). Se un’intera interfaccia comprende soltanto porte in cui si può solo leggere o solo scrivere (caso abbastanza raro), allora non abbiamo bisogno di uno dei due fili di comando /ior, /iow. La maggior parte delle interfacce avranno comunque porte di entrambi i tipi.
- Se un’interfaccia implementa **una sola porta**, non sono necessari i fili di indirizzo (**basta /s**).

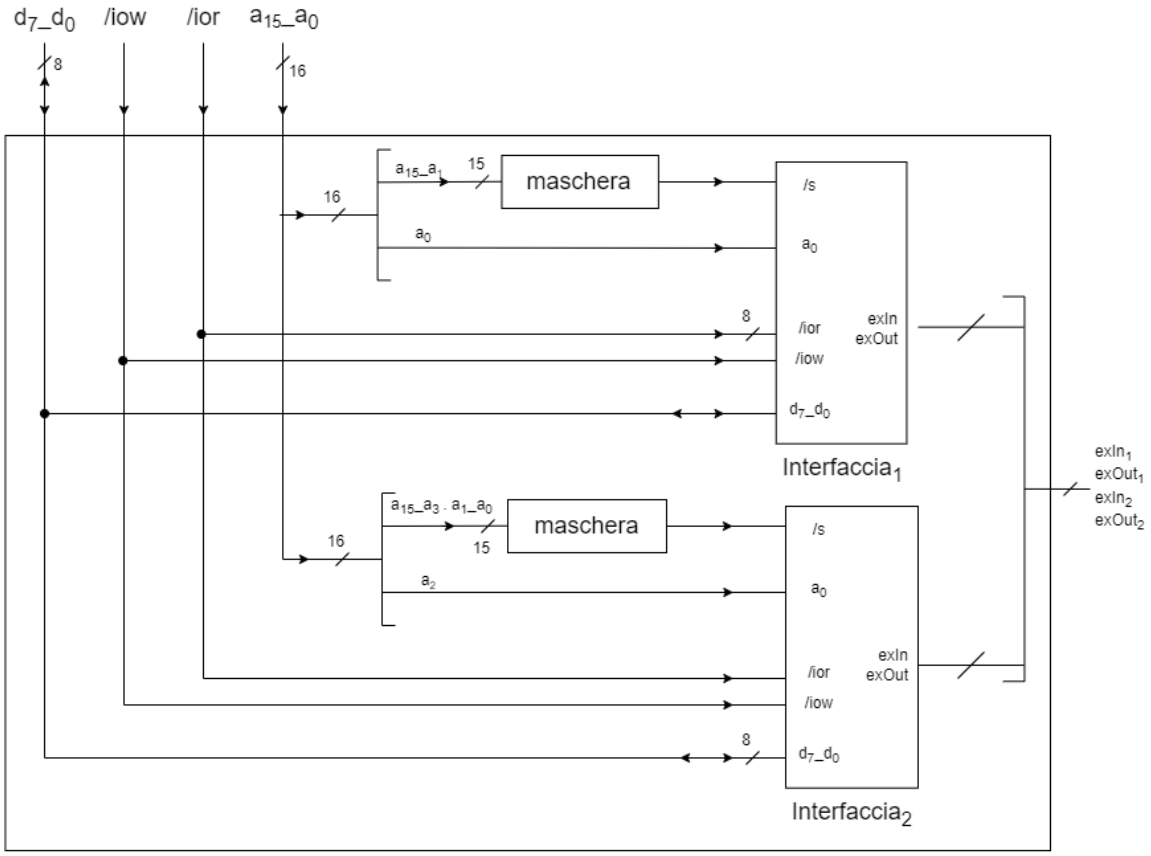
Dal lato dispositivo, invece, i collegamenti variano da interfaccia a interfaccia, e verranno descritti al momento opportuno. Il motivo per cui al bus si attaccano le interfacce, invece che direttamente i dispositivi, è duplice:

- i dispositivi hanno **velocità molto diverse tra loro** (per ordini di grandezza), e sono spesso **molto più lenti del processore**. Se sul bus ci fossero direttamente i dispositivi, il processore dovrebbe i) prevedere temporizzazioni diverse da dispositivo a dispositivo, e ii) perdere molto tempo ad aspettare i dispositivi lenti. In questo modo, il processore si attiene alla stessa temporizzazione (lettura/scrittura di interfacce), piuttosto veloce, e poi queste ultime si preoccupano di dialogare con i dispositivi con i tempi richiesti da questi ultimi.
- i dispositivi hanno **modalità di trasferimento dati molto diverse tra loro**. Alcuni trasferiscono **un bit alla volta** (seriali), altri **gruppi di bit (e.g., byte)**. Con un’interfaccia nel mezzo, il processore può comodamente fare letture e scritture **al byte**, e poi sarà l’interfaccia a comandare opportunamente il dispositivo.

Tutti questi aspetti dei dispositivi vengono appunto **mascherati** dalla presenza di interfacce.

Come **esempio**, supponiamo di avere uno spazio di I/O in cui sono montate **due interfacce**, ciascuna a due porte (sulle quali si può leggere e scrivere). La prima dà corpo a due porte che si trovano agli offset 'H03C8, 'H03C9. La seconda interfaccia dà corpo a due porte che si trovano agli offset 'H0060, 'H0064 **(non contigui).**

Ciascuna interfaccia riceverà, quindi, **un filo di indirizzi**, ed avrà il proprio **select** abilitato da una **maschera**, che deve dare 0 in uscita quando gli indirizzi sono corretti. In questo caso, come si vede dal disegno, il filo di indirizzo che va all'interfaccia n.1 è a_2, e non a_0. Gli altri fili vanno portati in parallelo ad entrambe. Ritorneremo più tardi sulle interfacce, descrivendone alcune significative dal punto di vista didattico.



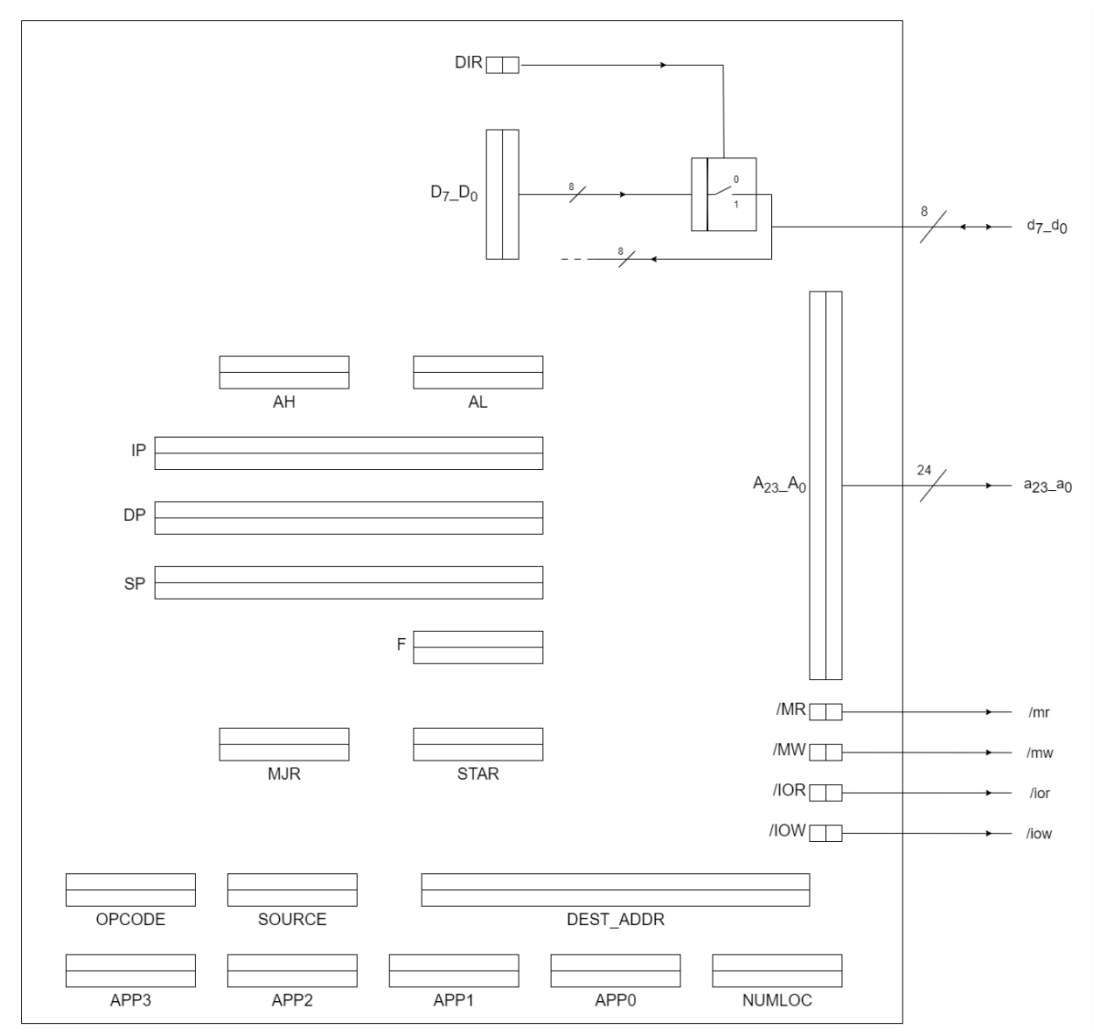
1.3.3 Processore

Il processore contiene un certo numero di registri. Alcuni, quelli visibili al programmatore, li abbiamo già visti. Altri li dobbiamo ancora descrivere.

- **`STAR`** ci sarà un registro di stato, essendo il processore una RSS;
- **`MJR`**: ne avrò bisogno, per i motivi che abbiamo già accennato;
- **Instruction registers (`OPCODE`, `SOURCE`, `DEST_ADDR`)**: vengono riempiti in fase di **fetch**, e contengono informazioni sull'istruzione da eseguire e sugli operandi. In particolare:
    - **`OPCODE`** conterrà il codice operativo dell'istruzione da eseguire;

- **SOURCE** conterrà l'operando sorgente, se questo sta in memoria;
- **DEST_ADDR** conterrà l'indirizzo dell'operando destinatario, se questo sta in memoria .

- Ho dei registri che **sostengono le uscite,** come deve essere in una RSS (indirizzi, dati, variabili di controllo);
- **Un registro DIR** per abilitare la tri-state quando il processore deve effettuare scritture sul bus (nello spazio di memoria o di I/O).
- **Dei registri di appoggio APPx** e **NUMLOC**, che servono per i cicli di lettura/scrittura (lo vedremo più avanti).

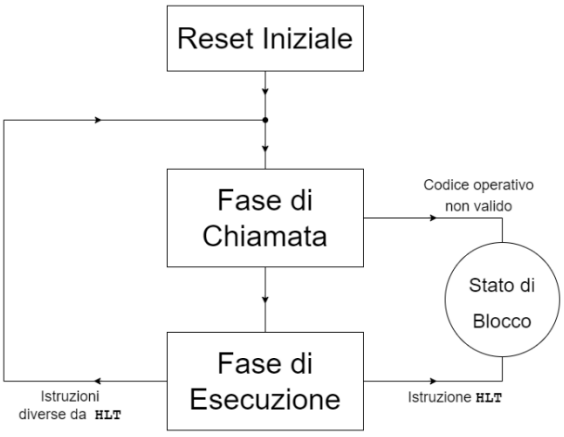


Abbiamo già avuto modo di vedere che il comportamento del processore si descrive attraverso un semplice diagramma a stati: c'è una fase di reset, in cui si inizializzano alcuni registri del processore (ad esempio `IP` e `F`, ma non solo). Seguono poi, ciclicamente, le fasi di fetch e di esecuzione. Si esce da questo ciclo per due motivi:

- Si esegue l'istruzione `HLT`, che blocca il processore
- Si preleva un'istruzione non valida.

In entrambi i casi il processore entra in un loop infinito, dal quale esce soltanto premendo il tasto di reset.

Alla luce delle informazioni che abbiamo, possiamo adesso dare qualche dettaglio in più su cosa avviene in ciascuna fase.



Al reset, si inizializzano:

- i registri **`IP` e `F`**, in modo da partire con un'evoluzione consistente. `IP` viene inizializzato a `'HFF0000`, cioè al primo indirizzo del blocco di EPROM. `F` viene inizializzato a 0.
- **Tutti i registri** che hanno a che fare con variabili di controllo del bus dovranno essere inizializzati in modo coerente: **`/MR, /MW, /IOR, /IOW`** dovranno tutti **contenere 1**.
- I fili di dati vanno posti **in alta impedenza**. `DIR` deve contenere 0. `DIR` starà **sempre a 0**, tranne quando devo scrivere qualcosa.
- **`STAR`** verrà inizializzato con l'etichetta del primo statement della fase di fetch.

Gli altri registri possono anche contenere valori casuali senza che ciò causi alcun problema (ci verranno comunque scritti dei valori in seguito).

Fase di fetch: il processore

- **preleva un byte** dalla memoria, all'indirizzo indicato in **`IP`**
- **incrementa `IP`** (modulo 2^{24})
- **controlla** che quel byte corrisponda all'opcode di una delle istruzioni che conosce. Se non è così si **blocca** (come se avesse eseguito una `HLT`)
- **inserisce** il byte letto nel registro **`OPCODE`**, e valuta il **formato dell'istruzione**.

A seconda del formato dell'istruzione, il processore deve fare alcune tra le seguenti cose:

- procurarsi **un operando sorgente a 8 bit**, ed inserirlo nel registro `SOURCE` (formati F2, F4, F5). A seconda del formato, dovrà fare:

- un accesso in mem. all'indirizzo contenuto in `DP` (formato **F2**, operando in mem. con indirizzamento indiretto)
- un accesso in mem. all'indirizzo contenuto in `IP` (formato **F4**, operando immediato)
- **due accessi** in memoria: all'indirizzo contenuto in `IP` per procurarsi l'indirizzo (che sta su 24 bit), e poi un altro accesso all'indirizzo trovato (formato **F5**, operando in mem. con indirizzamento diretto).

  In questi ultimi due casi dovrà anche **incrementare `IP`,** rispettivamente di uno (F4) e tre (F5) byte.

- procurarsi **l'indirizzo dell'operando destinatario,** ed inserirlo in `DEST_ADDR` (formati F3, F6, F7).
  - nel formato **F3** l'indirizzo sta già in DP. Basta copiarlo dove serve.
  - nei formati **F6** ed **F7** devo andarlo a leggere in memoria, leggendo 3 byte puntati da `IP`, ed incrementando opportunamente IP.
- Nel formato F0 non deve fare niente
- Nel formato F1 il processore farà cose particolari, che vedremo più avanti.

Come ultima cosa, in fase di fetch si guarda l'**`OPCODE`**, e si capisce quale istruzione dobbiamo realmente eseguire (finora ne avevamo preso in considerazione soltanto il formato). Come già osservato, gestire la fase di fetch in questo modo consente di eseguire nella stessa maniera operazioni che sono **simili** per fase di esecuzione, ma diverse per modalità di indirizzamento degli operandi.

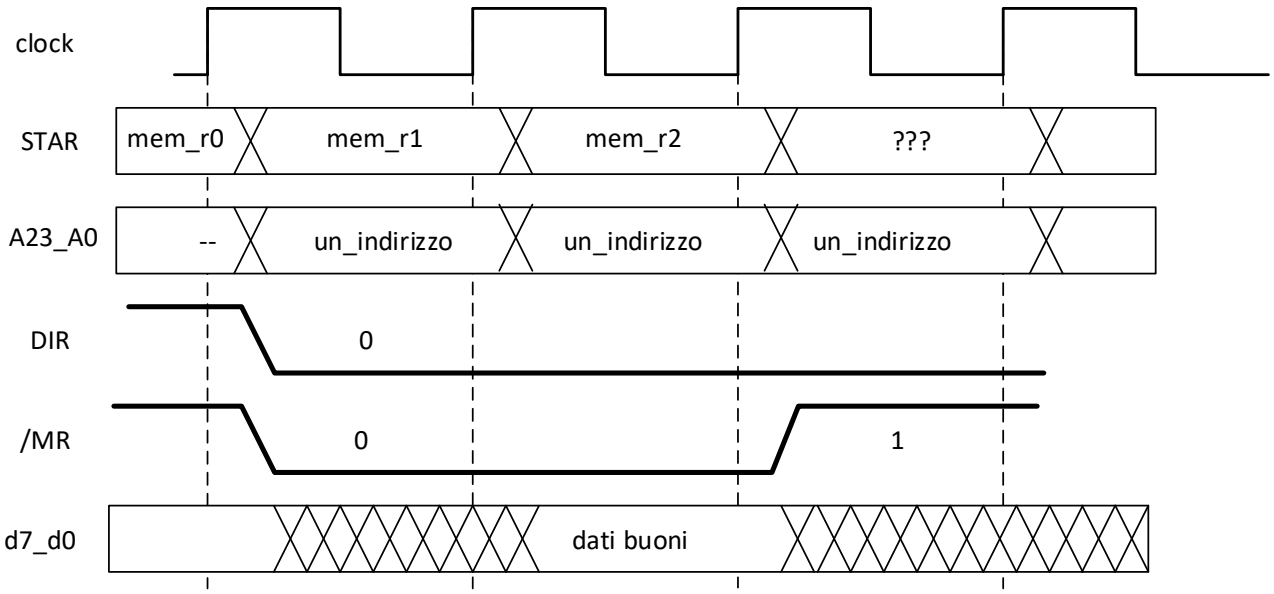
Fase di esecuzione: il processore esegue l'istruzione che ha decodificato, e poi torna nella fase di fetch, a meno che non stia eseguendo l'istruzione di `HLT`, nel qual caso si blocca e potrà essere sbloccato soltanto da un nuovo reset.

1.3.4 Lettura e scrittura in memoria e nello spazio di I/O

Durante la fase di fetch, il processore **legge in memoria.** Durante quella di esecuzione, il processore dovrà **leggere e scrivere in memoria (`MOV`) o nello spazio di I/O (`IN, OUT`)**. Vediamo come si fa a scrivere un frammento di μ-programma **compatibile con le temporizzazioni viste a suo tempo** per i cicli di lettura e di scrittura della memoria. Partiamo con **la lettura**. I registri coinvolti sono: `A23_A0`, `DIR`, `MR_`. I dati che vengono dal bus verranno appoggiati in qualche registro.

```
mem_r0: begin A23_A0<=un_indirizzo; DIR<=0; MR_<=0; STAR<=mem_r1; end
mem_r1: begin STAR<=mem_r2; end //stato di wait
mem_r2: begin QUALCHE_REGISTRO<=d7_d0; MR_<=1; ........
```

wait state. nel ciclo di lettura i dati **non sono buoni subito**.



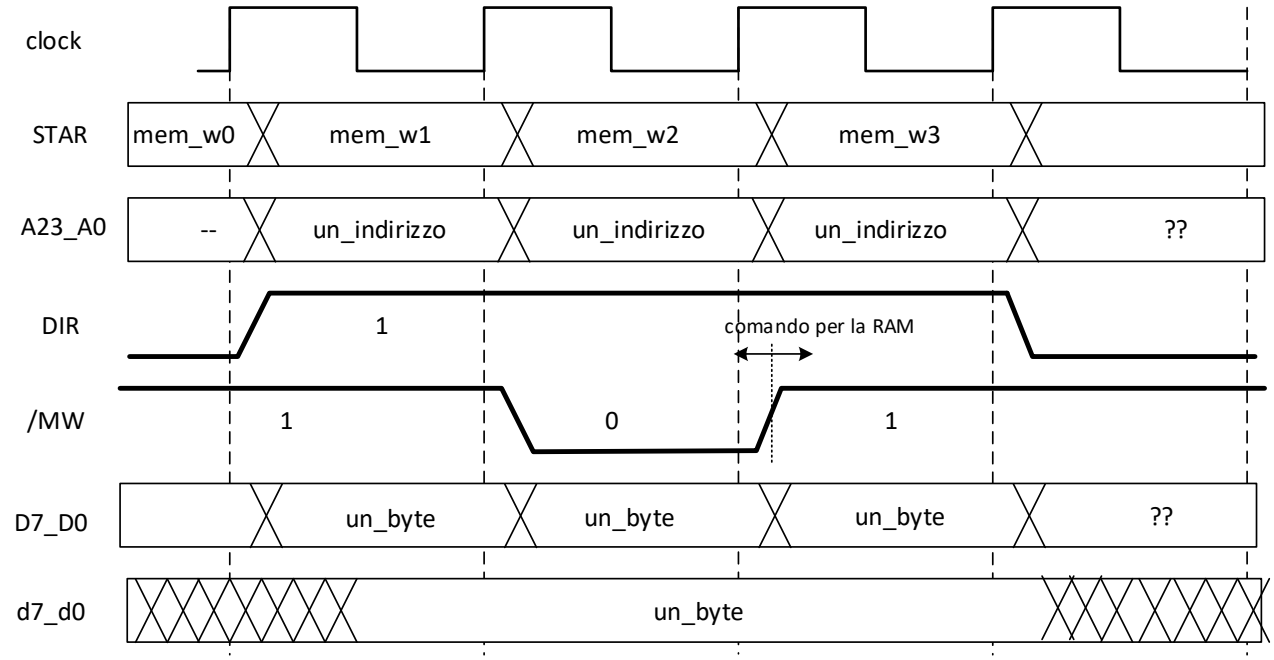
In `mem_r2`, se voglio, posso assegnare nuovamente `A23_A0` (ad esempio, per continuare la lettura ad un altro indirizzo). `DIR`, invece **non lo posso portare a** 1, in quanto le porte tri-state della RAM vanno in alta impedenza **un po' dopo il fronte di salita di /mr.** Quindi, se portassi subito `DIR` a 1, si rischierebbe un problema elettrico (anche se transitorio). In realtà `DIR` conviene tenerlo sempre a 0, tranne che durante le operazioni di scrittura.

D'ora in avanti faremo l'ipotesi che la memoria sia sufficientemente veloce da **non dover inserire uno stato di wait** (altrimenti le descrizioni vengono troppo lunghe).

Ciclo di scrittura

```
mem_w0: begin A23_A0<=un_indirizzo; D7_D0<=un_byte; DIR<=1;
              STAR<=mem_w1; end
mem_w1: begin MW_<=0; STAR<=mem_w2; end
mem_w2: begin MW_<=1; STAR<=mem_w3; end
mem_w3: begin DIR<=0; ..... ; end
```

Non posso riportare `DIR` a 0 subito, altrimenti i dati non sono stabili sul fronte di salita di /mw, come invece dev'essere.



