

Appunti su *Struttura del calcolatore*

Gabriele Frassi

A.A 2020-2021 - Primo semestre

Indice degli appunti

1 Unimap	3
I Struttura di un calcolatore	4
2 Giovedì 26/11/2020	5
• Recap sul calcolatore di Von Neumann: sottosistema di I/O, memoria principale e processore (sEP8).	
• Calcolatore visto dal programmatore,	
• Linguaggio Assembler per il processore sEP8: opcode, formati.	
• Architettura del calcolatore: processore e sottosistema di I/O visti come RSS, memoria principale vista come una RSA.	
2.1 Ricapitoliamo sui formati	12
3 Martedì 01/12/2020	13
• Spazio di memoria.	
• Spazio di I/O.	
• Processore.	
• Fasi del processore: reset iniziale, fase di fetch, fase di esecuzione, stato di blocco.	
• Letture e scritture in memoria: temporizzazione ed esempio Verilog.	
• Lettura e scritture nel sottosistema di I/O: temporizzazione ed esempio Verilog.	
• Sottoprogrammi per operazioni di lettura o scrittura in memoria (su 1, 2, 3, 4 byte).	
• Inizio della descrizione in Verilog del processore: reset, functions di supporto (valid_fetch, first_execution_state, alu_result, alu_flag, jmp_condition), fase di fetch.	
3.1 Ricapitoliamo sulle operazioni di lettura e scrittura...	27
3.1.1 Memoria principale	27
3.1.2 Sottosistema di I/O	28
3.2 Ricapitoliamo sui sottoprogrammi per la lettura/scrittura	29
3.3 Corsini - Registri processore sEP8 a seguito della fase di fetch	31
3.4 Ricapitoliamo sugli stati della fase di fetch	32

4 Mercoledì 02/12/2020 **33**

- Fase di esecuzione.
- Introduzione alle interfacce: descrizione di un'interfaccia a livello funzionale, tipi di interfacce, necessità di introdurre meccanismi di handshake.

5 Giovedì 03/12/2020 **41**

- Interfaccia parallela di ingresso senza handshake.
- Interfaccia parallela di uscita senza handshake.
- Montaggio di interfacce parallele di ingresso e uscita.
- Interfaccia parallela di ingresso con handshake: descrizione Verilog della RSS che gestisce l'handshake.
- Interfaccia parallela di uscita con handshake: descrizione Verilog della RSS che gestisce l'handshake.
- Interfaccia parallela di ingresso-uscita con handshake.
- Interfacce seriali: nozioni di comunicazione seriale, descrizione Verilog di Trasmettitore e Ricevitore.

6 Martedì 09/12/2020 **54**

6.1 Conversione analogico/digitale e digitale/analogica	54
6.1.1 Convertitore D/A	56
6.1.1.1 Interfaccia per la conversione D/A	59
6.1.2 Convertitore A/D	60
6.1.2.1 Interfaccia di conversione A/D	63

Capitolo 1

Unimap

1. **Mer 25/11/2020 10:45-12:00 (1:30 h)** esercitazione: Esercizi di descrizione e sintesi di reti sequenziali sincronizzate. (GIOVANNI STEA)
2. **Gio 26/11/2020 10:45-12:45 (2:0 h)** lezione: Il calcolatore visto come un insieme di moduli interconnessi: processore, spazio di memoria e spazio di I/O. Visione funzionale del processore didattico sEP8 (8 bit simple Educational Processor): i registri, le istruzioni, inizializzazione al reset. Modalità di indirizzamento degli operandi. Linguaggio Assembler e linguaggio macchina. Formato delle istruzioni. (GIOVANNI STEA)
3. **Ven 27/11/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con ing. Raffaele Zippo): svolgimento di esercizi di descrizione e sintesi di reti sequenziali e combinatorie in ambiente Verilog. (GIOVANNI STEA)
4. **Mar 01/12/2020 11:45-13:45 (2:0 h)** lezione: descrizione del calcolatore: memoria, spazio di I/O, processore. Descrizione dei registri del processore. Letture e scritture in memoria e nello spazio di I/O. Descrizione della fase di reset e di fetch in Verilog. (GIOVANNI STEA)
5. **Mer 02/12/2020 10:45-12:45 (2:0 h)** lezione: Descrizione del processore: fase di esecuzione. Interfacce: visione funzionale. Interfacce con e senza handshake. Accesso a controllo di programma. (GIOVANNI STEA)
6. **Gio 03/12/2020 10:45-12:45 (2:0 h)** lezione: Interfacce parallele con e senza handshake, ingresso ed uscita. Trasmissione seriale start/stop. Interfaccia seriale. (GIOVANNI STEA)
7. **Ven 04/12/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con ing. Raffaele Zippo). Svolgimento di esercizi di descrizione di reti complesse al calcolatore. (GIOVANNI STEA)
8. **Mer 09/12/2020 10:45-12:45 (2:0 h)** lezione: Descrizione del trasmettitore e ricevitore seriale. Conversione analogico/digitale e digitale/analogica. Convertitore D/A ed interfaccia di conversione. (GIOVANNI STEA)
9. **Gio 10/12/2020 10:45-12:45 (2:0 h)** lezione: Convertitore analogico/digitale e relativa interfaccia di conversione. Chiusura del corso. (GIOVANNI STEA)