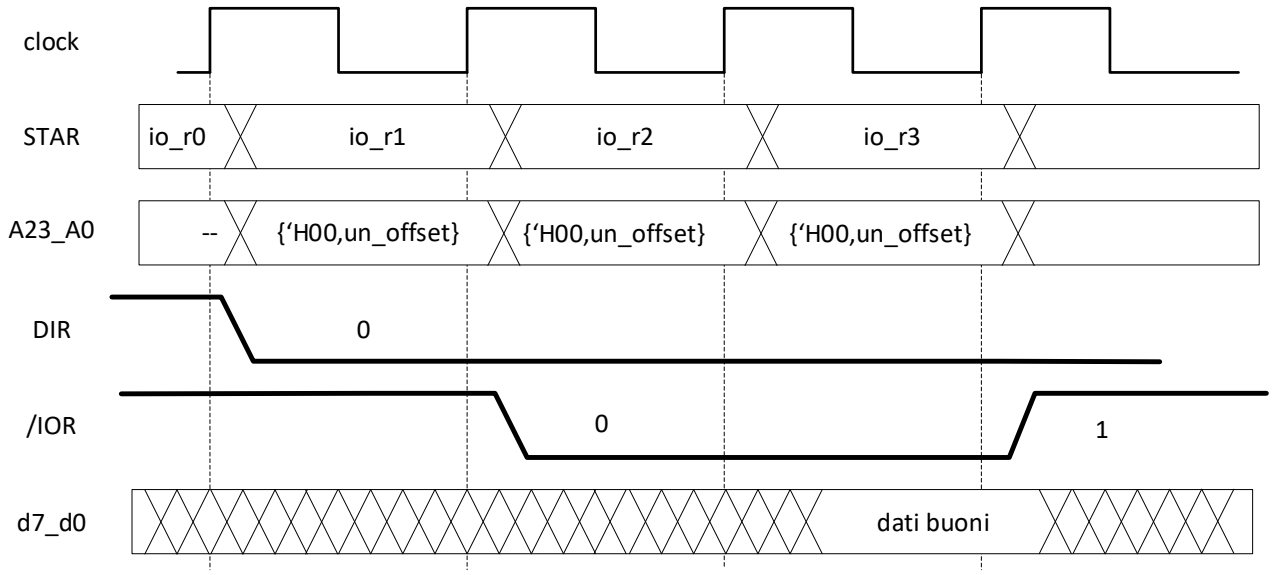
Se volessi, D7_D0 e DIR li potrei settare in mem_w1 senza problemi. In mem_w3 posso assegnare nuovamente D7_D0 e A23_A0, se necessario. Visto che DIR deve stare, normalmente, a 0, posso permettermi di tirare giù DIR un clock prima? **No**, perché la RAM memorizza il dato sul fronte di **salita** di /mw, e quindi i dati in scrittura e gli indirizzi devono essere stabili intorno a quel fronte. Quindi, se portassi a 0 DIR, i dati andrebbero in alta impedenza, e la RAM potrebbe memorizzare valori casuali. Per lo stesso motivo **non posso riassegnare A23_A0 in mem_w2**.

I cicli di lettura e scrittura **nello spazio di I/O** sono **simili ma non identici**: ci sono importanti differenze **che vanno ricordate:** le prime due sono che gli indirizzi sono a 16 bit**, e** si usano **/ior, /iow**, e **non /mr**, **/mw** (questa è una cosa che gli studenti tendono a dimenticare in sede d'esame).

Ciclo di lettura nello spazio di I/O

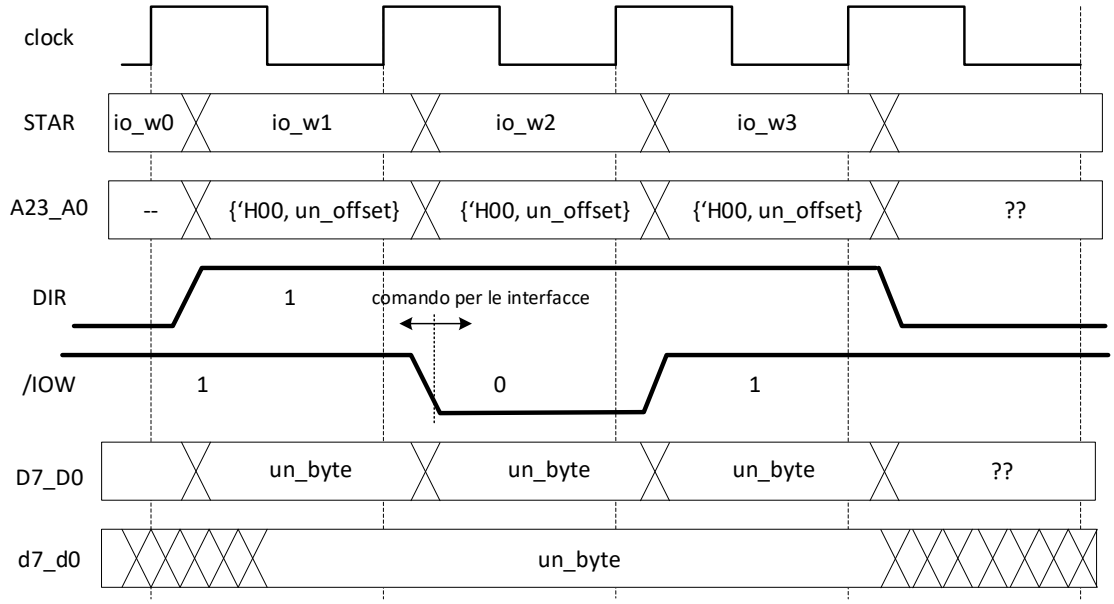
C'è una differenza **sostanziale** nel ciclo di lettura. **Gli indirizzi devono essere pronti un clock prima del comando di lettura (fronte di discesa di /ior)**. Il motivo è da ricercarsi nel particolare funzionamento delle interfacce. In alcuni casi (che vedremo), **leggere dei dati da una porta comporta la loro riscrittura da parte del dispositivo esterno**. Quindi, nello spazio di I/O anche le letture possono essere distruttive. In altri casi, inoltre, leggere dei dati da una porta può provocare **il cambiamento del contenuto di un'altra porta**. Ci sono infatti delle porte (registri di stato delle interfacce) che servono soltanto a contenere informazioni quali, ad esempio, "hai già (o non hai ancora) letto il dato che c'è in quest'*altra* porta". Quindi, se **per caso gli indirizzi o i select ballano (anche per poco) con /ior a 0**, si possono creare delle inconsistenze. Nel caso di letture in memoria, invece, questo problema non esiste. Per lo stesso motivo, A23_A0 non può essere riassegnato in io_r3, come avremmo fatto nel caso di lettura in memoria.

```
io_r0: begin A23_A0<={'H00,un_offset}; DIR<=0; STAR<=io_r1; end
io_r1: begin IOR_<=0; STAR<=io_r2; end
io_r2: begin STAR<=io_r3; end //stato di wait
io_r3: begin QUALCHE_REGISTRO<=d7_d0; IOR_<=1; .... ; end
```



Ciclo di scrittura nello spazio di I/O

```
io_w0: begin A23_A0<={'H00,un_offset}; D7_D0<=un_byte; DIR<=1;
             STAR<=io_w1; end
io_w1: begin IOW_<=0; STAR<=io_w2; end
io_w2: begin IOW_<=1; STAR<=io_w3; end
io_w3: begin DIR<=0; ........ ; end
```



Anche qui c'è una differenza **sostanziale** rispetto alla memoria. **I dati devono essere già pronti sul fronte di discesa di /iow.** Questo perché **molte interfacce memorizzano sul fronte di discesa di /iow**, (contrariamente alla memoria, che memorizza sul fronte di **salita di *mw***).

Accessi per più di un byte alla memoria

Il processore ha bisogno di fare accessi in memoria non solo al byte, ma anche ad operandi a **2 e 3 byte** (per prelevare **indirizzi nello spazio di I/O e di memoria**, rispettivamente), e, occasionalmente, di 4 byte (non nella parte di corso che vedremo). Per fare questo in modo semplice, fa comodo dotarsi

di μ-**sottoprogrammi** di lettura/scrittura di tipo modulare, che possano essere usati per leggere/scrivere 1, 2, 3, 4 byte. Lo facciamo utilizzando il registro `MJR`, e questo è uno dei motivi (non il solo) per cui dobbiamo aggiungerlo ai registri del processore.
I sottoprogrammi di lettura/scrittura usano il registro interno `NUMLOC` come contatore del numero di byte da leggere/scrivere (1, 2, 3, 4), ed i registri `APP0`, `APP1`, `APP2`, `APP3` per contenere i byte letti/da scrivere. La descrizione per le letture in memoria è questa (non ci sono *wait state*):

```
// MICROSOTTOPROGRAMMA PER LETTURE IN MEMORIA
readB: begin MR_<=0; NUMLOC<=1; STAR<=read0; end
readW: begin MR_<=0; NUMLOC<=2; STAR<=read0; end
readM: begin MR_<=0; NUMLOC<=3; STAR<=read0; end
readL: begin MR_<=0; NUMLOC<=4; STAR<=read0; end
read0: begin APP0<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
               STAR<=(NUMLOC==1)?read4:read1; end
read1: begin APP1<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
               STAR<=(NUMLOC==1)?read4:read2; end
read2: begin APP2<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
               STAR<=(NUMLOC==1)?read4:read3; end
read3: begin APP3<=d7_d0; A23_A0<=A23_A0+1; STAR<=read4; end
read4: begin MR_<=1; STAR<=MJR; end
```

Per leggere un byte contenuto in memoria alla locazione `un_indirizzo` si scrive:

```
Sx:   begin … A23_A0<=un_indirizzo; MJR<=Sx+1; STAR<=readB; end
Sx+1: begin … <utilizzo di APP0> end
```

Ed in maniera del tutto simile se vogliamo leggere 2 byte (`readW`), 3 byte (`readM`), 4 byte (`readL`). Al ritorno dalla chiamata, nei registri `APPx` ci saranno i byte letti, in ordine di lettura.

In modo duale, per la scrittura, abbiamo la seguente descrizione:

```
// MICROSOTTOPROGRAMMA PER SCRITTURE IN MEMORIA
writeB: begin D7_D0<=APP0; DIR<=1; NUMLOC<=1; STAR<=write0; end
writeW: begin D7_D0<=APP0; DIR<=1; NUMLOC<=2; STAR<=write0; end
writeM: begin D7_D0<=APP0; DIR<=1; NUMLOC<=3; STAR<=write0; end
writeL: begin D7_D0<=APP0; DIR<=1; NUMLOC<=4; STAR<=write0; end
write0: begin MW_<=0; STAR<=write1; end
write1: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write2; end
write2: begin D7_D0<=APP1; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
               STAR<=write3; end
write3: begin MW_<=0; STAR<=write4; end
write4: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write5; end
write5: begin D7_D0<=APP2; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
               STAR<=write6; end
write6: begin MW_<=0; STAR<=write7; end
write7: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write8; end
write8: begin D7_D0<=APP3; A23_A0<=A23_A0+1; STAR<= write9; end
write9: begin MW_<=0; STAR<= write10; end
write10: begin MW_<=1; STAR<= write11; end
write11: begin DIR<=0; STAR<=MJR; end
```

E quindi, per scrivere `dato_16_bit` a partire da `un_indirizzo`, dovremo scrivere:

```
Sx:   begin …    APP1<=dato_16_bit[15:8]; APP0<=dato_16_bit[7:0];
                 A23_A0<=un_indirizzo; MJR<=Sx+1; STAR<=writeW; end
Sx+1: begin …    end
```

1.3.5 Descrizione del processore in Verilog

Passiamo adesso in rassegna la descrizione del processore fatta in Verilog, che non farà altro che riassumere quello che abbiamo detto finora. Per rendere più semplice la descrizione, ci avvarremo di un certo numero di **reti combinatorie** che semplificano alcune operazioni.

```
//-----------------------------------------------------------------
// DESCRIZIONE COMPLETA DEL PROCESSORE
//-----------------------------------------------------------------
module Processore(d7_d0,a23_a0,mr_,mw_,ior_,iow_,clock,reset_);
  input          clock,reset_;
  inout [7:0]    d7_d0;
  output [23:0] a23_a0;
  output         mr_,mw_;
  output         ior_,iow_;

// REGISTRI OPERATIVI DI SUPPORTO ALLE VARIABILI DI USCITA E ALLE
// VARIABILI BIDIREZIONALI E CONNESSIONE DELLE VARIABILI AI REGISTRI
  reg            DIR;
  reg [7:0]      D7_D0;
  reg [23:0]     A23_A0;
  reg            MR_,MW_,IOR_,IOW_;
  assign         mr_=MR_;
  assign         mw_=MW_;
  assign         ior_=IOR_;
  assign         iow_=IOW_;
  assign         a23_a0=A23_A0;
  assign         d7_d0=(DIR==1)?D7_D0:'HZZ; //FORCHETTA

// REGISTRI OPERATIVI INTERNI
  reg [2:0]      NUMLOC;
  reg [7:0]      AL,AH,F,OPCODE,SOURCE,APP3,APP2,APP1,APP0;
  reg [23:0]     DP,IP,SP,DEST_ADDR;

// REGISTRO DI STATO, REGISTRO MJR E CODIFICA DEGLI STATI INTERNI
  reg [6:0]      STAR,MJR;
  parameter      fetch0=0, .... write11=86;

// RETI COMBINATORIE NON STANDARD
  function valid_fetch;
    input [7:0] opcode;
    ...
    ...
  endfunction
```

`valid_fetch()`. Prende in ingresso un byte e restituisce 1 se quel byte è l'opcode di un'istruzione nota, 0 altrimenti

```
  function [6:0] first_execution_state;
    input [7:0] opcode;
    ...
    ...
  endfunction
```

`first_execution_state()`. Prende in ingresso un byte, che interpreta come un opcode valido, e restituisce la codifica del primo stato interno dell'esecuzione dell'istruzione relativa

```
  function jmp_condition;
    input [7:0] opcode;
    input [7:0] flag;
    ...
    ...
  endfunction
```

`jmp_condition()`. Prende in ingresso due byte, che saranno il contenuto di `OPCODE` ed `F`. Restituisce 1 se `OPCODE` è la codifica di un salto incondizionato (`JMP`); `OPCODE` è la codifica di un salto condizionato, e la condizione richiesta da `OPCODE`, valutata testando il contenuto di `F`, risulta vera. È la rete che decide se si deve saltare o no.

```
  function [7:0] alu_result;
    input [7:0] opcode,operando1,operando2;
    ...
    ...
  endfunction
```

Simula la Arithmetic and Logic Unit (ALU) interna al processore. Interpreta i 3 byte passati in ingresso come un opcode, un operando sorgente, un operando destinatario, e restituisce il risultato su 8 bit dell'elaborazione svolta. Tale risultato sarà tipicamente usato per una scrittura dentro `AL/AH`

```
  function [3:0] alu_flag;
    input [7:0] opcode,operando1,operando2;
    ...
    ...
  endfunction
```

Prende in ingresso gli stessi byte e simula l'aggiornamento dei flag consistente con l'operazione specificata in `opcode` e con lo stato degli operandi. Ritorna quindi uno stato di uscita a 4 bit, che rappresentano i 4 flag significativi del registro `F`

```
// ALTRI MNEMONICI
  parameter [2:0]     F0='B000,F1='B001,F2='B010,F3='B011,
                      F4='B100,F5='B101,F6='B110,F7='B111;

//-------------------------------------------------------------
// AL RESET_ INIZIALE
  always @(reset_==0) #1 begin IP<='HFF0000; F<='H00; DIR<=0;
                           MR_<=1; MW_<=1; IOR_<=1; IOW_<=1;
                           STAR<=fetch0; end

//-------------------------------------------------------
// ALL'ARRIVO DEI SEGNALI DI SINCRONIZZAZIONE
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
```

- Fili di dati in alta impedenza
- fili di comando per il bus non attivi
- `F` e `IP` inizializzati correttamente
- Inizio dalla fase di fetch

Fase di fetch:

Per prima cosa, dovrò leggere il codice operativo ed incrementare `IP`. Questa parte è comune a tutti i formati. Nei formati F0 ed F1 non devo fare altro (nel formato F0 ho entrambi gli operandi nei registri, nel formato F1 me li procurerò in una fase successiva). In tutti gli altri casi devo inizializzare `SOURCE` o `DEST_ADDR`, compiendo azioni diverse. Ci sarà quindi una **prima parte uguale per tutti i formati**, e poi delle parti differenziate a seconda del formato. Dopo che ho letto il codice operativo avrei bisogno – in teoria – di eseguire un **salto a otto vie** per continuare la descrizione nello stato giusto. Per questo, uso il registro **`MJR`**, che serve apposta per gestire salti a molte alternative.

```
//-------------------------------------------------
// FASE DI CHIAMATA
fetch0: begin A23_A0<=IP; IP<=IP+1; MJR<=fetch1; STAR<=readB; end
fetch1: begin OPCODE<=APP0; STAR<=fetch2; end
fetch2: begin MJR<=(OPCODE[7:5]==F0)?fetchEnd:
                   (OPCODE[7:5]==F1)?fetchEnd:
                   (OPCODE[7:5]==F2)?fetchF2_0:
                   (OPCODE[7:5]==F3)?fetchF3_0:
                   (OPCODE[7:5]==F4)?fetchF4_0:
                   (OPCODE[7:5]==F5)?fetchF5_0:
                   (OPCODE[7:5]==F6)?fetchF6_0:
                   /* default */ fetchF7_0;
             STAR<=(valid_fetch(OPCODE)==1)?fetch3:nvi; end

fetch3: begin STAR<=MJR; end

fetchF2_0: begin A23_A0<=DP; MJR<=fetchF2_1; STAR<=readB; end
fetchF2_1: begin SOURCE<=APP0; STAR<=fetchEnd; end

fetchF3_0: begin DEST_ADDR<=DP; STAR<=fetchEnd; end

fetchF4_0: begin A23_A0<=IP; IP<=IP+1; MJR<=fetchF4_1;
                   STAR<=readB; end
fetchF4_1: begin SOURCE<=APP0; STAR<=fetchEnd; end

fetchF5_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF5_1;
                   STAR<=readM; end
fetchF5_1: begin A23_A0<={APP2,APP1,APP0}; MJR<=fetchF5_2;
                   STAR<=readB; end
fetchF5_2: begin SOURCE<=APP0; STAR<=fetchEnd; end

fetchF6_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF6_1;
                   STAR<=readM; end
fetchF6_1: begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; end

fetchF7_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF7_1;
                   STAR<=readM; end
fetchF7_1: begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; end

//-------------------------------------------------------------
// TERMINAZIONE DELLA FASE DI CHIAMATA
// TERMINAZIONE CON BLOCCO PER ISTRUZIONE NON VALIDA
nvi: begin STAR<=nvi; end

// TERMINAZIONE REGOLARE CON PASSAGGIO ALLA FASE DI ESECUZIONE
fetchEnd: begin MJR<=first_execution_state(OPCODE);
                   STAR<=fetchEnd1; end
fetchEnd1: begin STAR<=MJR; end
```

a. Prelevo opcode (ciclo di lettura in mem di un byte, all'indirizzo IP)
b. Incremento IP

a) A seconda dei primi 3 bit di OPCODE decido il formato.
b) Assegno a MJR l'etichetta del primo statement di fetch relativo al formato individuato.
c) Controllo se OPCODE è quello di un'istruzione, o sono bit a caso.

Vedere dopo

Quello che va fatto nei vari formati lo discutiamo tra un momento. Per adesso, mi interessa osservare che ciascuno dei blocchi di statement relativo ad un formato termina con

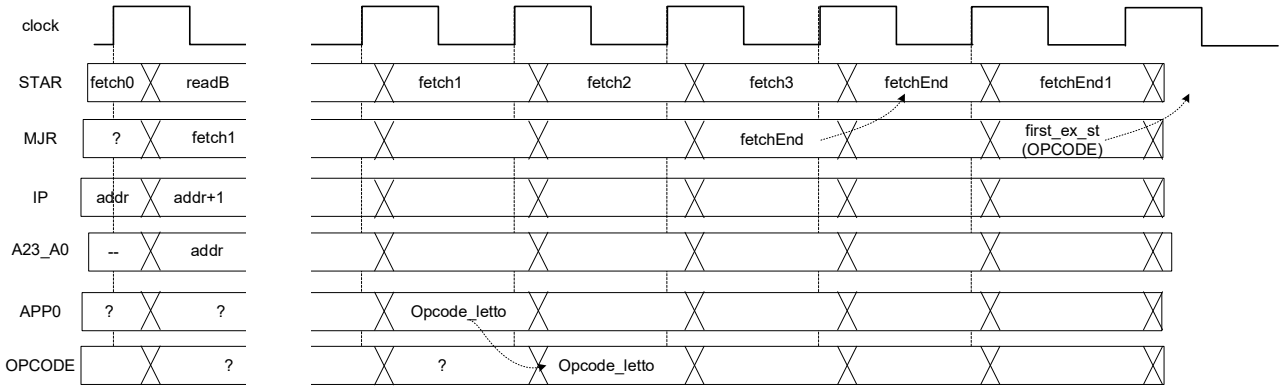
```
STAR<=fetchEnd;
```

Andiamo a vedere prima come termina la fase di fetch.

Se il processore legge un opcode non consentito, si blocca fino al nuovo reset

Al termine della fase di fetch, quello che devo fare dipende dal codice operativo. Devo quindi poter eseguire un salto a moltissime vie, tante quante sono le diverse istruzioni. Come al solito, userò MJR per gestire questo tipo di salto.

Temporizzazione della fase di fetch (esempio con formato F0/F1)



Vediamo adesso le operazioni da svolgere negli altri formati prima della terminazione della fase di fetch. Ricapitoliamo cosa succede in ciascuno dei formati, in particolare quelli che prevedono operandi in memoria.