Parte I

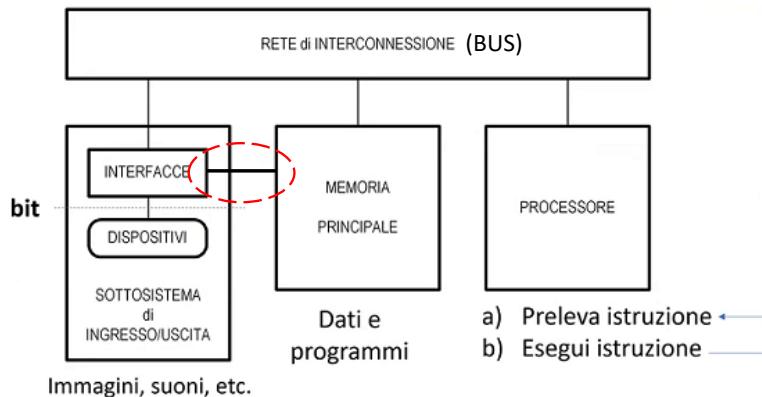
Struttura di un calcolatore

Capitolo 2

Giovedì 26/11/2020

Struttura del calcolatore

Il blocco finale consiste nel descrivere in Verilog un calcolatore, TUTTO, da cima a fondo. Riprendiamo la solita immagine del calcolatore di Von Neumann



Descriptoremo processore, memoria, interfacce e dispositivi di I/O utilizzando quanto studiato fino ad ora! Chiaramente il calcolatore che descriveremo non sarà quello che normalmente utilizziamo per seguire le lezioni (quello è troppo complicato), ma un qualcosa che fino a trent'anni fa (quando Stea studiava) poteva essere considerato piuttosto potente.

Cosa possiamo dire sui vari moduli?

- **Sottosistema di I/O.** Si gestisce la codifica delle informazioni ed il loro scambio col mondo esterno (in entrambi i sensi). All'interno abbiamo:
 - o **Dispositivi** (trasduttori), che effettuano la codifica vera e propria;
 - o **Interfacce**, che gestiscono i vari dispositivi, cioè standardizzano il colloquio tra processore e trasduttore (necessario, il processore non deve conoscere i trasduttori per poter lavorare).

Ciascuna interfaccia possiede un numero piccolo di registri dove il processore può svolgere operazioni di lettura o di scrittura (in casi rari entrambe): un registro in un interfaccia può contenere, ad esempio, l'ultimo tasto premuto della tastiera, oppure può indicare se deve essere accesa una spia del nostro dispositivo.

- **Memoria principale.** Contiene le istruzioni da eseguire e i dati elaborati dal processore (ricordiamo che alcuni dati possono essere ospitati nel sottosistema di I/O). Una parte di memoria è adibita a memoria video: non la faremo, ma dobbiamo tenere conto del collegamento diretto tra interfaccia e memoria principale (devo salvare in memoria la replica di ciò che stampo sullo schermo). La memoria è realizzata con tecnologia RAM, e in parte ROM (o EPROM, per eseguire certe istruzioni al reset).
- **Processore.** Il processore ciclicamente preleva istruzioni e le esegue. Normalmente abbiamo una sequenza di istruzioni eseguita nell'ordine posto, salvo utilizzo di istruzioni operative che alterano il normale flusso (si indica un nuovo indirizzo e il prelievo di istruzioni riparte da lì). Il processore si ferma quando incontra un'istruzione *HLT*.

Al momento del reset il processore deve essere avviato in modo consistente.

- Devo indicare la prima operazione da leggere (un'operazione ben precisa)
- Chiaramente l'indirizzo di memoria ci porta a un'area realizzata con tecnologia ROM/EPROM, quindi una memoria non volatile.

Posso fare questo inizializzando l'*IP* (*Instruction pointer*) e altri registri. La memoria non volatile contiene al suo interno un programma bootstrap eseguito all'accensione del calcolatore.

Quale processore utilizzeremo? Il sEP8 (acronimo di *8-bit simple Educational processor*), che presenta le seguenti proprietà:

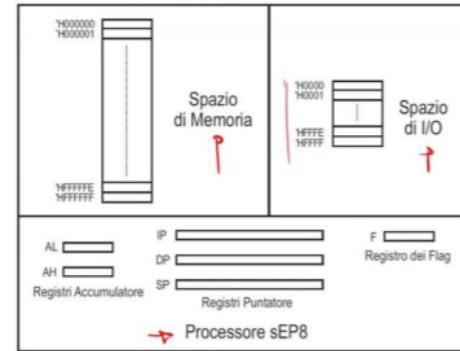
- o elabora dati a 8 bit;
- o lavora in aritmetica in base 2 con interi rappresentati in C2;
- o indirizza memoria di 16Mbyte (quindi avrà 24 fili per rappresentare tutti gli indirizzi).

Il calcolatore consiste sostanzialmente in un insieme di RSS: tutti gli elementi che vedremo, tranne dispositivi (parti elettromagnetiche che non sono di nostro interesse) e parte della memoria (che è una RSA), sono RSS. Tutte le RSS presenteranno un piedino di reset e saranno collegate allo stesso circuito di reset (in modo tale da avere un avvio consistente).

In particolare, descriveremo il processore come una RSS sintetizzabile in Parte Controllo e Parte operativa. Ne daremo una specifica come con una qualunque RSS. Oltre a questo vedremo come il processore si interfaccia con le altre reti e qual è il suo comportamento osservabile (il suo linguaggio macchina, perderemo un po' di tempo sulla questione).

Calcolatore visto dal programmatore

- La memoria consiste in uno spazio lineare di 2^{24} byte (16Mbyte)
- Gli indirizzi sono a 24bit (combinazioni possibili di valori sono, non a caso, 2^{24})
- Lo spazio di I/O (registri di interfaccia) consiste in uno spazio lineare di 2^{16} locazioni o porte da un byte.
- Gli indirizzi sono a 16bit (tenerne conto quando vediamo il bus).
- Queste locazioni consistono nell'insieme dei registri di interfaccia che il processore può teoricamente indirizzare.
Perché teoricamente? Non è detto che ad ogni locazione corrisponda un registro di interfaccia, anzi è molto probabile che la maggior parte di questo spazio non presenti implementazione fisica (le interfacce sono poche).
- Il processore, da queste porte, può leggere o scrivere un byte alla volta (quindi se dobbiamo leggere più byte dovremo svolgere più operazioni)



Registri del processore:

- o Registri accumulatore (AH, AL, 8 bit), contengono operandi di elaborazioni.
- o Registro dei flag (8 bit), i 4 bit più interessanti per noi sono il CF, lo ZF, il SF e l'OF.
- o Registri puntatore (a 24 bit, visto che devono contenere indirizzi):
 - IP (Instruction pointer), 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 delle modalità di indirizzamento.

Al reset dobbiamo inizializzare il registro dei flag e l'instruction pointer, nel seguente modo

```
F <= 'H00;  
IP <= 'HFF0000; //Chiaramente memoria ROM implementata a partire da questo indirizzo)
```

Osservazione: abbiamo anche altri registri, noi abbiamo visto solo i registri utilizzabili nelle istruzioni.