F	Byte	OPCODE	SOURCE	DEST_ADDR
F0	1	readB @ IP	--	--
F1	*?*	readB @ IP	--	--
F2	1	readB @ IP	readB @ DP	--
F3	1	readB @ IP	--	DP
F4	2	readB @ IP	readB @ IP	--
F5	4	readB @ IP	readM @ IP, readB	--
F6	4	readB @ IP	--	readM @ IP
F7	4	readB @ IP	--	readM @ IP

Formato F2: devo leggere un byte all'indirizzo puntato da DP, e devo metterlo in SOURCE:

```
fetchF2_0: begin A23_A0<=DP; MJR<=fetchF2_1; STAR<=readB; end
fetchF2_1: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

Formato F3: DP contiene l'indirizzo del destinatario. Devo quindi inizializzare DEST_ADDR a DP:

```
fetchF3_0: begin DEST_ADDR<=DP; STAR<=fetchEnd; end
```

Formato F4: devo leggere un byte di operando sorgente all'indirizzo puntato da IP, e metterlo in SOURCE:

```
fetchF4_0: begin A23_A0<=IP; IP<=IP+1; MJR<=fetchF4_1;
                STAR<=readB; end
fetchF4_1: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

Formato F5: devo prima procurarmi l'indirizzo dell'operando sorgente, leggendo 3 byte a partire da `IP`. Successivamente, devo leggere in memoria all'indirizzo che ho prelevato, e portare l'operando sorgente in `SOURCE`:

```
fetchF5_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF5_1;
                 STAR<=readM; end
fetchF5_1: begin A23_A0<={APP2,APP1,APP0}; MJR<=fetchF5_2;
                 STAR<=readB; end
fetchF5_2: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

Formati F6/F7: devo procurarmi l'indirizzo dell'operando destinatario, leggendo 3 byte a partire da `IP`, ed assegnarlo a `DEST_ADDR`. Si noti che i due formati sono identici, per quanto riguarda il fetch, ed in fase di ottimizzazione di un ipotetico circuito un progettista serio li avrebbe scritti una volta sola. Li manteniamo distinti perché adesso privilegiamo la leggibilità.

```
fetchF6_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF6_1;
                 STAR<=readM; end
fetchF6_1: begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; end

fetchF7_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF7_1;
                 STAR<=readM; end
fetchF7_1: begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; end
```

Importante. All'uscita della fase di fetch:

- `OPCODE` contiene il codice operativo dell'istruzione
- Se l'istruzione ha un operando sorgente in memoria, questo sta in `SOURCE` (indipendentemente dalla modalità di indirizzamento usata dal programmatore nell'istruzione Assembler);
- Se l'istruzione ha un operando destinatario in memoria, il suo indirizzo sta in `DEST_ADDR` (indipendentemente dalla modalità di indirizzamento usata dal programmatore nell'istruzione Assembler).
- `IP` è stato incrementato del numero di byte necessario, e punta alla **prossima** istruzione da prelevare.

Guardiamo adesso **la fase di esecuzione**. Sarà abbastanza semplice, perché:

- una parte della complessità è stata gestita in fase di fetch;
- la maggior parte delle istruzioni "complesse" (quelle logico-aritmetiche) sono implementate tramite reti combinatorie, e quindi si riducono ad un semplice assegnamento procedurale.

```
//------------- istruzione NOP ----------------
nop: begin STAR<=fetch0; end

//------------- istruzione HLT ----------------
hlt: begin STAR<=hlt; end

//------------- istruzione MOV AL,AH -----------
```

Alla fine della fase di esecuzione di ogni istruzione si torna al primo statement di fetch, a meno di non eseguire la **HLT**, nel qual caso ci dobbiamo bloccare fino al nuovo reset.

```
ALtoAH: begin AH<=AL; STAR<=fetch0; end

//------------- istruzione MOV AH,AL ----------------
AHtoAL: begin AL<=AH; STAR<=fetch0; end

//------------- istruzione INC DP ----------------
incDP: begin DP<=DP+1; STAR<=fetch0; end

//------------- istruzioni MOV (DP),AL -----------
                //         MOV $operando,AL
                //         MOV indirizzo,AL
ldAL: begin AL<=SOURCE; STAR<=fetch0; end

//------------- istruzioni MOV (DP),AH ------------
                //         MOV $operando,AH
                //         MOV indirizzo,AH
ldAH: begin AH<=SOURCE; STAR<=fetch0; end

//------------- istruzioni MOV AL,(DP) ---------
                //         MOV AL,indirizzo
storeAL: begin A23_A0<=DEST_ADDR; APP0<=AL;
MJR<=fetch0; STAR<=writeB; end

//------------- istruzioni MOV AH,(DP) ----------
                //         MOV AH,indirizzo
storeAH: begin A23_A0<=DEST_ADDR; APP0<=AH;
MJR<=fetch0; STAR<=writeB; end
```

Guardiamo le istruzioni con codice mnemonico `MOV`. Quando spostano qualcosa da un registro a un altro sono assolutamente banali.

Tutte le `MOV` che hanno un sorgente in memoria e lo stesso registro destinatario (e.g., `AL`) possono essere collassate in un unico statement, perché la fase di fetch le ha rese omogenee, portando il sorgente dentro `SOURCE`.

Tutte le `MOV` che hanno un destinatario in memoria e un sorgente nello stesso registro possono essere collassate nella stessa etichetta. In questo caso, la fase di esecuzione consiste in un **accesso in memoria** in scrittura (di un byte) per copiare il destinatario, all'indirizzo contenuto in `DEST_ADDR`. Il valore da scrivere va inserito in `APP0`.

Adesso ci sono le sei istruzioni del formato F1, quelle per cui in fase di fetch non è stato fatto niente – non perché non ci fossero da prelevare operandi, ma perché il modo di prelevarli è diverso per ciascuna delle istruzioni, e non riconducibile a nessuno degli altri formati. Per ciascuna di queste, dobbiamo procurarci gli operandi ed eseguire l'istruzione.

```
//---------- istruzione MOV $operando,SP ---------
ldSP: begin A23_A0<=IP; IP<=IP+3; MJR<=ldSP1;
                STAR<=readM; end
ldSP1: begin SP<={APP2,APP1,APP0}; STAR<=fetch0;
            end
//---------- istruzione MOV $operando,DP --------
ldimmDP: begin A23_A0<=IP; IP<=IP+3; MJR<=ldimmDP1; STAR<=readM; end
ldimmDP1: begin DP<={APP2,APP1,APP0}; STAR<=fetch0;
end

//------------- istruzione MOV indirizzo,DP -------
lddirDP: begin A23_A0<=IP; IP<=IP+3; MJR<=lddirDP1;
                STAR<=readM; end
lddirDP1: begin A23_A0<={APP2,APP1,APP0};
                 MJR<=lddirDP2; STAR<=readM; end
lddirDP2: begin DP<={APP2,APP1,APP0};
                STAR<=fetch0; end
```

Dobbiamo leggere in memoria 3 byte (a partire da dove punta il registro `IP`) ed assegnare il risultato della lettura a `SP` o `DP`.

Dobbiamo leggere in memoria 3 byte (a partire da dove punta il registro `IP`) per prelevare un **indirizzo** di memoria, da cui poi leggere i 3 byte da assegnare a `DP`. Usiamo `A23_A0` come appoggio.

```
//------------- istruzione MOV DP,indirizzo -----
storeDP: begin A23_A0<=IP; IP<=IP+3;
                MJR<=storeDP1; STAR<=readM; end
storeDP1: begin A23_A0<={APP2,APP1,APP0};
                {APP2,APP1,APP0}<=DP;
                MJR<=fetch0; STAR<=writeM; end

//------------- istruzione IN offset,AL ---------
in: begin A23_A0<=IP; IP<=IP+2; MJR<=in1;
          STAR<=readW; end
in1: begin A23_A0<={'H00,APP1,APP0}; STAR<=in2; end
in2: begin IOR_<=0; STAR<=in3; end
in3: begin AL<=d7_d0; IOR_<=1; STAR<=fetch0; end

//------------- istruzione OUT AL,offset ---------
out: begin A23_A0<=IP; IP<=IP+2;
          MJR<=out1; STAR<=readW; end
out1: begin A23_A0<={'H00,APP1,APP0}; D7_D0<=AL;
                DIR<=1; STAR<=out2; end
out2: begin IOW_<=0; STAR<=out3; end
out3: begin IOW_<=1; STAR<=out4; end
out4: begin DIR<=0; STAR<=fetch0; end
```

Dobbiamo leggere in memoria 3 byte (a partire da dove punta il registro `IP`) per prelevare un **indirizzo** di memoria, dove poi **scrivere** il contenuto di `DP` (3 byte). Usiamo `A23_A0` come appoggio.

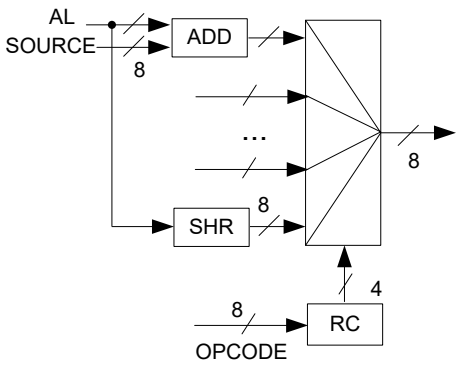
Dobbiamo leggere **2 byte in memoria** (a partire da dove punta il registro `IP`) per prelevare un **indirizzo nello spazio di I/O**. Da questo indirizzo leggeremo un byte (tramite un ciclo di lettura nello spazio di I/O) e lo metteremo in `AL`.

Dobbiamo leggere **2 byte in memoria** (a partire da dove punta il registro `IP`) per prelevare un **indirizzo nello spazio di I/O**. Dobbiamo poi eseguire un ciclo di scrittura nello spazio di I/O, copiando il contenuto di `AL`.

A questo punto – se possibile – è ancora più chiaro perché le istruzioni sono divise in formati. Parte degli statement relativi alle sei istruzioni "speciali" del formato F1 sono servite soltanto a procurarsi gli operandi. Dividendo le istruzioni in formati, abbiamo dovuto scrivere questi statement **una volta per formato**, invece che **una volta per istruzione**, riducendo notevolmente il numero degli stati interni necessari per descrivere il processore.

```
//------- istruzioni ADD (DP),AL -------
      // ADD $operando,AL
      // ADD indirizzo,AL
      // SUB (DP),AL
      // SUB $operando,AL
      // SUB indirizzo,AL
      // AND (DP),AL
      // AND $operando,AL
      // AND indirizzo,AL
      // OR (DP),AL
      // OR $operando,AL
      // OR indirizzo,AL
      // CMP (DP),AL
      // CMP $operando,AL
      // CMP indirizzo,AL
      // NOT AL
      // SHL AL
      // SHR AL
aluAL: begin
      AL<=alu_result(OPCODE,SOURCE,AL);
      F<={F[7:4],alu_flag(OPCODE,SOURCE,AL)};
         STAR<=fetch0; end

//----- istruzioni ADD (DP),AH --
      // ADD $operando,AH
      // ADD indirizzo,AH
      // SUB (DP),AH
      // SUB $operando,AH
      // SUB indirizzo,AH
      // AND (DP),AH
      // AND $operando,AH
      // AND indirizzo,AH
      // OR (DP),AH
      // OR $operando,AH
      // OR indirizzo,AH
      // CMP (DP),AH
      // CMP $operando,AH
      // CMP indirizzo,AH
      // NOT AH
      // SHL AH
      // SHR AH
aluAH: begin AH<=alu_result(OPCODE,SOURCE,AH);
                F<={F[7:4],alu_flag(OPCODE,SOURCE,AH)};
                STAR<=fetch0; end

//----- istruzioni JMP indirizzo -----------
      // JA indirizzo
      // JAE indirizzo
      // JB indirizzo
      // JBE indirizzo
      // JC indirizzo
      // JE indirizzo
      // JG indirizzo
      // JGE indirizzo
      // JL indirizzo
      // JLE indirizzo
      // JNC indirizzo
```

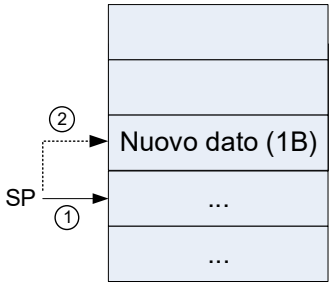
Per tutte le istruzioni **logico/aritmetiche**, quelle cioè che **fanno qualche conto** invece di spostare soltanto informazione da una parte all'altra (come quelle che abbiamo visto finora) faremo la seguente ipotesi: c'è una **rete combinatoria** detta ALU, che possiamo immaginare come la parte combinatoria che sta davanti a un registro multifunzionale a **otto funzioni**, in cui il registro è appunto `AL`. Le varie funzioni combinatorie prenderanno quindi l'ingresso da `AL` e (in qualche caso) da `SOURCE`, nel quale ho inserito gli operandi quando me li sono procurati in fase di fetch. Quindi, tutta la complessità delle operazioni logico/aritmetiche è mascherata da questa rete. Lo stesso per l'aggiornamento dei **flag.**



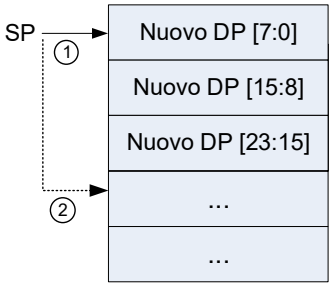
Per tutte le istruzioni di salto (condizionato e non), la decisione se saltare o meno è fornita dalla rete combinatoria
`jmp_condition()`,
che, sulla base dell'opcode (e quindi della condizione richiesta), stabilisce se la condizione sia vera o meno. Se si deve saltare, il nuovo valore di `IP` è il contenuto di `DEST_ADDR`, perché il formato F7 (quello delle istruzioni di salto) mette in `DEST_ADDR` l'indirizzo contenuto nell'istruzione stessa.

```
      // JNE indirizzo
      // JNO indirizzo
      // JNS indirizzo
      // JNZ indirizzo
      // JS indirizzo
      // JO indirizzo
      // JZ indirizzo
jmp: begin IP<=(jmp_condition(OPCODE,F)==1)?DEST_ADDR:IP;
                STAR<=fetch0; end

//------------- istruzione PUSH AL ----------------
pushAL: begin A23_A0<=SP-1; SP<=SP-1; APP0<=AL;
                MJR<=fetch0; STAR<=writeB; end

//------------- istruzione PUSH AH ----------------
pushAH: begin A23_A0<=SP-1; SP<=SP-1; APP0<=AH;
                MJR<=fetch0; STAR<=writeB; end

//------------- istruzione PUSH DP ----------------
pushDP: begin A23_A0<=SP-3; SP<=SP-3;
                {APP2,APP1,APP0}<=DP;
                MJR<=fetch0; STAR<=writeM; end

//------------- istruzione POP AL ----------------
popAL: begin A23_A0<=SP; SP<=SP+1;
                MJR<=popAL1; STAR<=readB; end
popAL1: begin AL<=APP0; STAR<=fetch0; end

//------------- istruzione POP AH ----------------
popAH: begin A23_A0<=SP; SP<=SP+1;
                MJR<=popAH1; STAR<=readB; end
popAH1: begin AH<=APP0; STAR<=fetch0; end

//------------- istruzione POP DP ----------------
popDP: begin A23_A0<=SP; SP<=SP+3;
                MJR<=popDP1; STAR<=readM; end
popDP1: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end
```

Le istruzioni di PUSH sono semplici **scritture** in memoria. L'indirizzo a cui scrivere è dato da SP-Nbyte. Devo poi **decrementare SP del numero di byte dell'operando.**



Le istruzioni di POP sono semplici **letture** in memoria. L'indirizzo da cui leggere è dato da SP. Devo poi **incrementare SP del numero di byte dell'operando.**



Rimangono le due istruzioni CALL e RET, che però sono estremamente semplici, perché:

- una CALL è una PUSH del valore corrente di IP (che è già stato incrementato in fase di fetch, quindi punta **all'istruzione di ritorno**), più la sostituzione di IP con l'indirizzo contenuto nella CALL, che si trova in DEST_ADDR;
- una RET è una POP di 3 byte dalla pila e la sostituzione di IP con il valore letto

```
//------------- istruzione CALL indirizzo ---------
call: begin A23_A0<=SP-3; SP<=SP-3; {APP2,APP1,APP0}<=IP;
                MJR<=call1; STAR<=writeM; end
call1: begin IP<=DEST_ADDR; STAR<=fetch0; end

//------------- istruzione RET ----------------
```

```
ret: begin A23_A0<=SP; SP<=SP+3; MJR<=ret1; STAR<=readM; end
ret1: begin IP<={APP2,APP1,APP0}; STAR<=fetch0; end
```

Per completare la descrizione del processore vanno aggiunti gli statement (che abbiamo già visto prima) per le letture/scritture di uno o più byte in memoria. In tutto abbiamo 86 statement, e quindi possiamo dimensionare correttamente i registri `STAR` e `MJR`, che devono essere di 7 bit.

Si noti che:

- in ogni stato ci sono μ-salti **al massimo a due vie**. Anzi, la maggior parte dei μ-salti è ad una sola via (i μ-salti a due vie si trovano quasi esclusivamente nelle sottoliste di lettura/scrittura in memoria);
- se vi venisse chiesto di sintetizzare la parte operativa relativa (e.g.) al registro `IP`, sapreste esattamente come fare. Non sarebbe particolarmente difficile.
- Le reti combinatorie riportate in cima alla descrizione (e.g., la ALU, la `jmp_condition`, etc.) non contengono niente che non abbiamo già visto in dettaglio a lezione.

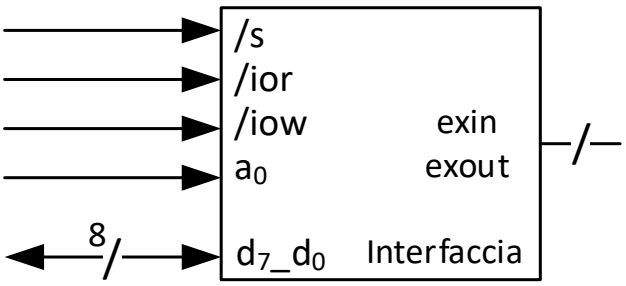
Pertanto, se vi venisse chiesto di sintetizzare questo processore in accordo al modello PO/PC, non ci sarebbero difficoltà concettuali. Dopo due mesi di corso, siete in grado di **sintetizzare dell'hardware capace di eseguire programmi software arbitrariamente complessi**.

1.3.6 Esercizi (da fare a casa)

- Sintetizzare la porzione di parte operativa relativa ad uno qualunque dei registri operativi del processore (si tenga conto che quella di `MJR`, ancorché banale, richiede più tempo).
- Sintetizzare la rete combinatoria di condizionamento del processore.

2 Interfacce

Iniziamo adesso a descrivere le varie interfacce che completano il calcolatore. In particolare, ci soffermeremo su quelle **parallele**, che sono in grado di colloquiare con dispositivi ai quali inviano (o dai quali prelevano) **un byte alla volta;** quelle **seriali**, che colloquiano con dispositivi con i quali scambiano **un bit alla volta**, e quelle per la **conversione analogico/digitale e digitale/analogica**, che trasformano gruppi di bit in tensioni e viceversa.

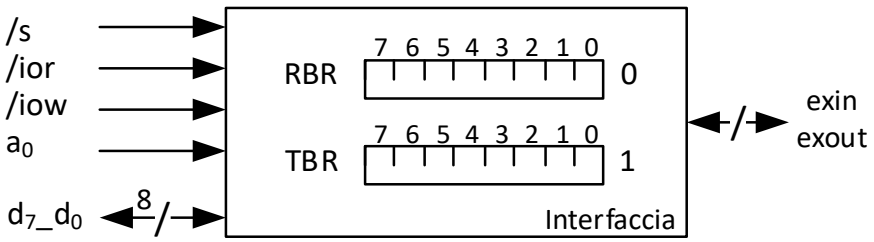


Abbiamo già visto un'interfaccia (qualunque) dal punto di vista del suo collegamento con il bus, ed abbiamo visto come il processore possa eseguire letture/scritture su porte di interfacce, dato l'indirizzo alle quali queste sono montate nello spazio di I/O. Un'interfaccia di ingresso/uscita è **sempre fatta allo stesso modo** dalla parte del processore (questo è appunto il suo scopo: presentare una visione **standard** del dispositivo che sta a valle). Sia i collegamenti del bus sia il modo di accedere alle porte di ingresso/uscita **non dipendono dalla natura del dispositivo a valle**. Quindi, per sapere se un'interfaccia è, ad esempio, **seriale o parallela** si deve guardare il modo in cui è connessa al suo dispositivo.

Domanda tipica d'esame: chi decide l'offset delle porte di I/O? **Il progettista che progetta la maschera che produce il segnale di select.**

L'interfaccia disegnata sopra dà corpo a **due porte** nello spazio di ingresso/uscita, distinte dal valore dell'unico filo di indirizzo che ci arriva.

Prima di descrivere le varie interfacce nel dettaglio, diamo alcune note generali. Dal punto di vista **funzionale**, cioè di chi ci deve interagire (un sistemista, per il montaggio, o un programmatore), un'interfaccia è dotata di collegamenti standard e qualche registro. Ad esempio, quella di figura ha due registri: **Receive Buffer Register** e **Transmit Buffer Register**. Pertanto, è un'interfaccia di **ingresso/uscita** (se fosse solo di ingresso mancherebbe `TBR`, etc.).



I due registri sono distinti dall'indirizzo interno (0 per `RBR`, 1 per `TBR`), che viene portato dal filo a0 del bus (gli altri 15 fili di indirizzo contribuiscono a generare il select). Il registro `RBR` contiene i dati scritti dal dispositivo esterno, mentre il registro `TBR` contiene i dati da mandare al dispositivo esterno. Gli altri collegamenti dalla parte del bus sono standard. I collegamenti dal lato dei dispositivi dipendono dai dispositivi medesimi, e possono essere molto diversi. Il programmatore che voglia leggere il dato fornito dal dispositivo dovrà eseguire un'istruzione del tipo:

```
IN offset_RBR, %AL
```

Se invece vuole scrivere qualcosa da mandare al dispositivo, dovrà scrivere:

```
OUT %AL, offset_TBR
```

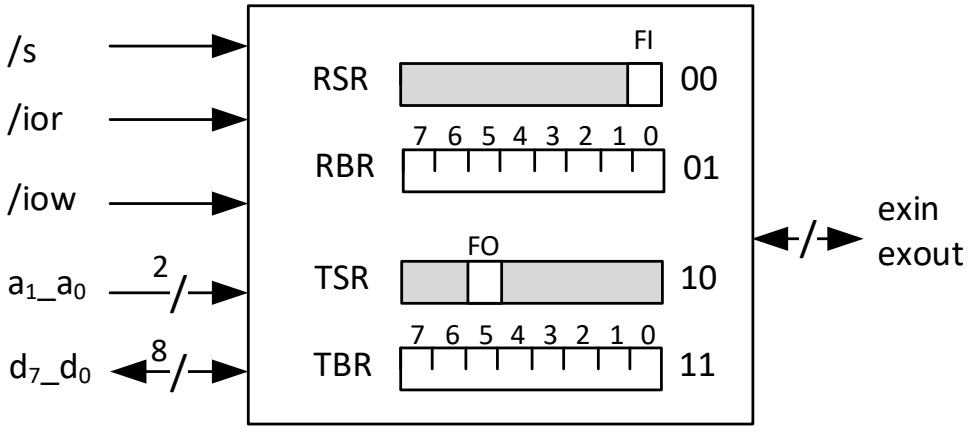
Un'interfaccia di questo tipo non consente alcuna **sincronizzazione tra processore e dispositivo**. Infatti, il processore non ha modo di sincronizzarsi con i dispositivi. Supponiamo che un programma contenga le seguenti istruzioni:

```
IN offset_RBR, %AL
…
IN offset_RBR, %AL
```

Nessuno può garantire che tra le due `IN` il dispositivo sia stato in grado di produrre un dato nuovo, e quindi la seconda `IN` potrebbe avere come esito l'ingresso di un dato non significativo. In maniera duale, se un programma contiene:

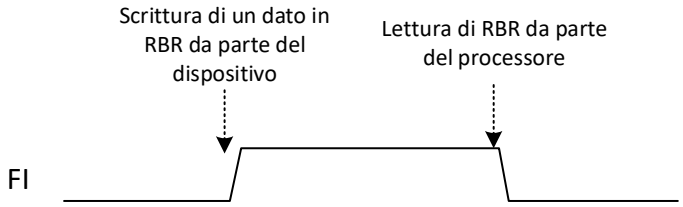
```
OUT %AL, offset_TBR
…
OUT %AL, offset_TBR
```

non c'è modo di garantire che tra le due `OUT` il dispositivo sia stato in grado di processare il dato. Il processore (e quindi il programmatore) non ha nessun modo per accorgersene. Per ovviare a questo problema è necessario usare interfacce leggermente più complesse, dotate di **registri di stato**, che servono ad implementare un handshake tra il processore ed i dispositivi. A livello di visione funzionale, le interfacce con handshake sono fatte in questo modo:



I due registri aggiunti si chiamano **Receive Status Register** e **Transmit Status Register**. Spesso sono collassati in un unico registro `RTSR`. Di ciascun registro è significativo **un solo bit** (il che dà appunto la possibilità di collassarli), detto rispettivamente **flag di buffer ingresso pieno (`FI`)** e **flag di buffer di uscita vuoto (`FO`)**. Questi due flag vengono gestiti dall'interfaccia medesima, che li setta e li resetta a seguito di eventi di cui si accorge. Si noti che per indirizzare quattro registri sono necessari due fili di indirizzo come ingressi all'interfaccia.

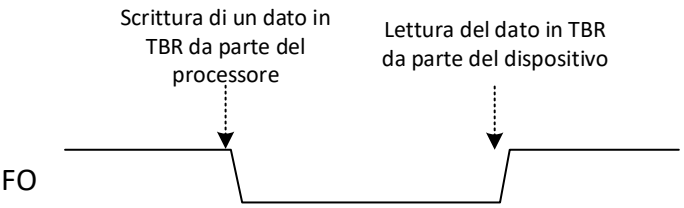
Per quanto riguarda `FI`, il flag è inizialmente a 0. L'interfaccia lo mette ad 1 quando il **dispositivo** scrive un dato in `RBR`, a segnalare che il dato in `RBR` è nuovo. Quando il processore, tramite un'istruzione di `IN`, accede in lettura al registro `RBR`, l'interfaccia porta a 0 il flag `FI`. Un programmatore, quindi, può testare il flag `FI` prima di fare una lettura in `RBR`: se lo trova ad 1, vuol dire che `RBR` contiene un dato nuovo, altrimenti il dato in `RBR` è quello che ha già letto.



Un sottoprogramma Assembler che legge dati "nuovi" da un'interfaccia con handshake e li mette dentro `AL` è il seguente:

```
testFI:    IN RSR_offset,%AL     # Copia in AL il contenuto di RSR
           AND $0x01,%AL         # Evidenzia in AL il contenuto di FI
           JZ testFI             # cicla finché FI vale 0
           IN RBR_offset,%AL     # Copia in AL il contenuto di RBR
           RET                   # Ritorna al chiamante
```

Per quanto riguarda `FO`, il flag è inizialmente a 1. L'interfaccia lo mette a 0 quando il **processore** scrive un dato in `TBR` (tramite un'istruzione `OUT`), a segnalare che il dispositivo non lo ha ancora processato. Quando il dispositivo, con le proprie tempistiche, accede al registro `TBR` per leggere il dato, l'interfaccia porta nuovamente a 1 il flag `FO`. Un programmatore, quindi, può testare il flag `FO` prima di fare una scrittura in `TBR`: se lo trova a 1, vuol dire che può scriverci, altrimenti il dato in `TBR` non è stato ancora processato dal dispositivo.



Un sottoprogramma Assembler che scrive il contenuto di `AL` dentro `TBR` in un'interfaccia con handshake, stando attento che il dispositivo connesso all'interfaccia riesca a processarli, è il seguente:

```
                PUSH %AL              # Salva in pila il contenuto di AL
testFO:         IN TSR_offset, %AL    # Copia in AL il contenuto di TSR
                AND $0x20,%AL         # Evidenzia in AL il contenuto FO
                JZ testFO             # Salta indietro se FO era a 0
                POP %AL               # Ripristina il contenuto di AL
                OUT %AL,TBR_offset    # Immette in TBR il contenuto di AL
                RET                   # Ritorna al chiamante
```

La tecnica di accesso alle interfacce appena descritta prende il nome di **accesso a controllo di programma**. Essa prevede che il processore resta in **attesa attiva**, cioè che cicli (all'interno del sottoprogramma) in attesa che il dispositivo esterno sia pronto. È una tecnica particolarmente inefficiente, perché fa perdere tempo inutilmente al processore. Si farebbe prima se il processore potesse andare avanti per conto proprio, e le interfacce gli notificassero quando i dispositivi sono pronti, "interrompendo" il lavoro del processore. Questa cosa si può fare, prende il nome di **accesso ad interruzione di programma**, ed è una tecnica che vedrete durante il corso di Calcolatori Elettronici. Un'altra tecnica, ancora più efficiente, è quella di **direct memory access (DMA)**, tramite la quale il processore demanda ad un'altra unità (detta *DMA controller*) il compito di trasferire dati tra la memoria e le interfacce, mentre lui va avanti con le sue elaborazioni.

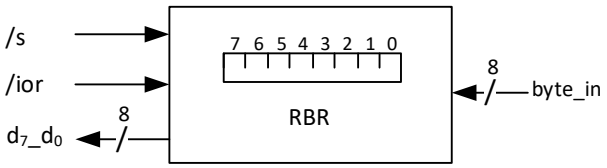
2.1 Interfacce parallele

Prendiamo il **tipo più semplice** di interfaccia **parallela di ingresso**. Un'interfaccia che dà corpo **ad una sola porta,** dalla quale si può **soltanto leggere** (cioè il cui offset può stare soltanto come operando sorgente di un'istruzione di `IN`). Dal punto di vista dei collegamenti con il processore essa avrà bisogno di:

- un segnale di **select**, al quale va l'uscita della maschera, tramite la quale il progettista di hardware decide **quale deve essere l'offset della porta di ingresso dell'interfaccia**
- un filo di **/ior** (e non di /iow, visto che la porta è di sola lettura)
- otto fili di dati
- **nessun filo di indirizzo**, visto che ha una sola porta.

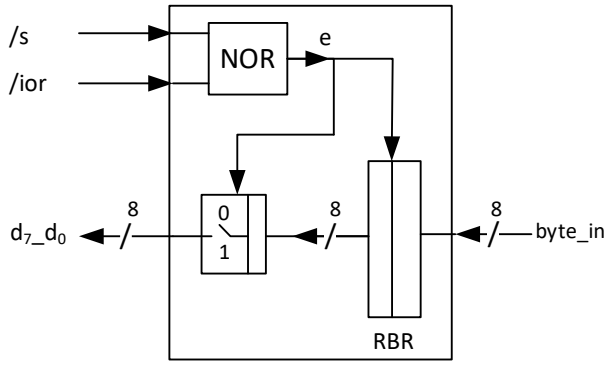
Dal lato del **dispositivo** con il quale, appunto, si interfaccia, ci saranno **8 fili di ingresso**, che chiamiamo `byte_in`, tramite i quali il dispositivo interno fa arrivare i dati. Questi dati saranno inseriti dal dispositivo nel registro `RBR`.

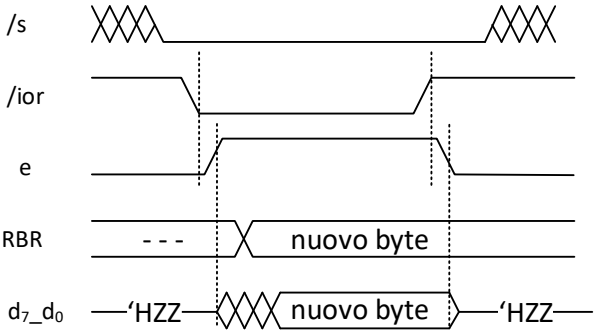
La **visione funzionale** dell'interfaccia è questa:



La **struttura interna** dell'interfaccia sarà la seguente:

- quando /s e /ior sono **entrambi** a zero (il che segna l'avvenuto inizio di un ciclo di lettura) il registro `RBR` **campiona** il dato sui fili di ingresso dal dispositivo, e le tristate entrano in conduzione.
- Dopo un tempo **breve** il dato è buono sul bus, e può essere letto dal processore





Il diagramma di **temporizzazione** conferma che /ior va giù un clock dopo che gli indirizzi si sono stabilizzati. Adesso è chiaro perché: **/s non può ballare con /ior=0**, altrimenti altre interfacce danno un comando di memorizzazione al loro RBR, e questo **non deve accadere**

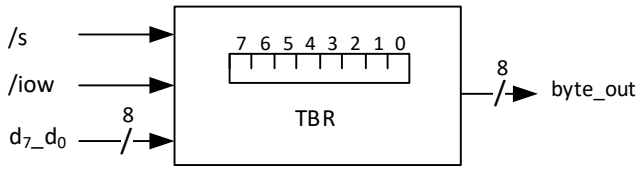
```
module Interfaccia_Parallela_di_Ingresso(d7_d0,s_,ior_,byte_in);
  input    s_,ior_;
  output[7:0] d7_d0;
  input[7:0] byte_in;
  reg[7:0] RBR;
  wire e;  assign e=({s_,ior_}=='B00)?1:0; //e=~(s_|ior_)
  assign d7_d0=(e==1)?RBR:'HZZ;
  always @(posedge e) #3 RBR<=byte_in;
endmodule
```

Dualmente, il **tipo più semplice** di interfaccia **parallela di uscita** è un'interfaccia che dà corpo **ad una sola porta,** nella quale si può **soltanto scrivere** (cioè il cui offset può stare soltanto come operando destinatario di un'istruzione di `OUT`). Dal punto di vista dei collegamenti con il processore essa avrà bisogno di

- un segnale di **select**, al quale va l'uscita della maschera, tramite la quale il progettista di hardware decide **quale deve essere l'offset della porta dell'interfaccia**
- un filo di **/iow** (e non di /ior, visto che la porta è di sola scrittura)
- otto fili di dati
- **nessun filo di indirizzo**, visto che ha una sola porta.

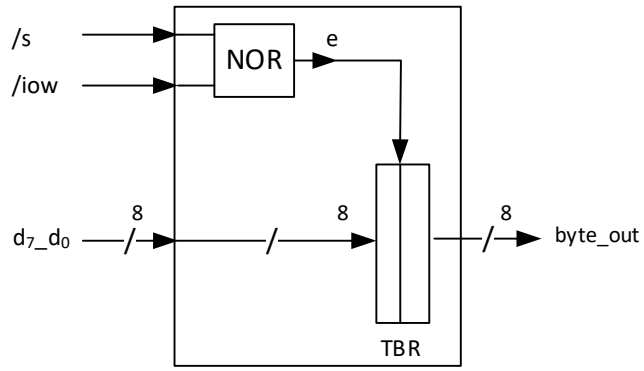
Dal lato del **dispositivo** con il quale, appunto, si interfaccia, ci saranno **8 fili di uscita**, che chiamiamo `byte_out`, tramite i quali l'interfaccia fa arrivare i dati al dispositivo. Questi dati saranno scritti dal processore nel registro `TBR`.

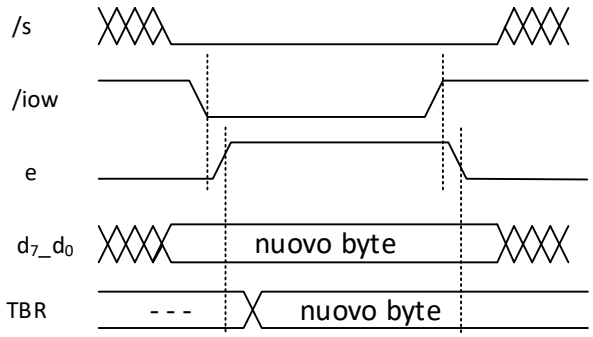
Visione funzionale:



La **struttura interna** dell'interfaccia sarà la seguente:

- quando /s e /iow sono **entrambi** a zero (il che segna l'avvenuto inizio di un ciclo di scrittura) il registro TBR **campiona** il dato sui fili di ingresso dal dispositivo (i dati, infatti, devono essere **già pronti**). **Non ci sono porte 3state**.
- Dopo un tempo **breve** dal campionamento il dato è buono in uscita.

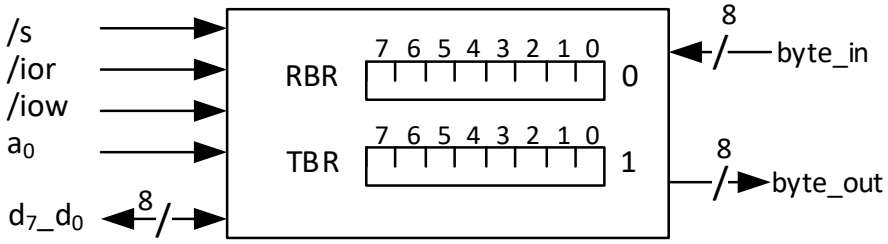


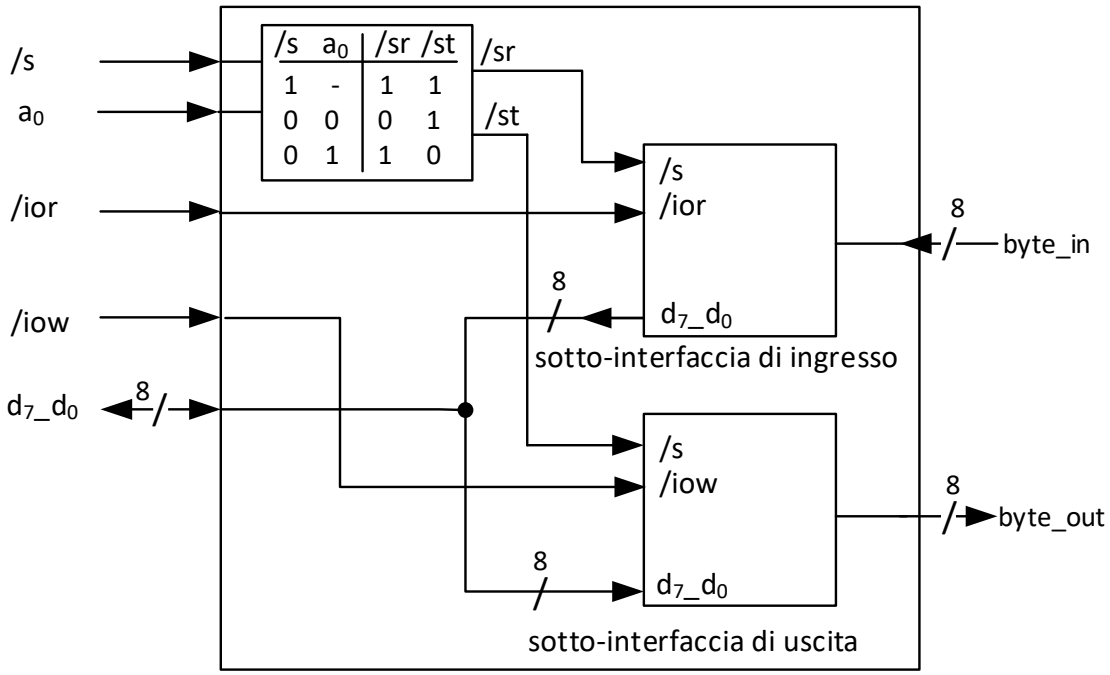


Si osservi nel diagramma di temporizzazione che /iow, stando a quello che succede nella fase di esecuzione dell'istruzione di `OUT`, va giù un clock dopo che gli indirizzi si sono stabilizzati. Infatti, **l'atto di scrittura in TBR avviene in seguito al fronte di <u>discesa</u> di /iow**.

```
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
```

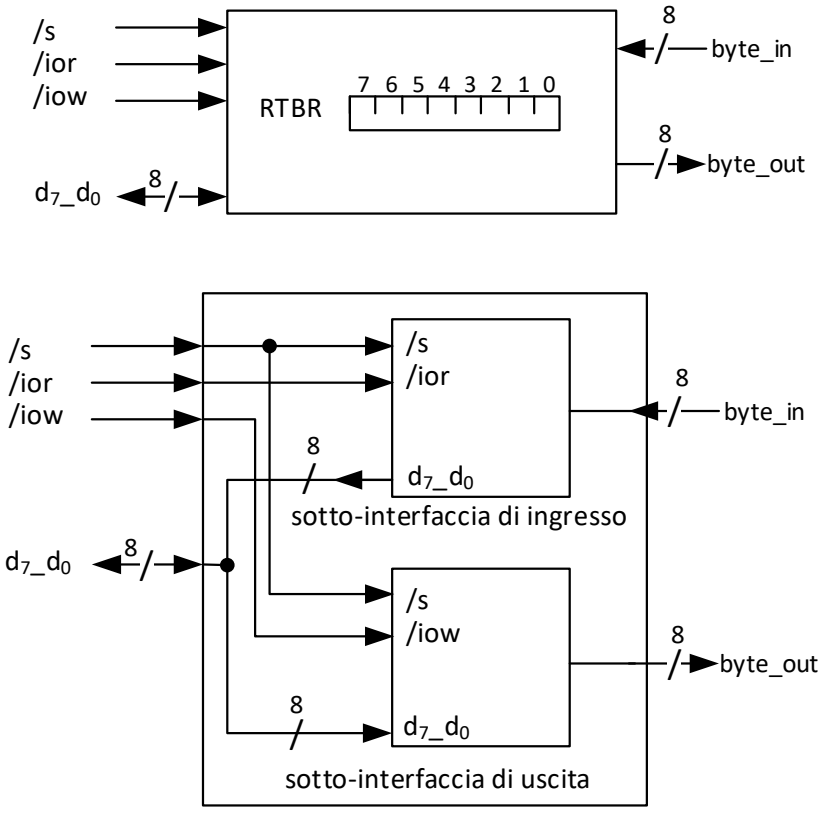
Possiamo anche avere **interfacce di ingresso/uscita**, che contengono sia porte di ingresso che di uscita. Un esempio semplice, in cui c'è una porta di ciascun tipo, è questo.





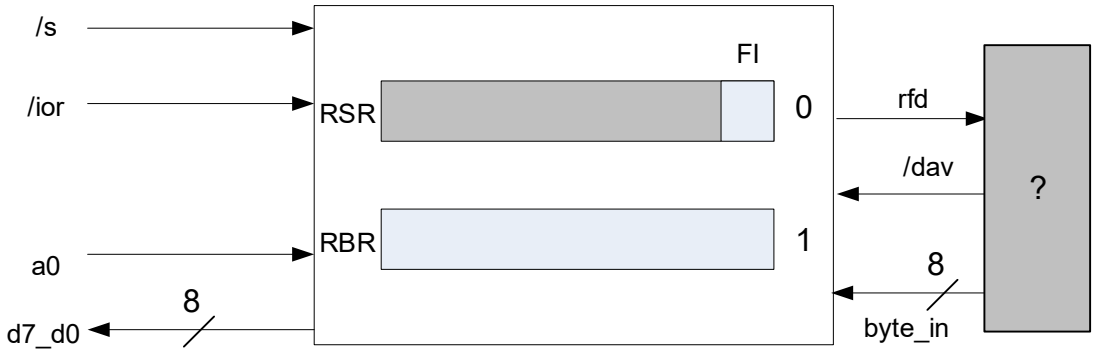
La porta di indirizzo **pari** è accessibile in **lettura**, mentre la porta accessibile in scrittura è quella di indirizzo **dispari**. Ci vuole un po' di logica combinatoria per produrre i due **select**.

Esiste anche la possibilità di montare una coppia di interfacce, **una con una porta di ingresso, e una con una porta di uscita**, in modo tale che il programmatore possa indirizzarle usando **un unico offset**. Le due porte saranno distinguibili in base al tipo di accesso.

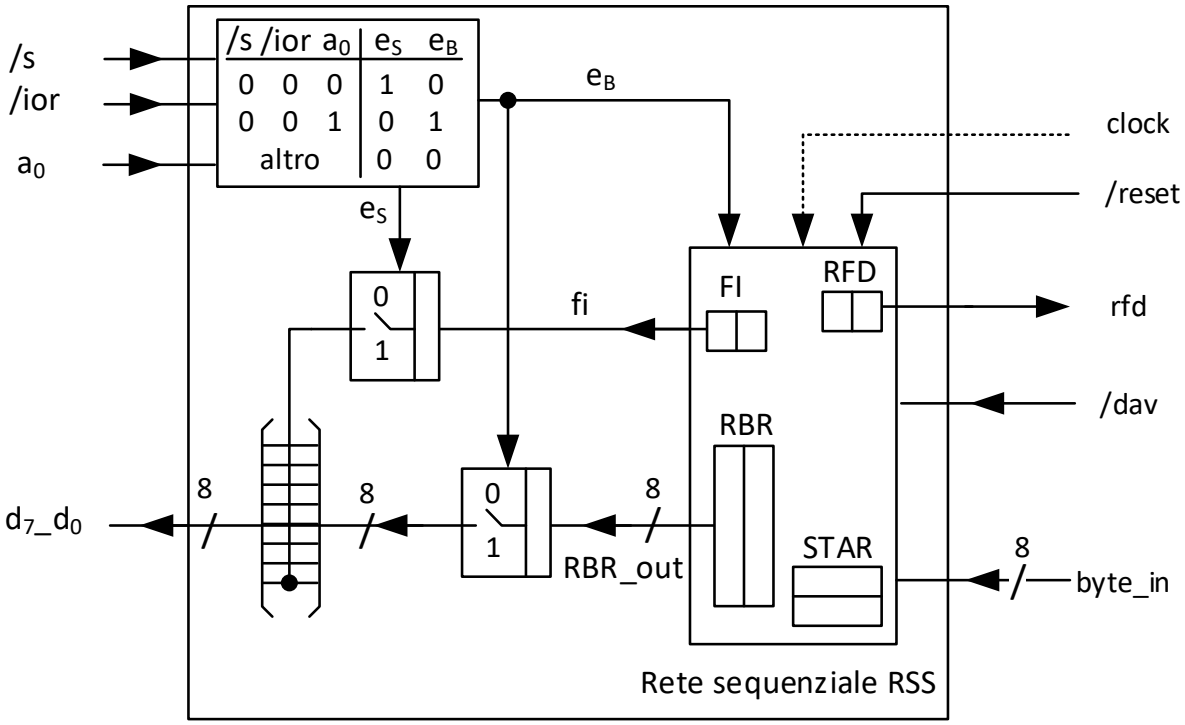


2.1.1 Interfacce parallele con handshake - ingresso

Le interfacce parallele con handshake sono dotate di meccanismi per la sincronizzazione: dal lato del processore, hanno un **flag di ingresso pieno**, che il processore accede a controllo di programma. Dal lato del dispositivo, hanno dei **fili di handshake in più**, uguali a quelli già visti nell'esempio di "produttore e consumatore" fatto a lezione. In questo caso l'interfaccia è un **consumatore** rispetto al dispositivo, che funge da produttore. La visione **funzionale** di un'interfaccia parallela di ingresso con handshake è la seguente. L'interfaccia di ingresso avrà solo /ior (non /iow, che non serve) ed un filo di dati per distinguere gli accessi a RSR e RBR.



Vediamo come è fatta l'interfaccia di ingresso al suo interno:



La RC interna deve, per prima cosa, generare i segnali di abilitazione per le tri-state quando il processore accede in lettura a `RBR` o `RSR`. Le due tri-state (una delle quali ha 8 bit) non sono mai in conduzione contemporaneamente, e sono entrambe in alta impedenza quando non ci sono accessi all'interfaccia. Inoltre, c'è una RSS, che gestisce l'handshake con il dispositivo e setta/resetta il flag

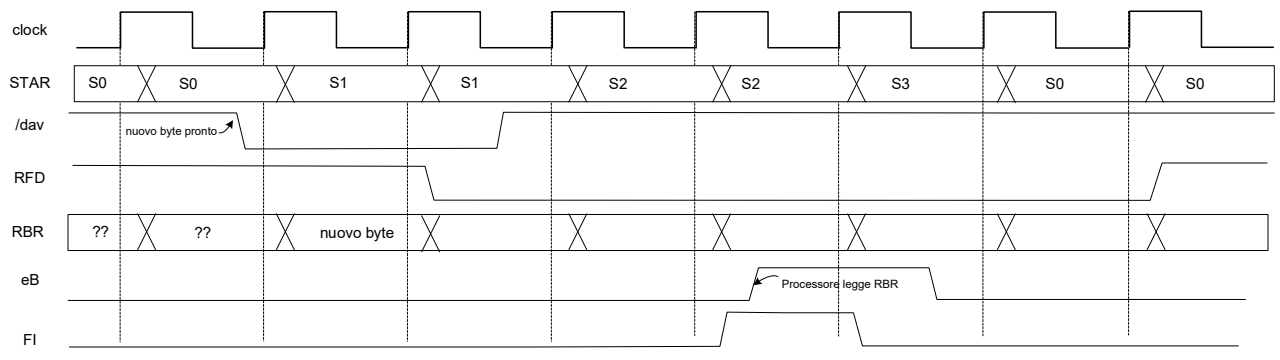
FI. Per poterlo fare deve avere come ingresso anche e_B. Quest'ultimo vale 1 quando il processore vuole leggere RBR (e quindi manda in conduzione la tri-state, in modo tale che il valore di RBR venga messo sul bus).

Il resto dell'interfaccia è una RSS che gestisce l'handshake con il dispositivo, simile a quelle che abbiamo visto per esercizio. La descrizione della RSS è la seguente.

```
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

    //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
```

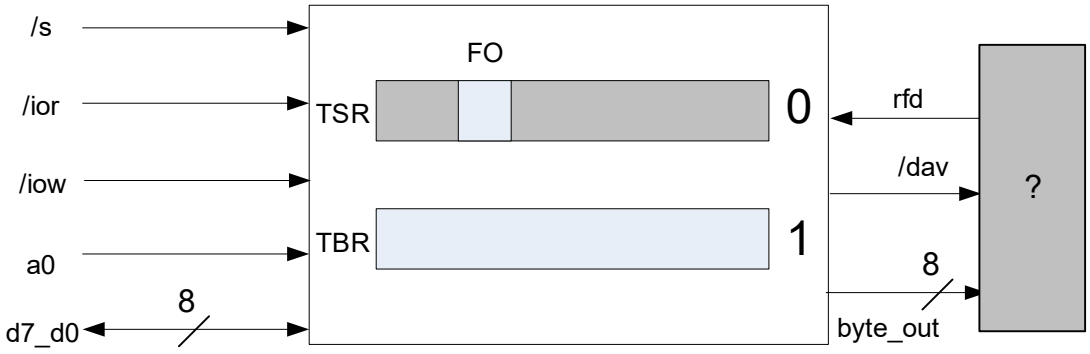
Il clock della RSS va ritardato di qualche ns rispetto a quello del processore, per evitare problemi di campionamento di eB

Una possibile temporizzazione è riportata sotto. In realtà lo stato S3 dovrebbe durare due clock (vedere il ciclo di lettura nell'I/O), ma si assume per semplicità di disegno che duri un clock soltanto.



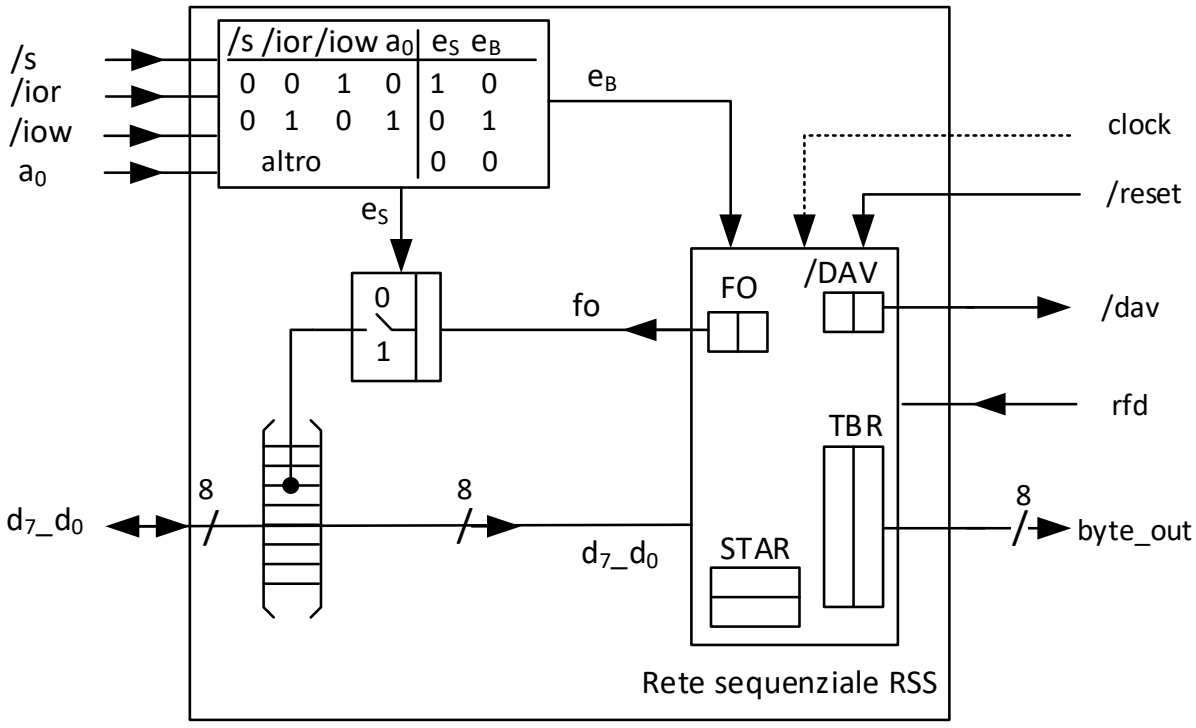
2.1.2 Interfacce parallele con handshake - uscita

La visione **funzionale** di un'interfaccia parallela di uscita con handshake è:



Il flag `FO` vale **uno** quando nel registro `TBR` può essere scritto **un nuovo dato**. Ci vuole un filo di indirizzo, perché ci sono **due** registri, e quindi è necessario distinguerli. Vediamo come è fatta l'interfaccia di uscita al suo interno:

- C'è una rete combinatoria che ha un ruolo analogo a quella dell'interfaccia di ingresso. L'unica differenza è che e_B stavolta non serve ad abilitare una tri-state, perché i dati stavolta vanno nella direzione opposta. Ciononostante, e_B deve entrare nella RSS per far progredire l'handshake.
- La RSS gestisce gli handshake, in maniera duale a prima. Si gestisce prima l'handshake con il processore (che coinvolge e_B e `FO`) e, finito quello, quello con il dispositivo. Si noti che il contenuto di `TBR` balla, ma /dav viene tenuto a 1, quindi il dispositivo non lo può leggere.



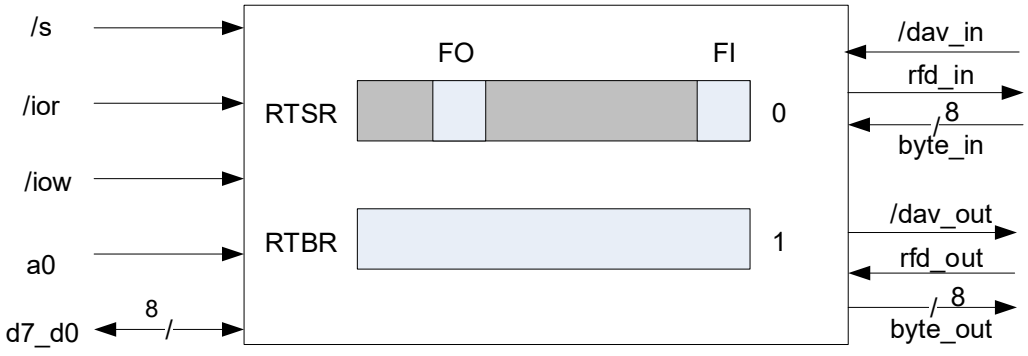
La descrizione della RSS contenuta all'interno è la seguente.