Linguaggio Assembler e Linguaggio macchina

- All'inizio del corso abbiamo introdotto le peculiarità del linguaggio Assembler, che è legato al linguaggio macchina da un rapporto 1:1. Per descrivere il calcolatore dovremo parlare di linguaggio macchina, in modo da poter descrivere il comportamento osservabile della macchina.
- Il linguaggio Assembler, ricordiamo, è diverso da processore a processore. In questo caso utilizzeremo un linguaggio Assembler simile il più possibile a quello già visto per i processori Intel. Chiaramente l'obiettivo non è programmare in Assembler, ma disporre il linguaggio in modo tale che il calcolatore sia facilmente descrivibile.

Linguaggio Assembler

Formato delle istruzioni:

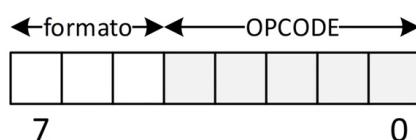
OPCODE source, destination

Dove OPCODE è il codice operativo dell'istruzione, mentre source e destination indicano, rispettivamente, l'operando sorgente e quello destinatario. In alcuni casi è assente l'operando source, in altri (rari) entrambi gli operandi (*NOP* e *HLT*).

- **Modalità di indirizzamento:**
 - o Indirizzamento di registro, uno o entrambi gli operandi sono nomi di registro
OPCODE AL, AH
OPCODE DP
 - o Indirizzamento immediato, l'operando sorgente è specificato direttamente nell'istruzione come costante (chiaramente possiamo fare questa cosa solo con l'operando sorgente)
OPCODE \$0x10, AL
 - o Indirizzamento di memoria, valido per il sorgente o per il destinatario (mai contemporaneamente). L'indirizzamento di memoria può essere
 - 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
 - o Indirizzamento delle porte di I/O: le porte di I/O si indirizzano in modo diretto, specificando l'indirizzo della porta nell'istruzione stessa
IN 0x1010, AL
OUT AL, 0x9F10
- **Istruzioni di controllo:** istruzioni che alterano il flusso di esecuzione del programma. Permettono salti, condizionati e non, chiamate di sottoprogramma ed istruzioni di ritorno
JMP indirizzo
JCon indirizzo
CALL indirizzo
RET
Nelle prime tre istruzioni si specifica l'indirizzo a cui si salta (si sostituisce l'indirizzo di IP), le ultime due interagiscono con la pila. La CALL salva in pila il contenuto di IP (3 BYTE), cioè l'indirizzo dell'istruzione successiva alla CALL. La RET preleva dalla pila un indirizzo (3 byte) e lo sostituisce ad IP.

Linguaggio macchina e formati

- Un processore deve tradurre un'istruzione Assembler
OPCODE source, destination
in una sequenza di zeri e uni con una certa sintassi. La sintassi è il linguaggio macchina del processore, che deve essere compatta e facile da interpretare (per il compilatore, non per noi).
 - o La prima cosa che guardano gli esseri umani, normalmente, è il tipo di operazione (l'Assembler, concepito per gli esseri umani, pone il tipo prima degli operandi).
 - o I processori, invece, guardano per prima cosa gli operandi. Vediamo degli esempi
 - MOV AH, AL
Gli operandi sono già posti all'interno di registri
 - MOV \$0x10, AL
Il processore deve leggere in memoria l'operando sorgente indicato nell'istruzione
 - MOV (DP), AL
Il processore dovrà leggere in memoria per procurarsi l'operando sorgente. L'indirizzo non è il registro DP, ma il valore contenuto nel registro stesso.
- Dobbiamo distinguere, in queste operazioni, la fase di fetch dalla fase di esecuzione: la prima consiste nel procurarsi gli operandi (e può essere diversa in base all'indirizzamento scelto), la seconda è uguale per tutte queste istruzioni (cioè spostare valori)
- Ciascuna istruzione macchina è lunga almeno un byte. Il primo byte di ogni istruzione codifica sia il **tipo di operazione** (su 5 bit, 32 opcode possibili) che il modo in cui si devono recuperare gli operandi (su 3 bit, 8 formati possibili). Quest'ultima cosa è detta **formato dell'istruzione**.