```
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
```

Anche qui il clock della RSS va ritardato di qualche ns rispetto a quello del processore, per evitare problemi di campionamento di eB

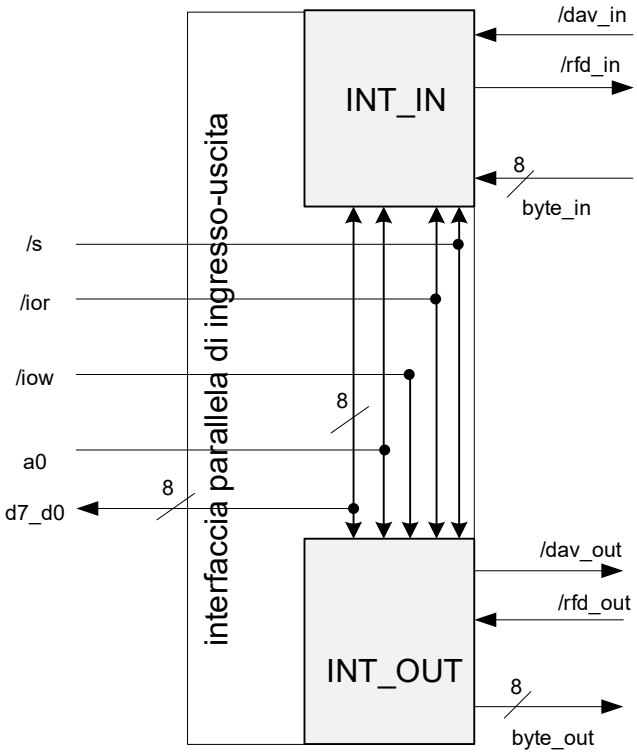
2.1.3 Interfaccia parallela di ingresso-uscita

Interfacce parallele di ingresso e di uscita con handshake possono essere connesse in un'unica interfaccia parallela con handshake di ingresso-uscita come segue.



In questo caso, i due registri `RBR` e `TBR` sono mappati sullo stesso indirizzo interno, e sono acceduti rispettivamente in lettura e scrittura. I due flag `FI` e `FO` danno corpo a due bit in un unico registro di controllo, detto `RTSR`. Dal punto di vista della descrizione interna dell'interfaccia, le cose stanno in questo modo:

Ci sono problemi se **/s=0, /ior=0, a0=0? No**, perché vanno in conduzione le tri-state di INT_IN e INT_OUT, ma i registri `FO` e `FI` stanno **su due fili diversi** del bus dati.



2.2 Interfaccia seriale start/stop

Un'interfaccia seriale è un'interfaccia nella quale la trasmissione dei **singoli bit** avviene in modo seriale. Un byte viene trasmesso "un bit alla volta", partendo (ad esempio) dal bit meno significativo. A dire il vero, **tutte le interfacce** che avete visto (incluse quelle parallele) sono, in qualche modo, seriali, nel senso che, dovendo trasmettere molti byte li trasmetterò in serie. Quello che rende questa interfaccia seriale **speciale** è il fatto che **al suo interno** avviene la serializzazione di unità trasmissive più grandi: l'interfaccia:

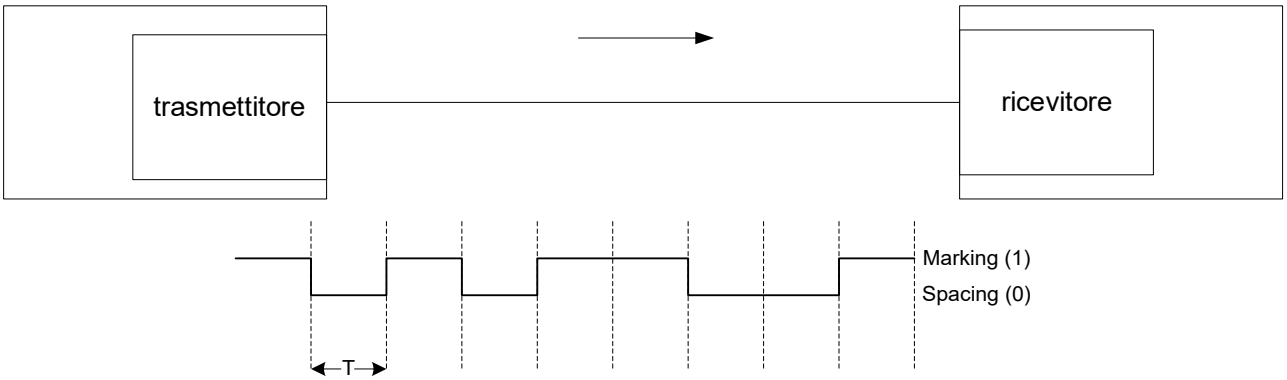
- riceve dal bus **byte** (perché il processore scrive **byte** in opportuni registri di I/O) e trasmette all'esterno sequenze di **bit**,
- riceve dall'esterno sequenze di **bit** e presenta al processore **byte** componendo quelle sequenze di bit in un registro che si possa leggere.

Un'interfaccia *parallela* riceve invece **byte** dal processore e trasmette **byte** all'esterno, o riceve byte dall'esterno e presenta byte al processore.

Un PC ha di norma più di una interfaccia (**porta**) seriale. Ad esse possono essere connessi, ad esempio, **modem** (esterni) e (un tempo) **mouse**. Parecchi dispositivi che hanno del **firmware** configurabile

sono, appunto, configurabili tramite un'interfaccia **seriale** (ad esempio i **router**, cioè quei dispositivi che inoltrano il traffico di rete). Un tempo, i calcolatori erano grossi elaboratori centrali (**mainframe**) che venivano connessi a terminali semplici tramite, appunto, linee **seriali** (è infatti per questo che sono state inventate).
Le interfacce seriali che sono sul PC sono piuttosto complesse. Ne vediamo una versione **semplificata**, che comunque contiene alcuni concetti importanti.

Prima di descrivere l'interfaccia seriale, diamo uno sguardo a come avviene la **comunicazione seriale** tra due entità.



Da un punto di vista **fisico**, il mezzo trasmissivo sul quale esce l'informazione si presenta (nella sua versione più semplice) come un insieme di **due linee**: una linea di **massa,** che funge da riferimento**,** ed una linea che porta una **tensione** riferita a massa. La massa non porta informazione, quindi non viene disegnata, né contata tra le variabili logiche di ingresso o uscita. Sono leciti due valori sulla linea:

- **marking**, cioè **1 logico**,
- **spacing**, cioè **0 logico**.

La trasmissione di un bit consiste nel tenere la linea in uno stato di marking o spacing per un determinato tempo T, detto **tempo di bit**.
Un insieme di bit scambiato si chiama **trama** o **frame.** Per adesso supponiamo che una trama sia costituita da **un byte**, trasmesso dal bit meno significativo al più significativo.

Affinché il mezzo trasmissivo possa sostenere trasmissione **in entrambe le direzioni contemporaneamente** (**full duplex**, mentre **half-duplex** indica la trasmissione da un lato solo), sono necessari **tre fili**, due dei quali portano le tensioni riferite a massa (uno per direzione).

Visto che le linee seriali sono fatte in questo modo, esiste un problema fondamentale: **come si fa a sincronizzare un trasmettitore ed un ricevitore?** Detto in altre parole: come fa il ricevitore a sapere che il permanere della linea a 1 indica che ci sono 2 (3, 4, …) bit a 1 consecutivi?

Per risolvere questo problema avete visto – nel corso delle lezioni, ed in ambiti differenti – due tecniche, nessuna delle quali è applicabile a questo campo:

- **Condividere un clock** (è quello che si fa, ad esempio, sul bus. Tutti i dispositivi vedono un clock comune).
- Aggiungere delle **linee dedicate alla sincronizzazione** (tipo ***rfd***, ***/dav***), cioè linee che non portano di per sé informazione, ma servono a dire quando l'informazione presente su altre linee è valida.

Per implementare una di queste soluzioni ci vogliono più di due fili (o un filo di clock, o dei fili di handshake), e noi vogliamo usarne soltanto due. Il problema si risolve in questo modo:

- entrambi i lati devono **concordare sul tempo di bit *T*** (ovviamente **prima** che la comunicazione abbia inizio).
- Entrambi i lati della comunicazione devono concordare sul **numero di bit** di cui si compone una trama. Tipicamente, questo numero va da **5 ad 8**.
- Una **trama** deve essere resa **riconoscibile**. In particolare, è necessario che entrambi i lati concordino sul modo di rendere noto che **una trama è iniziata**.
    - Si fa così: la linea sta, normalmente, in uno stato di **marking**. Quando voglio iniziare la trasmissione di una trama, la porto nello **stato di spacing**. Ciò significa che **ogni trama inizia con il bit 0**, il quale bit 0 **non è un bit informativo della trama**, ma è un bit che serve soltanto a dire "la trama è iniziata". Infatti, si chiama **bit di start**.
    - Analogamente, quando ho trasmesso **l'ultimo** bit di una trama, devo riportare la linea in uno stato di **marking** per **<u>almeno</u> un tempo di bit** (bit di **stop**)**,** in modo tale che possa poi iniziare una nuova trama con una transizione marking/spacing.

<u>Osservazione:</u> se voglio trasmettere trame di **un byte** (8 bit), questo procedimento rende necessario **utilizzare almeno due bit in più**, che non hanno nessun significato informativo, ma servono a rendere possibile una corretta decodifica dell'informazione. Ciò vuol dire che, se posso trasmettere un certo numero di bit al secondo $\boldsymbol{x = 1/T}$ sulla linea, non posso trasmettere delle trame lunghe un byte ad una velocità **netta** maggiore di $x \cdot 8/10$ bit al secondo.

La mancanza di linee di sincronizzazione **si paga** sotto forma di incapacità di sfruttare interamente la velocità della linea di trasmissione. Per massimizzare l'efficienza, allora, **sembra che convenga mandare un numero di bit estremamente elevato in una trama.** Se ogni n ne devo inviare $n + 2$, tanto più grande è n, tanto meglio funziona questo protocollo.

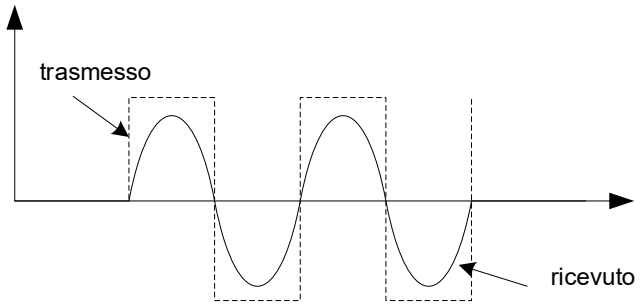
C'è però un problema: il clock del trasmettitore e del ricevitore **<u>non saranno mai identici</u>**. Per quanto possa cercare di farli simili, avranno sempre una leggera discrepanza di frequenza, che di norma è nell'ordine di qualche punto percentuale. Questo implica che, se T è il tempo di bit secondo il trasmettitore, il ricevitore misurerà un tempo di bit $\boldsymbol{T \pm \Delta T}$, con $\boldsymbol{\Delta T}$ **non nullo.**

Affinché sia garantita una corretta ricezione dei bit, è indispensabile che la discrepanza tra i due tempi sia contenuta. In particolare, è necessario che il ricevitore non accumuli un ritardo (o anticipo) tale da "uscire dal bit", andare cioè a campionare il bit sbagliato.

Visto che **non conosco** il segno davanti a $\boldsymbol{\Delta T}$, mi conviene **cercare di campionare i bit sulla linea il più vicino possibile alla metà.** In questo modo, minimizzo il rischio di uscire dal bit. La transizione da **marking** a **spacing** serve a sincronizzare i due dispositivi: il ricevitore sa quando la trama comincia, e sa quanto durano i bit (tempo di bit T). Per sapere **quali bit** vengono trasmessi dal trasmettitore, un ricevitore deve:

- aspettare $3/2\ T$ da quando vede la linea transisce da marking a spacing la prima volta
- campionare il valore della linea
- aspettare nuovamente T, e così via per tutti i bit utili della trama.

In questo modo il ricevitore finisce per sentire ogni bit utile "nel mezzo". Questo garantisce che, se la linea introduce dei disturbi (attenuazioni dovute alla resistenza interna dei conduttori, variazioni della pendenza nei fronti di salita e di discesa dovute alle **reattanze** dei conduttori, ed altro), vado a campionare il segnale nel punto più opportuno.



Se devo poter ricevere e decodificare correttamente n bit tra due **segnali di sincronizzazione successivi** (bit di start), **cercando di campionarli nel mezzo**, è necessario che: $n \cdot \Delta T \leq \frac{T}{2}$, cioè che

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

Ciò significa che l'errore relativo che si può tollerare sul clock è **inversamente proporzionale** al numero di bit che devono essere trasmessi tra due segnali di sincronizzazione. Per $n = 10$, abbiamo che il limite superiore è del **5**%. Quindi, non solo ho bisogno di **sincronizzare** il trasmettitore con il ricevitore, ma ho anche bisogno di assicurarmi che la **precisione** dei due clock è tale da consentire la corretta ricezione di un congruo numero di bit (pari a quelli che devo trasmettere in una trama):

Non si può aumentare a dismisura il numero di bit trasmessi in una trama, perché altrimenti si creano problemi di decodifica dovuti all'imprecisione dei clock. Visto che i clock sono, per motivi

costruttivi, **non troppo precisi**, c'è bisogno di trasmettere trame "non troppo lunghe", e di risincronizzarsi ogni volta con le transizioni marking/spacing.

L'inverso del tempo di bit si chiama **bitrate**, ed è misurato in bit al secondo. In genere, va da poche decine alle decine (ora anche centinaia) di migliaia.

Concludendo, in una trama **non tutti i bit sono utili**. In particolare abbiamo:

- bit necessari alla sincronizzazione (start, stop);
- bit di informazione (da 5 a 8);
- eventualmente altri (bit di parità, non fanno parte dell'esempio)

Tutto questo assomma a dire che è necessario dare un **formato** ad una trama di bit, cioè stabilire delle **regole univoche** (e note ad entrambi i lati della comunicazione) perché la comunicazione abbia luogo.

Tutto quanto raccontato finora è infatti parte di uno **standard**, detto **EIA-RS232C** (EIA sta per Electronic Industries Association, ed è un ente di standardizzazione, come ISO, ANSI, etc.), sviluppato all'inizio degli anni '60. Uno standard fissa delle regole uguali per tutti per eseguire un certo compito, con ovvi benefici:

- **garanzia di funzionamento**
- **interoperabilità** tra realizzazioni indipendenti della stessa funzione

Lo standard copre:

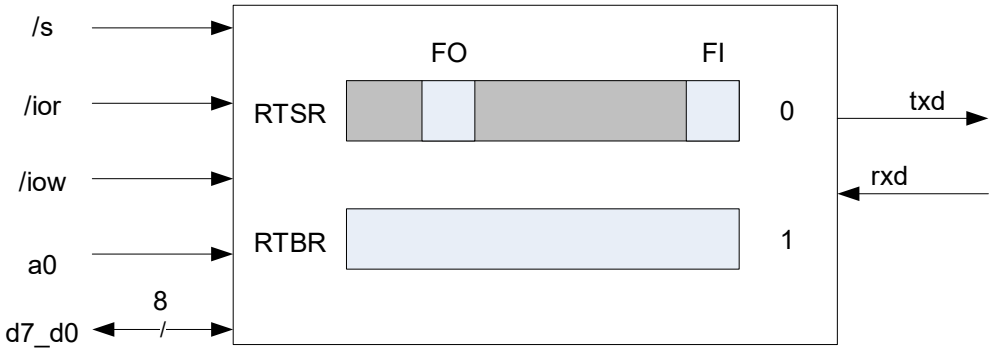
- voltaggi elettrici dei segnali
- temporizzazione
- funzione dei segnali
- formato e piedinatura dei connettori
- formato delle trame
- protocollo di comunicazione

Di tutto questo noi vediamo soltanto alcuni aspetti.

Ad esempio, le **tensioni** che vengono fatte corrispondere a 1 e 0 **non sono quelle caratteristiche di una rete logica** (0 e 5 volts). In particolare, l'1 logico corrisponde ad una tensione **negativa,** compresa tra -3V e -25V, mentre lo 0 logico corrisponde ad una tensione **positiva**, compresa tra +3 e +25.

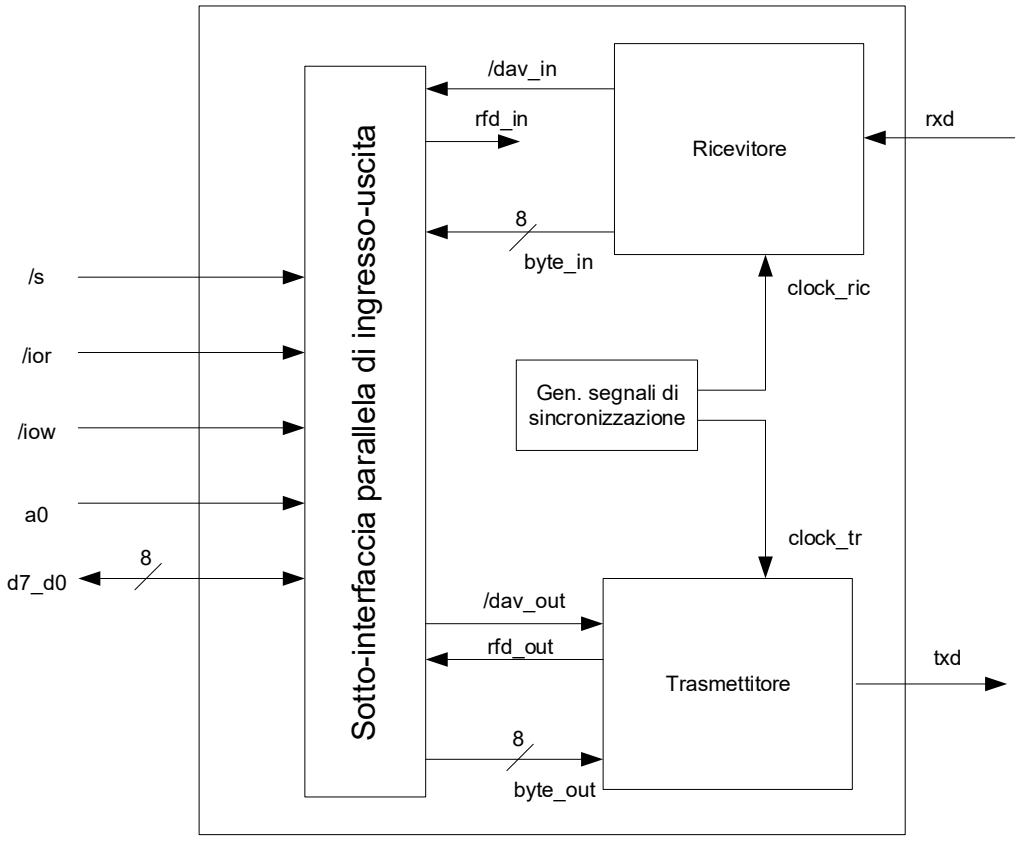
2.2.1 Visione funzionale e struttura interna dell'interfaccia

Dal punto di vista funzionale, un'interfaccia seriale di ingresso/uscita è simile ad un'interfaccia parallela di ingresso/uscita. Per il programmatore è addirittura identica. Abbiamo infatti:



- 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`**.
- Un registro `RTBR` ad 8 bit che serve per contenere i dati da trasmettere o quelli ricevuti.

Dal punto di vista della struttura interna:



C'è una **sottointerfaccia parallela** di I/O **con handshake**, che colloquia con due **due reti sequenziali sincronizzate**, dette trasmettitore e ricevitore. Cominciamo a descriverne una. Si parte dal trasmettitore.

2.2.2 Descrizione del trasmettitore

Il trasmettitore:

- accetta un nuovo byte dalla sottointerfaccia parallela di uscita, con la quale ha un handshake.
- trasmette tutti i bit di quel byte sul mezzo trasmissivo tramite il filo *txd*.

Il trasmettitore pertanto ha bisogno di alcuni **registri**:

- **`TXD`**, registro ad **1 bit** che contiene il bit da trasmettere ad un dato istante.
- **`RFD`**, registro ad **1 bit** che sostiene il segnale di uscita dell'handshake.
- **`BUFFER`** è il registro nel quale tengo tutto il byte da trasmettere. Pertanto, deve essere largo **almeno** 8 bit. In realtà, conviene dimensionarlo in modo da **contenere l'intera trama**, compresi i bit di start e di stop (nell'esempio ignoriamo il bit di parità). Così facendo, nella macchina a stati la trasmissione del bit di start, del bit di stop e di un qualunque bit informativo potranno aver luogo all'interno del medesimo ciclo. Ottengo quindi una descrizione più semplice.
- **`COUNT`** è un registro nel quale tengo il **conto** dei bit ancora da trasmettere. Visto che i bit da trasmettere sono 10, il registro deve essere dimensionato in maniera da contenere il numero 10, cioè deve avere **4 bit**.

Posso pensare (faremo questa ipotesi) che il **trasmettitore** sia pilotato con un clock uguale al tempo di bit (anche se normalmente **non è così**). In realtà, nelle interfacce seriali **vere (**più complesse di queste**),** visto che il tempo di bit T è configurabile via software (scrivendo nelle interfacce in un opportuno registro di controllo), è abbastanza ovvio che sia il ricevitore che il trasmettitore avranno un clock interno sufficientemente maggiore del **minimo** tempo di bit che può essere impostato.

Le **ipotesi al reset** sono:

- /dav a 1, dalla parte dell'interfaccia
- da parte del trasmettitore, dovrò tenere `RFD` a 1 e la linea di uscita in **marking**.

Come esercizio simile ad un esercizio di esame, descriviamo il trasmettitore ed il ricevitore.

S0: A **regime,** il trasmettitore si trova in uno stato iniziale S0 in cui ha, `rfd=1`, `txd=1` (la linea è infatti in stato **marking**), l'ingresso */dav_out* ad 1, il contenuto dei `COUNT` e `BUFFER` non significativo. La prima mossa la fa la sottointerfaccia parallela, iniziando l'handshake:
porta a zero la linea */dav_out* e presenta sui fili *byte_out* gli 8 bit informativi della trama da trasmettere. Quando ciò accade, mi devo **muovere** dallo stato iniziale S0 per iniziare le operazioni. Mi conviene, come già visto altre volte, campionare il byte in ingresso sui fili *byte_out*. Anzi, meglio: devo preparare una **trama di 10 bit** con un bit di start, **quegli otto bit**, un bit di stop. Posso anche inizializzare il registro **`COUNT`** al numero di bit che devo trasmettere, con l'intenzione di trasmettere un nuovo bit ad ogni ciclo di clock.

S1: Mi muovo quindi in uno stato S1, nel quale eseguo **un ciclo**, fatto come segue:

- metto in `TXD` il bit meno significativo di `BUFFER`
- mettere `RFD` a zero per far avanzare l'handshake

- shifto a destra **tutto il contenuto di `BUFFER`**, in modo tale che, ad ogni nuovo ciclo, il bit meno significativo sia quello adiacente a quello che ho appena trasmesso. Già che devo inserire qualcosa nel bit più significativo, ci metto il valore di riposo della linea (marking), anche se la cosa non ha molta importanza.
- Decremento il **`COUNT`**, in modo tale da tener conto dei bit ancora da trasmettere.

Ciclo in questo stato finché non ho trasmesso **tutti i bit**. La condizione di uscita dal ciclo dipende dall'inizializzazione. Come già visto, per fare il numero di iterazioni corretto devo uscire con `COUNT==1`.

S2: A questo punto, esco dallo stato S1 e, prima di ritornare nello stato di riposo iniziale, devo **attendere che *dav_out*** sia tornato a 1, altrimenti tornerei in uno stato in cui metto `RFD` ad 1 senza essere sicuro che l'handshake sia terminato correttamente.

```
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
```

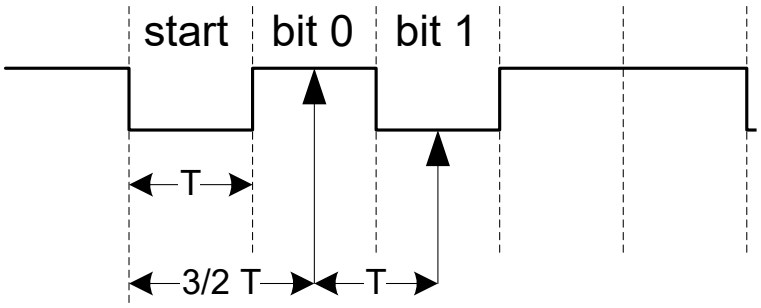
2.2.3 Descrizione del ricevitore

Vediamo adesso il ricevitore. Ci sono alcune differenze significative:

- il ricevitore **non è in grado di controllare il flusso dei dati in ingresso** dalla linea seriale. Pertanto, non ha nessun senso che gestisca un handshake completo con la sottointerfaccia parallela. Più in dettaglio, se la sottointerfaccia parallela **non fosse in grado di accettare** un nuovo dato, sarebbe un problema suo. Il dato verrebbe sovrascritto da una nuova trama di bit. Pertanto, il filo di ***rfd*_in** non è collegato al ricevitore. Infatti, se quest'ultimo memorizza in un unico registro i

dati ricevuti, **o vengono letti in tempo, o verranno sovrascritti (overrun)**, in quanto non esiste modo di bloccare il trasmettitore dall'altra parte del filo.

- Abbiamo supposto che il **trasmettitore** avesse un **clock con periodo pari al tempo di bit** (o, detto in altro modo, **di frequenza pari alla bit rate del canale**). Vediamo cosa possiamo dire per il **ricevitore.** Il ricevitore:
  - Deve andare a campionare i bit (per quanto possibile) a **metà** del tempo di bit.
  - Capisce che la trama è iniziata quando vede un fronte di discesa da marking a spacing.
  - Quindi, deve campionare il **primo** bit dopo $\mathbf{3/2}\ \boldsymbol{T}$ da quando vede l'inizio della trama, e ciascun successivo bit dopo un tempo T.



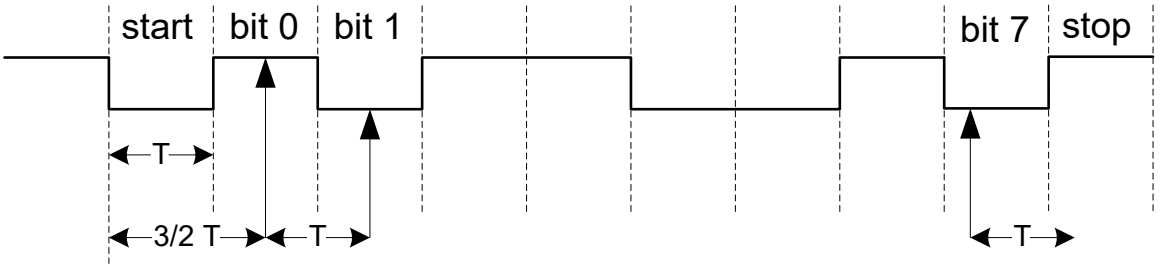
Se anche il ricevitore avesse **un clock di periodo $\boldsymbol{T}$ questo non sarebbe possibile**. Al massimo, il clock del ricevitore dovrà avere un **periodo pari a $\boldsymbol{T/2}$**, cioè **frequenza doppia rispetto alla bitrate del canale**. Esistono però altre considerazioni che rendono conveniente fare il clock del ricevitore **ancora più veloce**. Il ricevitore è una **macchina sincrona**, e quindi cambia stato ad ogni fronte di clock. L'evento che dà il via al processo di ricezione di una trama è il **fronte di discesa** di un segnale di ingresso. Questo evento può avvenire in un istante qualunque di un ciclo di clock, e quindi è reso noto al ricevitore con **un'incertezza temporale** che non può essere inferiore al periodo del clock. Visto che lo scopo del ricevitore è cercare di **stare nel bit** ad ogni campionamento, tanto minore è quest'incertezza, tanto più tranquillo sono per i campionamenti successivi. Quindi, avere un clock più veloce significa avere una **miglior stima** dell'istante in cui inizia il bit di start.

Nella costruzione della nostra interfaccia, facciamo l'ipotesi che il periodo del clock è pari ad **un sedicesimo del tempo di bit**.

Detto questo, vediamo come è fatto il ricevitore. Ci vorranno dei registri

- **`DAV_`** che sostiene il corrispondente segnale
- **`BUFFER`** è il registro nel quale tengo la parte di trama ricevuta fino a questo momento. Pertanto, deve essere largo **almeno** 8 bit. In questo caso, non mi interessa di dimensionarlo per tenere l'intera trama. Anche qui, come per il ricevitore, mi converrà **far scorrere** i bit all'interno del registro ogni volta che ne aggiungo uno.

- **`COUNT`** è un registro nel quale tengo il **conto** dei bit **buoni** ancora da ricevere. Visto che i bit da ricevere sono 8, il registro deve essere dimensionato in maniera da contenere il numero 8, cioè deve avere **4 bit**.
- **`WAIT`**: serve a contare gli stati che devo attendere tra due campionamenti successivi del bit in ingresso. Abbiamo detto che il clock del ricevitore è **16 volte più veloce** del tempo di bit. Ciò significa che tra un campionamento ed il successivo dovrò attendere un certo numero di clock, e quindi mi serve un registro per tenere il numero di cicli che devo aspettare. Attenzione che la quantità di tempo che devo attendere **non è sempre la stessa**:
  - Quando vedo il fronte di discesa del bit di start, devo attendere **un bit e mezzo,** cioè **24 cicli di clock**, prima di campionare il primo bit utile
  - Tra un bit utile ed il successivo devo attendere **un bit, cioè 16 cicli di clock**.
  - Campionato l'ultimo bit utile, dovrei attendere **almeno mezzo bit** (**8 clock),** nell'ipotesi in cui il tempo di bit del trasmettitore e del ricevitore fossero **identici.** In realtà, per via delle discrepanze fisiche nei due clock, quando campiono l'ultimo bit **non so dove mi trovo** all'interno del bit, per cui mi conviene attendere **almeno un intero bit (16 clock**) perché così sono sicuro che, se il clock del trasmettitore e del ricevitore sono sufficientemente vicini, finisco all'interno del bit di stop.



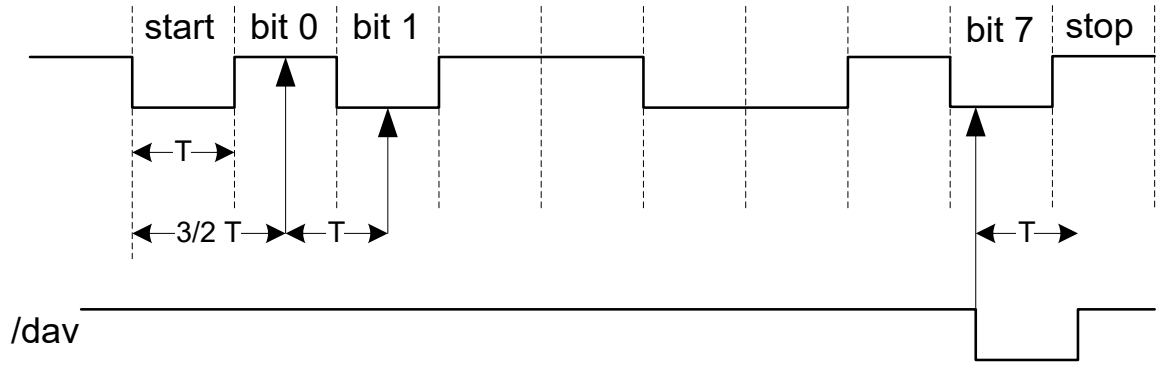
Pertanto, il registro `WAIT` dovrà contenere un numero simile a 24, cioè dovrà essere almeno su 5 bit.

Cosa faccia il ricevitore è adesso abbastanza chiaro.

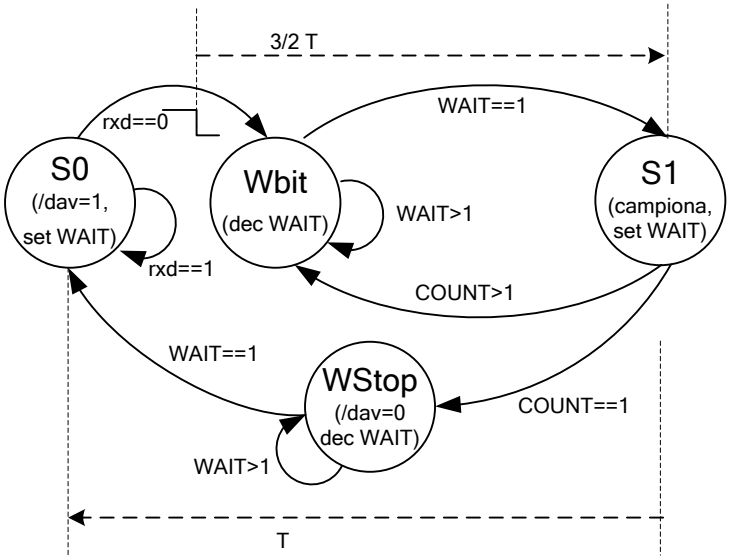
- inizialmente si trova in uno stato in cui non fa niente, da cui esce quando la linea passa da marking a spacing;
- **aspetta** un bit e mezzo;
- **campiona** la linea, aggiornando il conto dei bit letti;
- **aspetta un bit** e rifà la stessa cosa;
- quando ha letto 8 bit, attende ancora un bit e poi ritorna nello stato iniziale.

Per quanto riguarda l'**handshake** con la sottointerfaccia parallela di ingresso, c'è da stabilire soltanto quando `/dav` debba essere messo a zero e ad uno.

- i dati saranno validi quando avrò letto **tutti i bit utili,** quindi dopo il campionamento del bit 7. A quel punto la sottointerfaccia parallela di ingresso può leggere il contenuto del buffer. Devo quindi portare a zero `/dav`.
- dopo che ho atteso l'arrivo del bit di stop, in teoria ogni clock è buono perché la linea si abbassi ad indicare l'inizio della successiva trama. Quindi, a quel punto o il byte ricevuto è stato letto, o pazienza. Devo comunque riportare `/dav` a 1.



Proviamo a fare un diagramma a stati di quello che succede, coerente con quello che abbiamo appena detto.



- Inizialmente sono in uno stato **S0**, nel quale `/dav` è pari ad 1. Esco da questo stato quando vedo la transizione marking-spacing su `rxd`.
- Dopo aver visto la transizione, non posso campionare subito il bit. Devo infatti attendere $3/2\,T$ dalla transizione. Ho quindi bisogno di uno stato di **attesa `Wbit`**, nel quale perdo tempo. Faccio scorrere il clock decrementando `WAIT`, ed esco dallo stato quando `WAIT` vale (ad esempio) 1. Ovviamente, `WAIT` dovrà essere stato inizializzato **prima** di entrare in questo stato. Lo posso, ad esempio, fare in S0. **A quanto lo dovrò inizializzare?** Lo sapremo dopo aver finito **la descrizione,** guardando le temporizzazioni.

- S1 è lo stato in cui **campiono** un bit. Campionare un bit vuol dire **shiftare** a destra il vecchio contenuto del registro `BUFFER`, inserendo come bit più significativo il bit appena campionato (ricordare che lo standard richiede che i bit vengano trasmessi dal meno significativo al più significativo). A questo punto:
  - Se ho letto **meno di 8 bit** devo attendere *T* e ripetere l'operazione. Lo faccio, come al solito decrementando un contatore di `WAIT` e tornando indietro allo stato precedente in modo tale che tra due successivi campionamenti siano passati 16 cicli di clock. Il registro `WAIT` dovrà essere inizializzato prima di entrare in `Wbit`, e quindi in S1.
  - Se ho letto **8 bit**, vado in un nuovo stato, in cui pongo */dav* a zero ed attendo il bit di stop

  Per sapere quanti bit ho letto, quando sono **in S1 decremento `COUNT`**. `COUNT` dovrà quindi essere già stato inizializzato quando arrivo in S1, quindi l'inizializzazione di `COUNT` la devo fare in S0.
- Una volta posto `/dav` a zero, non lo posso riportare subito ad 1. Devo aspettare, e lo faccio decrementando `WAIT` opportunamente. Al solito, l'attesa dovrà esser tale per cui tra il campionamento del bit 7 ed il ritorno in S0 (in cui vado ad ascoltare se inizia una nuova trama) siano passati 16 cicli di clock. `WAIT` va inizializzato prima di entrare in `Wstop`.

A questo punto, resta da strigare la condizione su `COUNT`, che lì è stata indicata genericamente come "meno di 8 bit" "8 bit". Se la condizione su `COUNT` è

```
STAR<=(COUNT==1?Wstop:Wbit)
```

Allora devo inizializzare `COUNT` ad 8.

Per quanto riguarda l'inizializzazione di `WAIT`: con una descrizione scritta in questo modo, se lo inizializzo a k e lo testo ad 1, farò $k - 1 + 1 = k$ iterazioni negli stati `Wbit` e `Wstop`. Visto che l'attesa deve durare in tutto 24 e 16 clock, è necessario che inizializzi `WAIT` a 23 e 15, rispettivamente (devo contare anche il clock perso in S0 ed S1).

```
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
```

2.3 Conversione analogico/digitale e digitale/analogica

Finora abbiamo visto soltanto interfacce che consentono a due calcolatori di dialogare tra loro. Se in un calcolatore devono entrare/uscire delle informazioni da/verso il resto del mondo, è necessario che queste vengano **convertite** dalla forma in cui si trovano ad una forma comprensibile per il calcolatore. In particolare:

- nel mondo fisico, l'informazione è di norma associata a grandezze **analogiche**, che variano "con continuità" (nel senso che la granularità di variazione è – a livello atomico – troppo piccola perché se ne possa tener conto).
- All'interno del computer, le informazioni sono associate a **stringhe di bit**, cioè a grandezze **digitali**, che variano in modo discreto.

Si pone quindi il problema di realizzare **conversioni da analogico a digitale** per far entrare informazioni, e da **digitale ad analogico** per farle uscire. La grandezza analogica che consideriamo nel nostro caso è una **tensione.** Convertiremo questa tensione in un **numero (naturale o intero) in base 2**, e viceversa.

La tensione $\boldsymbol{v}$ da convertire sarà su una scala di FSR volts (**Full-Scale Range**). Il numero $\boldsymbol{x}$ nel quale sarà convertita è su N bit. Valori tipici sono $N = 8{,}16$, $FSR = 5 \leftrightarrow 30$. A seconda dell'interpretazione del numero e della tensione, posso distinguere:

- conversione **unipolare**: $v \in [0, FSR],\ x \in [0, 2^N - 1]$
- conversione **bipolare**: $v \in \left[-\frac{FSR}{2}, \frac{+FSR}{2}\right],\ x \in [-2^{N-1}, +2^{N-1} - 1]$

Definiamo $K = \frac{FSR}{2^N}$, **costante di proporzionalità** tra i due intervalli. Una conversione *ideale* sarebbe $v = K \cdot x$. In realtà, dovremo accontentarci di $|v - K \cdot x| \leq err$, con err detto **errore di conversione**, che auspichiamo essere il più piccolo possibile. Gli errori di conversione sono dovuti a:

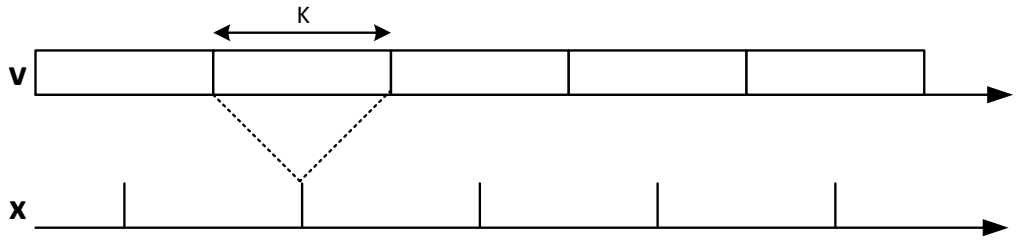
- imprecisioni a livello **circuitale**: i convertitori sono circuiti con resistenze, fili, reattanze, che non si comportano in maniera **ideale**. Ci sarà un'imprecisione dovuta alla non idealità dei componenti, difficilmente eliminabile. Questo tipo di errore è presente sia nella conversione D/A che in quella A/D, e viene detto **errore di non linearità**.
- Quantizzazione. Nella conversione A/D (e soltanto in quella), devo convertire una grandezza **continua** in una **discreta**. Facendo questo si perde dell'informazione a causa dell'**arrotondamento**. Questo tipo di errore si chiama **errore di quantizzazione**.

Facciamoci un'idea del **limite massimo tollerabile** per questi errori.

L'errore di **non linearità** deve essere più piccolo di $\frac{K}{2}$. Se così non fosse, visto che la formula può essere riscritta come:

$$v \in [K \cdot x - err, K \cdot x + err]$$

Vorrebbe dire, ad esempio, che nella conversione **da digitale ad analogico,** gli intervalli centrati in due numeri consecutivi sarebbero **parzialmente sovrapposti**, il che comporterebbe che potrei convertire un numero più grande in una tensione più piccola e viceversa.



L'errore massimo di **quantizzazione** è indipendente dalla natura del convertitore (A/D). Data una costante K, è pari a $\frac{K}{2}$. Infatti, se divido il FSR in 2^N intervalli larghi K e converto tutto un intervallo nello stesso numero, la conversione sarà:

- **esatta** per la tensione al **centro** dell'intervallo
- errata di $\pm\frac{K}{2}$ per le tensioni agli **estremi**

Riassumendo, abbiamo:

- conversione D/A: $err < \frac{K}{2}$ (soltanto errore di non linearità)
- conversione A/D: $err < \frac{K}{2} + \frac{K}{2} = K$ (errore di non linearità e di quantizzazione)

Come esempio, vediamo una conversione **bipolare**, con $N = 8$, $FSR = 10.24v$. La costante K vale quindi **40mv**, e l'errore di non linearità è minore di **20mv**. Il numero x varia tra –128 e +127.

Nel caso di conversione **D/A ideale**, quando

- $x = +127$, allora $v = 127 \cdot 40mv = +5.08v$.

- $x = -128$, allora $v = -128 \cdot 40mv = -5.12v$.

Visto che la conversione **non è ideale**, avremo invece:

- $x = +127$, -> $v = 127 \cdot 40mv \pm 20mv \rightarrow v \in (+5.06, +5.10)v$

- $x = -128$ -> $v = -128 \cdot 40mv \pm 20mv \rightarrow v \in (-5.14, -5.10)v$

Per una conversione **A/D ideale**, potremmo dire che, ad esempio:

- $v \in (-0.02, +0.02)v \rightarrow x = 0$

- $v \in (-5.14, -5.10)v \rightarrow x = -128$

- $v \in (+5.06, +5.10)v \rightarrow x = +127$

I valori non sono *puntuali* perché comunque ho l'errore di quantizzazione, che non può essere evitato. Se la conversione **non è ideale**, affetta cioè anche da errori di non linearità, gli intervalli, invece di essere tutti grandi 40 mv, avranno una grandezza compresa tra 20mv e 60mv.

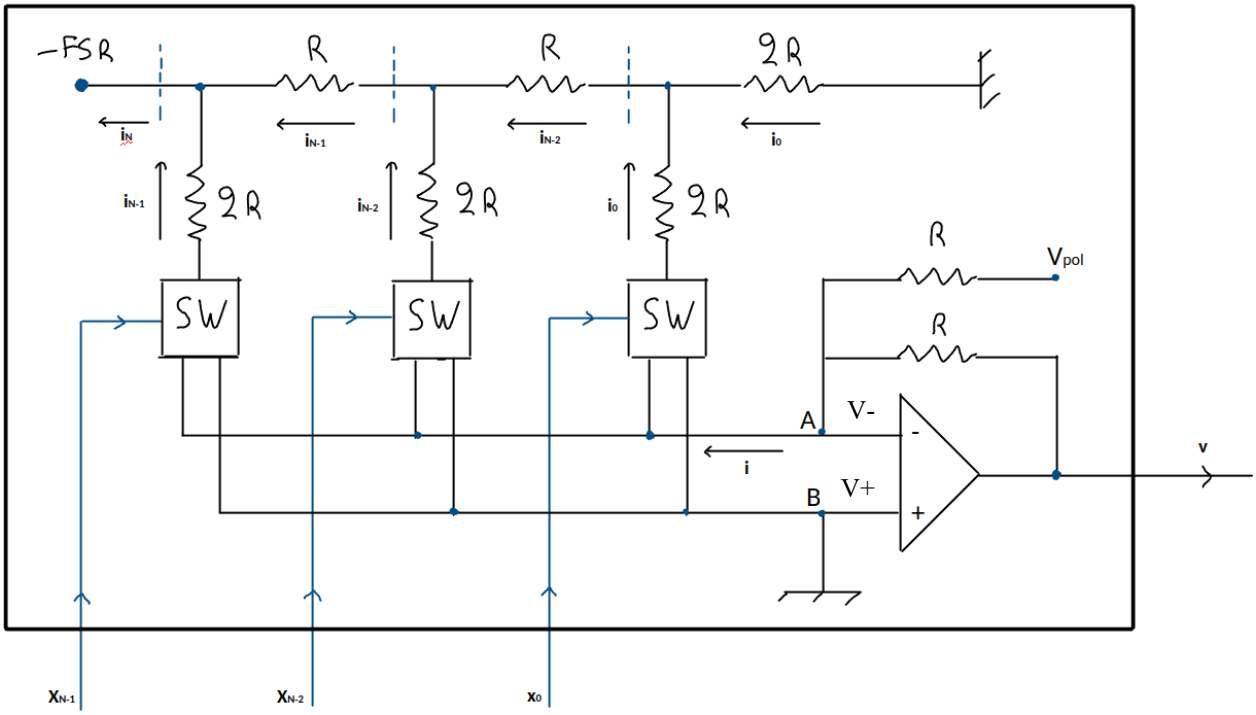
A livello di **tempi di risposta**, i convertitori hanno le seguenti prestazioni:

- quelli D/A, essendo circuiti "combinatori" estremamente semplici, sono velocissimi (pochi *ns*)
- quelli A/D hanno tempi di risposta variabili, perché sono circuiti sequenziali che possono avere architetture diverse. Noi vedremo quelli **ad approssimazioni successive (SAR),** che hanno tempi di risposta di qualche centinaio di *ns*. Ne esistono di altro tipo (quelli **paralleli**), che sono in genere un po' più veloci.

Prima di introdurre i **convertitori** e, a seguire, le **interfacce**, è necessario osservare che i convertitori **bipolari** lavorano rappresentando i numeri interi con **rappresentazione in traslazione** (detta anche, appunto, **binaria bipolare**, che abbiamo visto nella parte di Aritmetica). Ciò vuol dire che il numero intero x è rappresentato col naturale $X = x + 2^{N-1}$. La tensione negativa di fondo scala, che corrisponde al numero intero negativo più piccolo, verrà convertita nel numero naturale 0.

Per convertire un numero da rappresentazione in traslazione a complemento a 2, basta **complementare il bit più significativo**. Quando si fanno esercizi che lavorano con convertitori bipolari è necessario ricordarselo.

2.3.1 Convertitore Digitale/Analogico e relativa interfaccia di conversione



La **resistenza vista** a destra di ogni tratteggio **è pari ad *R*** (se gli switch sono messi a massa). Quindi, in ogni ramo verticale scorre **la stessa corrente che scorre nel ramo orizzontale alla sua destra**. Detto in un altro modo, nel ramo verticale a sinistra scorre **il doppio della corrente** che scorre nel ramo verticale a destra. La corrente che scorre verso l'estremo sinistro del circuito è pari a 2^N volte

la corrente che esce da massa all'estremità destra: $i_N = 2^N \cdot i_0$. Tale corrente vale, secondo il verso delle frecce, FSR diviso la resistenza vista tra quel punto e massa, cioè $i_N = \frac{FSR}{R}$, e quindi:

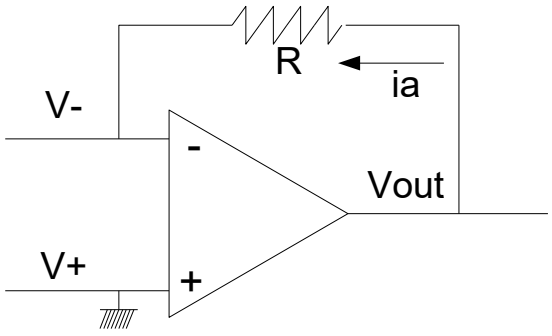
$$i_0 = \frac{FSR}{2^N} \cdot \frac{1}{R} = \frac{K}{R}$$

Gli switch sono degli interruttori che, a seconda del valore di una variabile di comando, che sarà l'*i-esimo* bit della rappresentazione del numero da convertire in analogico, commutano ciò che sta in alto su una linea o su un'altra. Quando la variabile vale 0, all'altro capo c'è **massa** (B). Quando invece vale 1, all'altro capo c'è una linea (A). Tutto il discorso fatto finora si regge sull'ipotesi che gli switch siano sempre connessi a massa. Dimostriamo che questo è vero anche quando uno switch è dirottato sulla linea A.

Abbiamo un **amplificatore operazionale**, cioè un oggetto attivo che:

a) **non fa passare corrente** al suo interno (quindi nei bilanci di corrente non ce lo contiamo)

b) in uscita dà una tensione $V^{out} = \alpha \cdot (V^+ - V^-)$, con (importante) $\alpha \gg 1$.



Riscrivendo l'equazione di prima, si ottiene:

$$V^{out} = \alpha \cdot (V^+ - V^-) = -\alpha \cdot V^-$$

ma è anche vero che:

$$V^{out} - R \cdot i_a = V^-$$

e quindi

$$V^{out} = -\alpha \cdot V^{out} + \alpha R \cdot i_a$$

da cui, infine,

$$V^{out} = \frac{\alpha}{1+\alpha} \cdot R \cdot i_a \cong R \cdot i_a$$

Dal che si conclude che in questo montaggio è $V^- \cong 0$, e che questa cosa non dipende da cosa sia attaccato all'ingresso negativo dell'amplificatore. In pratica, l'operazionale serve ad ancorare a zero la tensione del punto A, senza alterare il bilanciamento di corrente. La corrente i che esce da A verso sinistra vale quindi

$$i = x_0 \cdot i_0 + x_1 \cdot i_1 + \ldots + x_{N-1} \cdot i_{N-1} = i_0 \cdot x_0 + (2 \cdot i_0) \cdot x_1 + \ldots + (2^{N-1} \cdot i_0) \cdot x_{N-1}$$
$$= i_0 \cdot \sum_{i=0}^{N-1} 2^i \cdot x_i$$

La sommatoria altro non è che la rappresentazione posizionale in base 2 di un numero **naturale *X***, e sostituendo il valore trovato prima di i_0 si trova subito:

$$i = \frac{K}{R} \cdot X$$

Quindi, variando gli switch si varia la corrente che scorre da **A verso sinistra**. Detto questo, vediamo come si fa per far uscire la tensione giusta dal convertitore.

Possiamo scrivere le equazioni di bilancio della corrente al nodo A, ottenendo:

$$\frac{K}{R} \cdot X = \frac{V_{pol} + V}{R}$$

cioè:

$$V = K \cdot X - V_{pol}$$
$$V = K \cdot \left(X - V_{pol} \cdot \frac{2^N}{FSR} \right)$$

Quindi:

- se imposto $V_{pol} = 0$, ho un convertitore **unipolare** $V = K \cdot X$
- se imposto $V_{pol} = \frac{FSR}{2}$, ottengo un convertitore **bipolare**, con $V = K \cdot (X - 2^{N-1})$

Appare adesso chiaro che un convertitore D/A è un circuito, in pratica, combinatorio. Da un lato, questo ci garantisce che è molto veloce. Dall'altro, ci possono essere problemi di transizioni multiple dello stato di uscita. Se, ad esempio, passiamo da una configurazione di bit di ingresso $X = 01111111$ alla sua complementata $X = 1000000$, ci saranno di sicuro in uscita delle tensioni **spurie**, che possono creare problemi. Per questo motivo, in genere, a valle di questi convertitori si mette un filtro **passa-basso**, cioè un filtro che taglia le variazioni a frequenza troppo elevata.

Diamo, per finire, un'occhiata alle **due resistenze sulla destra**, entrambe assunte uguali ad R. Che succede se sono un po' diverse? Ripartiamo dal bilancio di corrente:

$$\frac{K}{R} \cdot X = \frac{V_{pol}}{R_1} + \frac{V}{R_2} = \frac{V_{pol} \cdot \gamma_1 + V \cdot \gamma_2}{R}$$

con $\gamma_i = \frac{R}{R_i}$

e quindi ottengo:

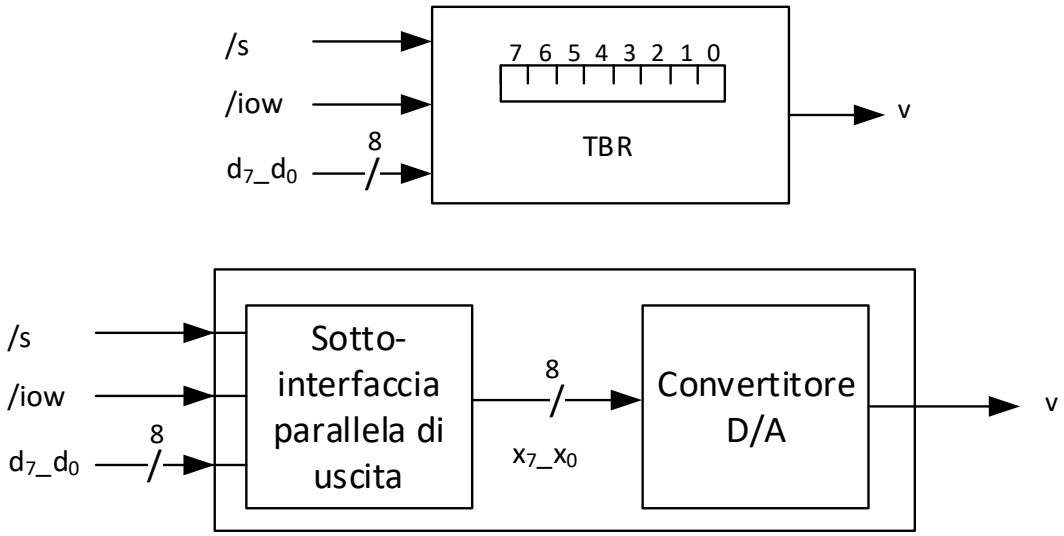
$$V = \frac{K}{\gamma_2} \cdot \left(X - V_{pol} \cdot \gamma_1 \cdot \frac{2^N}{FSR} \right)$$

I due errori sulle resistenze si riflettono in modo diverso sulla tensione di uscita del convertitore. Nella relazione lineare tra la tensione di uscita ed il numero di ingresso X,

- L'errore sulla resistenza in serie a V_{pol} si traduce in un errore di ***offset***, cioè **trasla la retta** in alto o in basso lasciandone inalterata la pendenza (se $V_{pol} \neq 0$)
- L'errore sulla resistenza in serie alla tensione di uscita si traduce in un errore *di* ***guadagno***, cioè modifica la **pendenza della retta**.

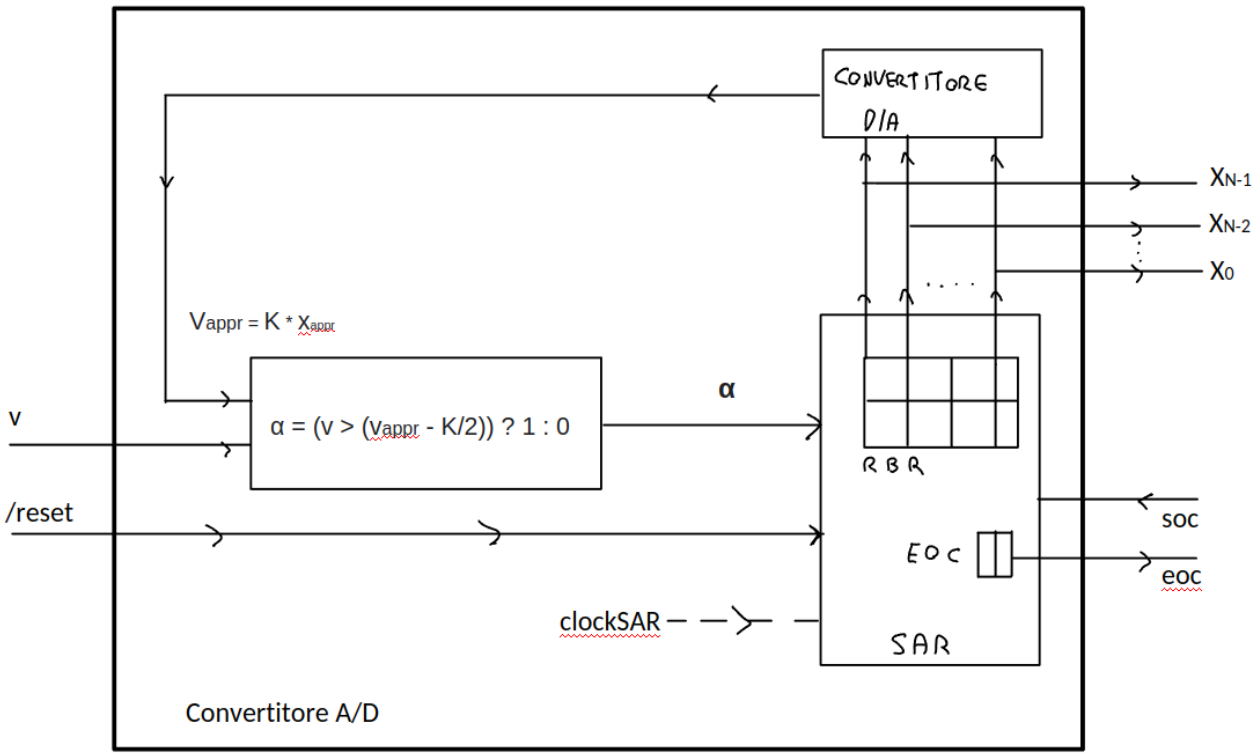
Per questo motivo, il convertitore va **tarato** prima di essere messo in funzione. Le due resistenze appena menzionate sono, in realtà, dei **potenziometri,** cioè delle resistenze variabili che possono essere tarate. Una volta che il convertitore è stato tarato, possiamo contare sul fatto che il suo errore di non linearità sia minore di $\frac{K}{2}$.

Dal punto di vista **funzionale**, un'interfaccia per la conversione D/A appare come un'interfaccia di uscita **senza handshake**, dotata di un solo registro **`TBR`**, nel quale il processore scriverà, a suo piacimento, i dati da convertire in tensione analogica con una istruzione di `OUT`. Visto che il convertitore D/A è più veloce del processore, non c'è bisogno di altro.



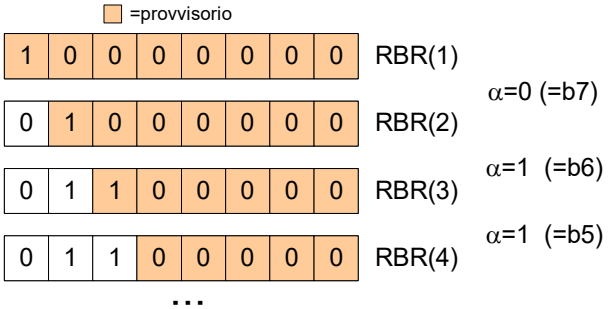
2.3.2 Convertitore Analogico/Digitale e relativa interfaccia di conversione

Descriviamo un convertitore A/D ad approssimazioni successive **ad 8 bit**. È, in apparenza, un oggetto un po' più complesso. La sua parte centrale è una RSS, con un suo clock, detta **SAR** (Successive Approximation Register). Al suo interno ha, inoltre, un **convertitore D/A** (**dello stesso tipo** del convertitore A/D: bipolare se quest'ultimo è bipolare, etc.), con il quale fa una cosa estremamente semplice: quando riceve una tensione di ingresso, il SAR comincia una **ricerca logaritmica** per "indovinare" il byte corrispondente alla tensione fornita. Comincia producendo un byte intermedio, lo converte in analogico, confronta la tensione in ingresso con quella prodotta, e di conseguenza decide se produrre un nuovo byte più grande o più piccolo del precedente.



La ricerca logaritmica (o per *bisezione*) si fa prendendo il punto medio x di un intervallo, confrontandolo con il valore da cercare, e spostandosi nella metà destra o sinistra a seconda del confronto. Quando i numeri sono rappresentati in base 2, la ricerca logaritmica diventa **mettere a posto un bit alla volta, partendo dal più significativo.**

Quindi, si comincia presentando in uscita sul registro `RBR` il byte "al centro dell'intervallo di rappresentabilità". Sia che il convertitore lavori in modalità **unipolare**, sia che lavori in **bipolare**, il byte 10000000 ($= 2^{N-1}$) è il centro dell'intervallo di rappresentabilità (nel caso di conversione bipolare corrisponderebbe all'intero 0, che dovrebbe tradurre una tensione nulla). Questo byte va in ingresso al **convertitore D/A**, e da questo al **comparatore**. Se la **tensione esterna è maggiore**, allora il numero che la rappresenta sarà **più grande di 10000000.** Quindi, avrà di sicuro il primo bit a 1. Altrimenti avrà il primo bit a 0. Quindi, α **rappresenta il bit più significativo del numero da convertire**.



Allora, al successivo clock, andrò a mettere a posto il **secondo bit** (da sinistra), che sarà dato ancora una volta dal valore di α, e così via.

Per terminare la ricerca logaritmica serve un po' di tempo. Quindi il convertitore A/D porta avanti, con l'interfaccia di conversione cui è connesso, un **handshake di tipo soc/eoc**. Inoltre, **la tensione analogica di ingresso deve rimanere costante** per tutto il tempo di conversione. Questa ipotesi viene resa vera inserendo, **prima del convertitore**, un **latch analogico**, che mantiene stabile la tensione.

La descrizione del SAR è estremamente semplice. Sono necessari **due registri**, uno `RBR` a 8 bit per sostenere l'uscita ed uno a 1 bit per `EOC` (più, ovviamente, `STAR`). Al **reset** si assume che l'ingresso `soc` sia a 0 (condizione di riposo) e si tiene `EOC` ad 1.
La prima mossa la fa l'esterno, portando a 1 `soc`. Il convertitore risponde mettendo a 0 `EOC`, e dando il via alla ricerca logaritmica. Quando ha finito riporta `EOC` a 1, **non prima di aver testato `soc`**, altrimenti è errore di handshake, e nel caso torna in S0, dove il dato convertito resta stabile fino a nuova richiesta di conversione. Una conversione termina in una decina di clock (numero proporzionale al numero di bit del convertitore), e ci si può quindi aspettare che duri non più di qualche decina di nanosecondi.