- **Formati possibili:**

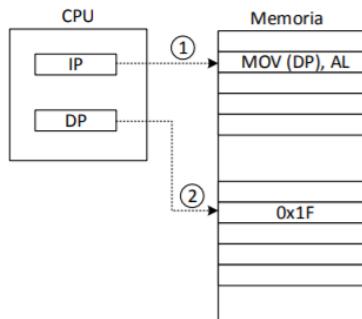
o Formato F0 (000)

- Categoria in cui rientrano tutte le istruzioni per le quali non è necessario compiere nessuna azione per procurarsi gli operandi.
- In questa categoria rientrano operazioni in cui gli operandi sono registri o dove non sono presenti operandi (HLT, NOP, RET).
- La fase di fetch consiste esclusivamente nella lettura di un byte (quello dell'istruzione) visto che non c'è altro da fare.

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

o Formato F2 (010)

- Categoria in cui rientrano le istruzioni dove l'operando sorgente si trova in memoria ed è indirizzato tramite DP.
- Il sorgente deve essere ripescato in memoria. Dovrà fare una seconda lettura in memoria per portare l'operando sorgente dentro il processore.



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

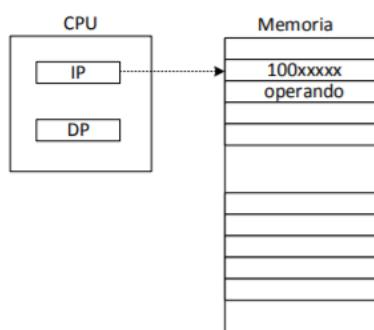
o Formato F3 (011)

- Categoria in cui rientrano le istruzioni dove l'operando destinatario si trova in memoria ed è indirizzato usando DP (solo le MOV)
- Codifico su un unico byte l'istruzione. La fase di fetch consiste nel non fare niente: il contenuto da spostare è già presente nel processore, stessa cosa l'indirizzo da raggiungere.
- La scrittura del destinatario avviene in fase di esecuzione.

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

o Formato F4 (100)

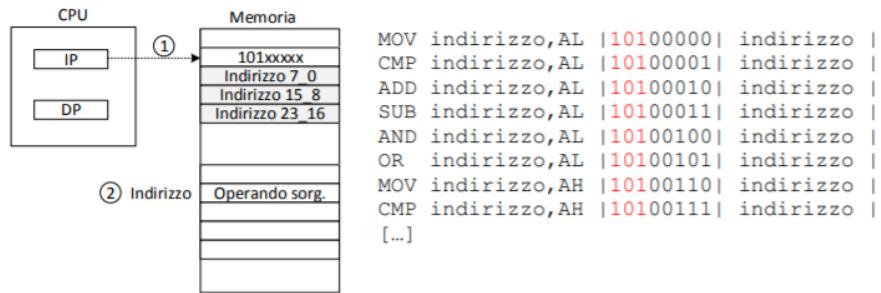
- Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo immediato, e sta su 8 bit.
- L'istruzione è lunga due byte: il primo contiene l'istruzione, il secondo l'operando indirizzato in modo immediato. La fase di fetch consiste nel fare due letture consecutive.



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
[...]		

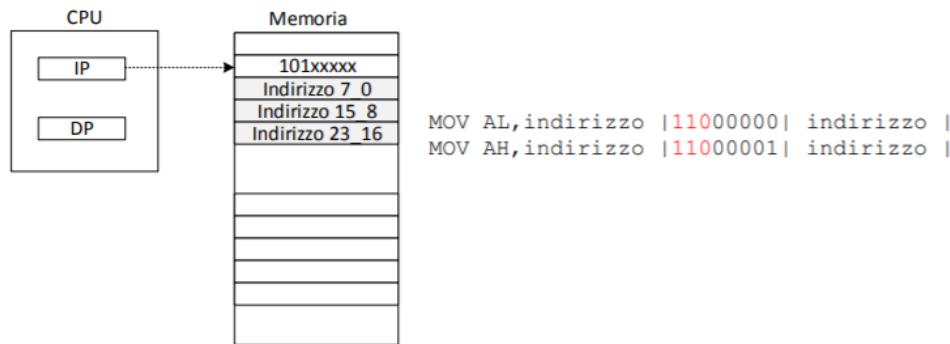
- Formato F5 (101)

- CATEGORIA in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo diretto. Ciò pongo direttamente l'indirizzo del sorgente.
- In fase di Fetch l'operando sorgente deve essere riportato nel processore.
- L'operazione sarà lunga 4 byte: uno di opcode e tre di indirizzo di memoria (24 bit per poter rappresentare qualunque indirizzo). Seguono tre cicli di lettura consecutivi a partire da IP. Ciò non basta: devo fare un'altra lettura all'indirizzo trovato: a quel punto ho raggiunto l'operando sorgente e posso porlo nel processore.



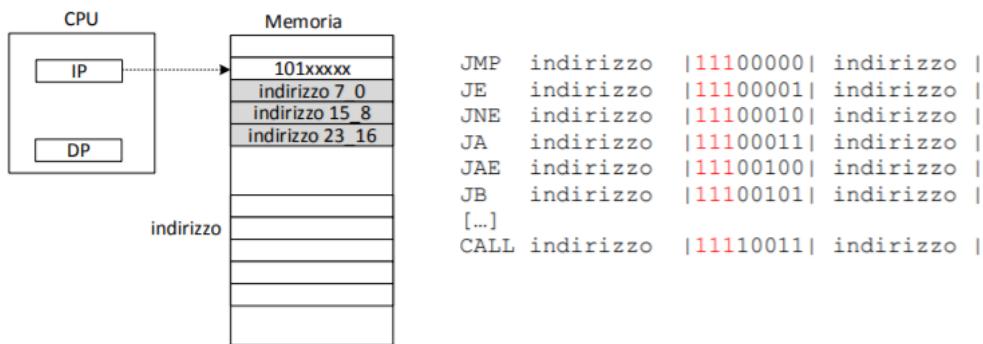
- Formato F6 (110)

- CATEGORIA in cui rientrano le istruzioni dove l'operando destinatario è in memoria, indirizzato in modo diretto.
- Il processore dovrà leggere 4 byte in memoria: uno per l'opcode, tre per l'indirizzo del destinatario.
- La scrittura del destinatario avviene in fase di esecuzione.



- Formato F7 (111)

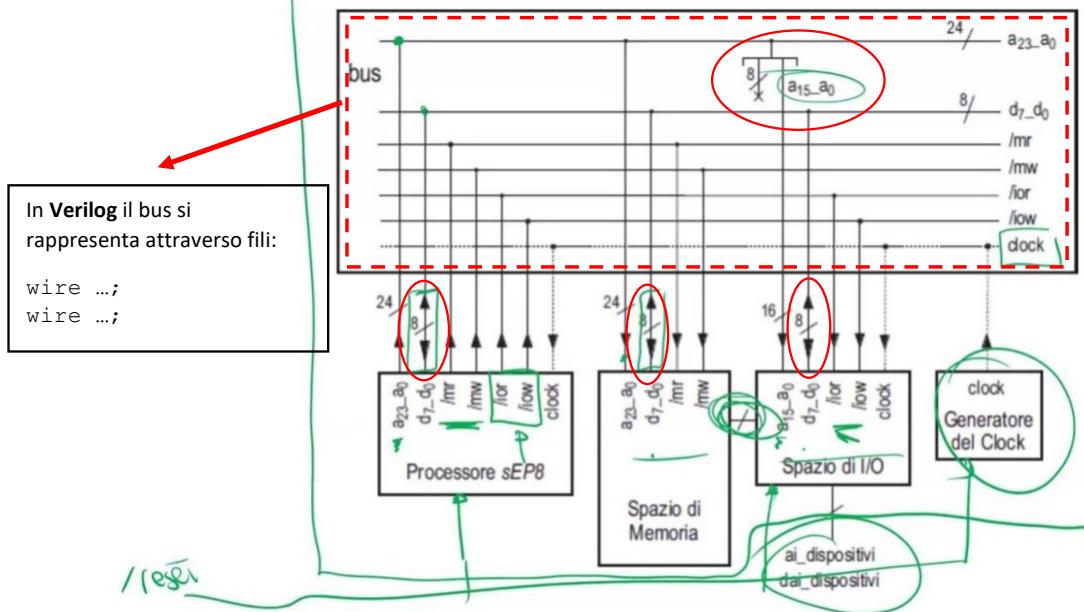
- Uguale al precedente, raggruppa le istruzioni di controllo (CALL, JMP, Jcon) in cui ho un indirizzo di salto.
- Utilizzo un byte per l'opcode, altri tre per l'indirizzo. In fetch abbiamo la lettura di 4 byte consecutivi, a partire da IP.