```
module SAR(eoc,x7_x0,soc,alpha,clockSAR,reset_);
  input clockSAR,reset_;
  input soc,alpha;
  output eoc;
  output[7:0] x7_x0;

  reg      EOC; assign eoc=EOC;
  reg[7:0] RBR; assign x7_x0=RBR;
  reg[3:0] STAR;
  parameter S0=0,S1=1,S2=2,S3=3,S4=4,S5=5,S6=6,S7=7,S8=8,S9=9,S10=10;

  always @(reset_==0) #1 begin EOC<=1; STAR<=S0; end
  always @(posedge clockSAR) if (reset_==1) #3
    casex(STAR)
    S0: begin EOC<=1; STAR<=(soc==0)?S0:S1; end
    S1: begin RBR<='B10000000; EOC<=0; STAR<=S2; end
    S2: begin RBR<={         alpha,'B1000000}; STAR<=S3; end
    S3: begin RBR<={RBR[7],  alpha,'B100000}; STAR<=S4; end
    S4: begin RBR<={RBR[7:6],alpha,'B10000}; STAR<=S5; end
    S5: begin RBR<={RBR[7:5],alpha,'B1000}; STAR<=S6; end
    S6: begin RBR<={RBR[7:4],alpha,'B100}; STAR<=S7; end
    S7: begin RBR<={RBR[7:3],alpha,'B10}; STAR<=S8; end
    S8: begin RBR<={RBR[7:2],alpha,'B1}; STAR<=S9; end
    S9: begin RBR<={RBR[7:1],alpha }; STAR<=S10; end
    S10: begin EOC<=(soc==1)?0:1; STAR<=(soc==1)?S10:S0; end
  endcase
endmodule
```

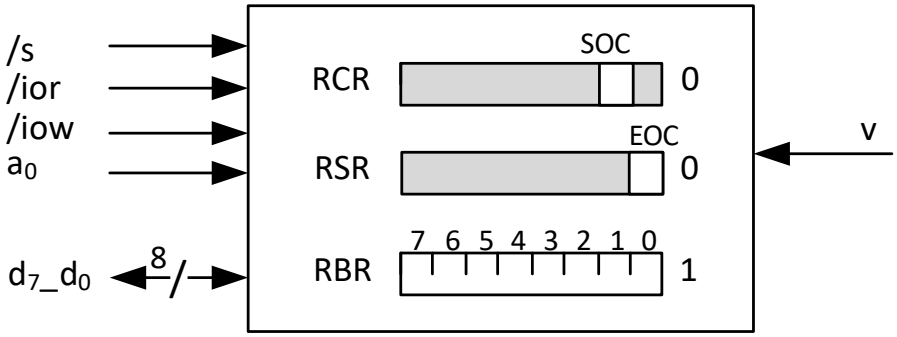
Si può scrivere la descrizione usando un numero di stati inferiore, osservando che i valori da assegnare a RBR possono essere prodotti da una rete combinatoria che abbia come ingressi RBR, α e un contatore che indica il numero di iterazione. Serve un registro COUNT in più, che conta le iterazioni.

```
module SAR(eoc,x7_x0,soc,alpha,clockSAR,reset_);
  input clockSAR,reset_;
  input soc,alpha;
  output eoc;
  output[7:0] x7_x0;

  reg      EOC; assign eoc=EOC;
  reg[7:0] RBR; assign x7_x0=RBR;
  reg[3:0] STAR;
  reg[2:0] COUNT;
  parameter S0=0,S1=1,S2=2,S3=3;

  always @(reset_==0) #1 begin EOC<=1; COUNT<=7; STAR<=S0; end
  always @(posedge clockSAR) if (reset_==1) #3
    casex(STAR)
    S0: begin EOC<=1; STAR<=(soc==0)?S0:S1; end
    S1: begin RBR<='B10000000; EOC<=0; STAR<=S2; end
    S2: begin RBR<=nuovobyte(RBR, alpha, COUNT); COUNT<=COUNT-1;
              STAR<=(COUNT==0)?S3:S2; end
    S3: begin EOC<=(soc==1)?0:1; STAR<=(soc==1)?S3:S0; end
  endcase

  function [7:0] nuovobyte;
    input [7:0] vecchiobyte;
    input alpha;
    input [2:0] posizione;
    casex (posizione)
      7: nuovobyte={                  alpha,'B1000000};
      6: nuovobyte={vecchiobyte[7],   alpha,'B100000};
      5: nuovobyte={vecchiobyte[7:6],alpha,'B10000};
      4: nuovobyte={vecchiobyte[7:5],alpha,'B1000};
      3: nuovobyte={vecchiobyte[7:4],alpha,'B100};
      2: nuovobyte={vecchiobyte[7:3],alpha,'B10};
      1: nuovobyte={vecchiobyte[7:2],alpha,'B1};
      0: nuovobyte={vecchiobyte[7:1],alpha };
    endcase
  endfunction
endmodule
```

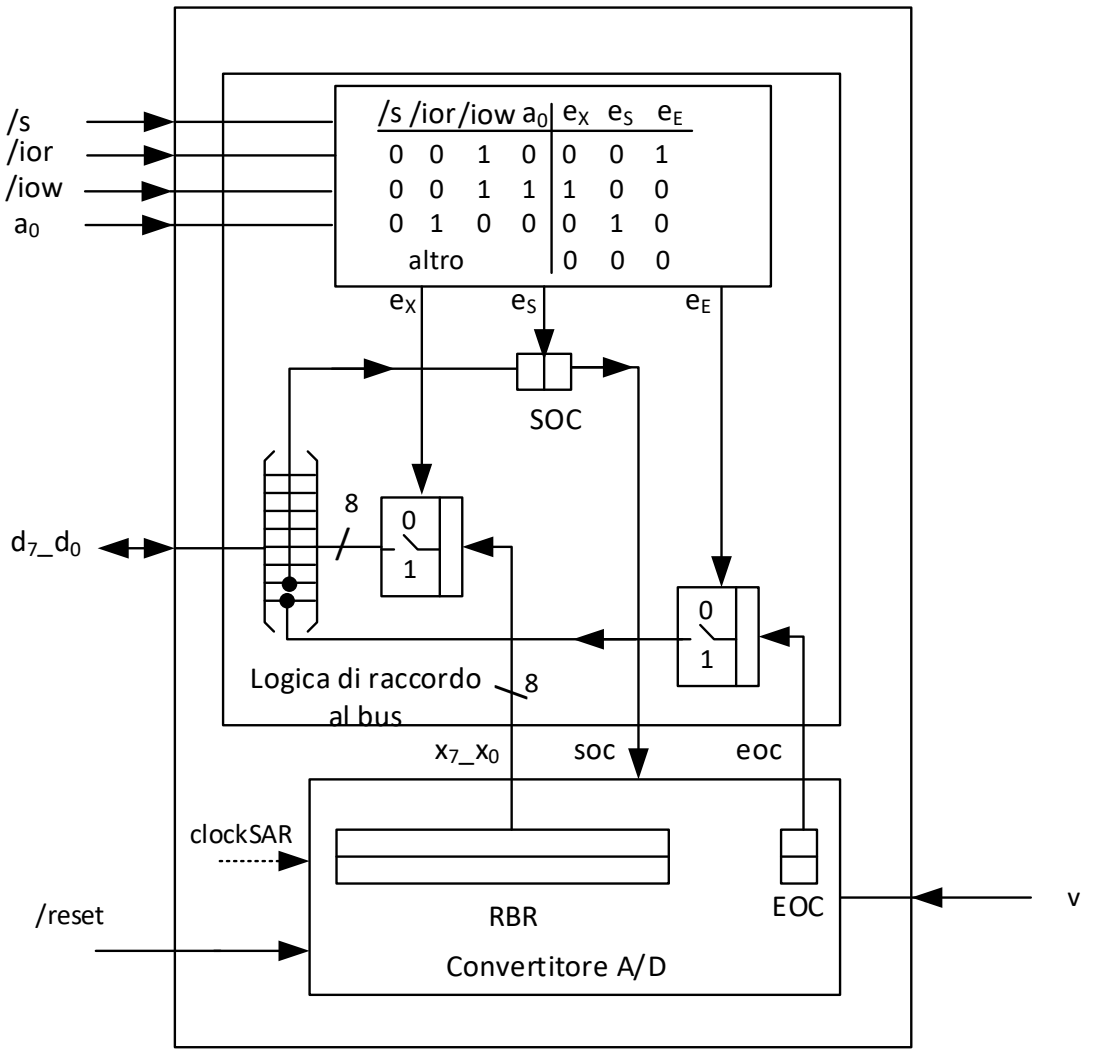
Guardiamo adesso **l'interfaccia di conversione A/D**, che include al suo interno il convertitore. Dal punto di vista **funzionale**, sarà un'interfaccia di ingresso/uscita: infatti, la tensione convertita in numero è un ingresso per il processore, così come il segnale di eoc, ma il processore deve poter **scrivere** per poter iniziare una conversione. Ci vorranno quindi **due registri**:

- un **Receive Status and Control Register** (RSCR), a 8 bit, con due bit significativi: SOC (bit 1), che può essere scritto, ed EOC (bit 0), che può essere letto.

- Un **Receive Buffer Register** (`RBR`) a 8 bit, che – quando `EOC` vale 1 – contiene il byte che converte l'ultima tensione di ingresso vista, secondo la legge del convertitore (unipolare, bipolare).



A livello di **struttura interna**, l'interfaccia di conversione dovrà abilitare delle tri-state per le proprie uscite (`eoc` e `x7_x0`, che vengono dal convertitore). Queste saranno abilitate quando il processore opera una **lettura**, discriminando la porta in base all'indirizzo. Inoltre, dovrà **inviare il segnale di memorizzazione** al registro `SOC`, quando il processore scrive nel `RSCR`. Quindi, c'è soltanto **logica combinatoria.** Si noti che `SOC` viene memorizzato sul fronte di discesa di `/iow` (per le scritture in memoria basta che siano buoni sul fronte di salita di `/mw`).



Il **software** per la gestione di questo convertitore è riportato di seguito (non vanno confusi il **software di gestione di un'interfaccia ed il μ-programma che descrive il convertitore**, che sono due cose totalmente diverse).

In Assembler:

```
            MOV $0x02, %AL
            OUT %AL, RCR_offset         # SOC=1
test1:      IN RSR_offset, %AL
            AND $0x01, %AL
            JNZ test1                   # attendi EOC=0
            MOV $0x00, %AL
            OUT %AL, RCR_offset         # SOC=0
test2:      IN RSR_offset, %AL
            AND $0x01, %AL
            JZ test2                    # attendi EOC=1

            IN RBR_offset, %AL          # prelievo dato convertito
            RET
```

In C++:

```
byte acquisizione( ) {
      #define RCR_offset ...
      #define RSR_offset RCR_offset
      #define RBR_offset ...
      byte tmp;
      //Attiva la conversione immettendo 1 in SOC
      outport(RCR_offset,0x02);
      //Attende che il contenuto di EOC vada a 0 e quindi immette 0 in SOC
      do {tmp=inport(RSR_offset)&0x01;} while (tmp!=0x00);
      outport(RCR_offset,0x00);
      //Attende che il contenuto di EOC vada a 1
      do {tmp=inport(RSR_offset)&0x01;} while (tmp==0x00);
      //Ritorna il risultato della conversione
      return inport(RBR_offset);
}
```

Visto che il convertitore è **molto veloce a iniziare la conversione**, possiamo evitare di attendere che `EOC` vada a zero, e togliere le due parti riquadrate. **Attenzione** a non confondere questa cosa con il fatto di omettere di testare `eoc` in un esercizio di descrizione. Se ho un esercizio con un convertitore A/D, e devo descrivere una rete che si interfaccia con esso per ottenere dei dati, **non posso certo scrivere**:

```
S1:   begin SOC<=1; STAR<=S2; end
S2:   begin SOC<=0; …
```

Ma devo invece scrivere qualcosa di equivalente a:

```
S1:   begin SOC<=1; STAR<=S2; end
S2:   begin SOC<=eoc; STAR<=(eoc==1)?S2:S3; end
```

Altrimenti sarebbe errore di handshake.

3 Appendice

3.1 Domande a risposta multipla

Domande di questo tipo capitano nella prima parte della prova scritta. Le risposte si trovano alla pagina successiva.

1) Un'interfaccia di uscita memorizza i dati provenienti dal bus:

a) Sul fronte di discesa di $/mw$
b) Sul fronte di salita di $/mw$
c) Sul fronte di salita di $/iow$
d) Nessuna delle precedenti

2) Il tempo impiegato da un convertitore A/D del tipo visto a lezione:

a) è costante e molto piccolo
b) dipende dal numero di bit su cui si effettua la conversione
c) dipende dalla tensione di ingresso
d) Nessuna delle precedenti

3) Il colloquio tra il processore ed un'interfaccia parallela con handshake avviene tramite

a) i fili /dav, rfd
b) letture/scritture su un particolare registro dell'interfaccia
c) letture/scritture su un particolare registro del dispositivo gestito dall'interfaccia
d) Nessuna delle precedenti

4) Un trasmettitore seriale è collegato al ricevitore con:

a) tre linee: /dav (uscita), rfd (ingresso), d (dati)
b) tre linee: soc (ingresso), eoc (uscita), d (dati)
c) Una sola linea, d (dati)
d) Nessuna delle precedenti

3.2 Risposte

1) Un'interfaccia di uscita memorizza i dati provenienti dal bus:

e) Sul fronte di discesa di $/mw$
f) Sul fronte di salita di $/mw$
g) Sul fronte di salita di $/iow$
h) Nessuna delle precedenti

2) Il tempo impiegato da un convertitore A/D del tipo visto a lezione:

e) è costante e molto piccolo
f) dipende dal numero di bit su cui si effettua la conversione
g) dipende dalla tensione di ingresso
h) Nessuna delle precedenti

3) Il colloquio tra il processore ed un'interfaccia parallela con handshake avviene tramite

e) i fili /dav, rfd
f) letture/scritture su un particolare registro dell'interfaccia
g) letture/scritture su un particolare registro del dispositivo gestito dall'interfaccia
h) Nessuna delle precedenti

4) Un trasmettitore seriale è collegato al ricevitore con:

e) tre linee: /dav (uscita), rfd (ingresso), d (dati)
f) tre linee: soc (ingresso), eoc (uscita), d (dati)
g) Una sola linea, d (dati)
h) Nessuna delle precedenti