- Formato F1 (001) "formato delle varie ed eventuali"
 - Categoria in cui rientrano le istruzioni non classificabili nei formati precedenti: istruzioni I/O, MOV con uno dei registri a 24 bit.
 - Le azioni differiscono da un'istruzione a un'altra: si è preferito fare un unico formato dove ci si limita ad estrarre solo l'opcode. La gestione degli operandi viene rimandata alla fase di esecuzione.
 - La cosa è poco pulita, ma molto più semplice.

Lista completa delle istruzioni con formato corrispondente alle pagine 185-187 del libro di Corsini

Architettura del calcolatore



- Spogliamo il calcolatore e vediamo cosa è presente sulla rete di interconnessione
- Abbiamo:
 - **24 fili di indirizzo** che partono dal processore sEP8 e indicano l'indirizzo delle locazioni di memoria o delle porte di I/O dove vuole leggere e scrivere. Entrano in ingresso in tutti gli altri moduli. Attenzione allo spazio di I/O: avendo solo 64K mi bastano le 16 cifre meno significative per rappresentare tutti gli indirizzi possibili.
 - **Fili di dati.** Il processore legge e scrive singoli byte alla volta: ho bisogno di 8 fili che dovranno essere pilotati alternativamente dal processore e dagli altri dispositivi (porte tristate). Dobbiamo evitare cortocircuito sui fili di dati.
 - **Fili di controllo.** Attivi bassi con cui possiamo pilotare alcuni moduli a partire dal processore: /mr per leggere e scrivere in memoria, /ior e /iow per leggere e scrivere nello spazio di I/O. Chiaramente l'uso di queste variabili terrà conto delle leggi di temporizzazione viste per i cicli di lettura in memoria RAM.
 - **Generatore di clock:** tutti i moduli del calcolatore, tranne la memoria, sono collegati allo stesso generatore di clock.
 - **Fili di interconnessione tra interfacce e dispositivi.**
 - **Fili di comunicazione tra memoria video e adattatore grafico** (non vedremo)
 - Tenere conto anche della presenza dei **piedini per il reset** (il reset arriva contemporaneamente a tutti i moduli che ne hanno necessità). Per semplicità grafica la parte relativa al reset viene solitamente omessa.

2.1 Ricapitoliamo sui formati

I formati non vanno imparati a memoria, ma ricordati mediante la logica.

Quanti sono gli operandi? Otto, lo capisco dal numero di bit dedicati al formato (3).

Quali sono i formati? Seguire la scaletta.

- **Primo formato** (F0): quello dove la vita è più semplice, non dobbiamo andare a recuperare nessun operando.
- **Terzo e quarto formato** (F2, F3): registri puntatore, rispettivamente in source e destination. L'altro operando è un registro.
- **Quinto formato** (F4): costante in source. L'altro operando è registro.
- **Sesto e settimo formato** (F5, F6): indirizzamento diretto, rispettivamente in source e dest. L'altro operando è registro.
- **Ottavo formato** (F7): istruzioni di salto (sia quello condizionato che non). L'operando è registro.
- **Secondo formato** (F1): varie ed eventuali, precisamente istruzioni IN/OUT o modifica di registri a 24bit (i registri puntatore). Lo si lascia in fondo nel ragionamento.

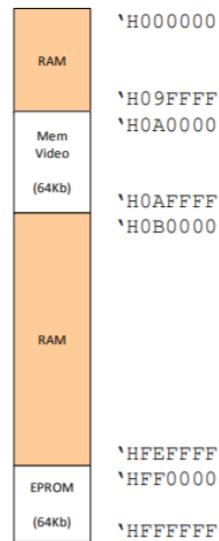
Capitolo 3

Martedì 01/12/2020

Spazio di memoria

- Lo spazio di memoria consiste in 16Mbyte di locazioni realizzate in larga parte con tecnologia RAM e in parte con tecnologia EPROM (questa parte conterrà il *programma bootstrap* da eseguire all'avvio).
- Una parte ulteriore della memoria è adibita a memoria video: questa è di tipo diverso dalle altre, visto che dobbiamo permettere lo svolgimento di operazioni di lettura e scrittura in simultanea da più oggetti (l'interfaccia per stampare la schermata aggiornata, il processore per aggiornare il contenuto della schermata).
- Lo spazio di memoria è strutturato in più parti:
 - o Due realizzate in tecnologia RAM
 - o Una adibita a memoria video
 - o Una realizzata in tecnologia EPROM.

Gli intervalli di indirizzo sono evidenti nell'immagine. Per ottenere una memoria del genere dobbiamo recuperare quanto detto sul montaggio in parallelo di memorie. Utilizzeremo delle maschere (rete combinatoria) per indicare quale chip di memoria, tra quelli introdotti, ci interessa visitare (genera il segnale di abilitazione).

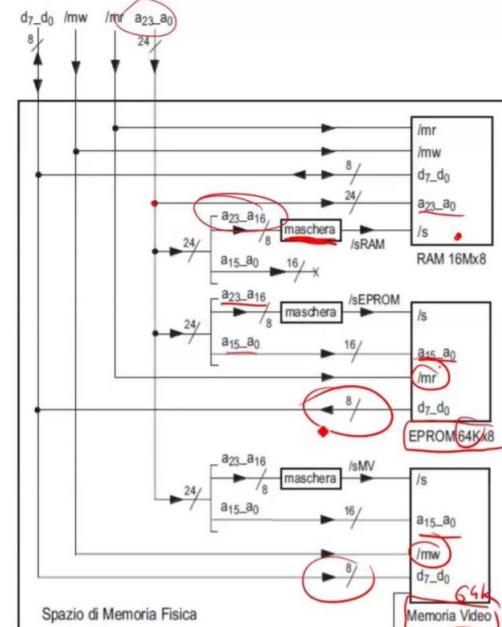


Nella rete avremo tre chip: uno di RAM 16Mx8, due a 64Kx8 (la EPROM e la memoria video).

- Tutti gli indirizzi dello spazio di memoria sono a 24bit: mentre nella RAM sono rilevanti anche le 8 cifre significative per determinare la posizione all'interno del CHIP, nelle altre sono rilevanti solo le cifre rimanenti. Le cifre più significative, in quei casi, servono solo a indicare quale tra i due chip vogliamo raggiungere.
- Pongo in ingresso nelle maschere le 8 cifre più significative dell'indirizzo. Le maschere determineranno i valori degli attivi alti /sRAM, /sEPROM, e /sMV.
- Chiaramente non abbiamo l'ingresso /mw nella memoria EPROM (possibili solo operazioni di lettura).
- **Come scriviamo la tavola di verità della maschera?**

a ₂₃	a ₂₂	a ₂₁	a ₂₀	a ₁₉	a ₁₈	a ₁₇	a ₁₆	/sRAM	/sMV	/sEPROM
0	0	0	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	0	1	0

altro F F

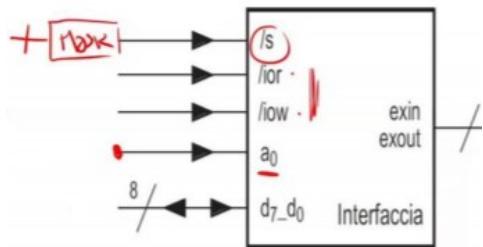


- Consideriamo che la RAM è divisa in due parti: considero le cifre più significative (fisse) della memoria video e della EPROM, e la RAM la lascio alle varie ed eventuali.
- Le cifre più significative degli indirizzi della memoria video sono **0A**: 0000 e 1010
- Le cifre più significative degli indirizzi della EPROM sono **FF**: 1111 e 1111
- Con cifre significative diverse vado in memoria RAM.
- **Conclusione:** il segnale di select viene generato dall'indirizzo, non abbiamo un filo di select sul bus.
- **Ulteriore osservazione:** la RAM copre anche gli indirizzi coperti dalla EPROM e dalla memoria video. Se indico questi indirizzi, tuttavia, la RAM non risponde (la maschera non genera il segnale di selezione).

Spazio di I/O

- Spazio di 64K celle (dette porte), non tutte implementate (l'implementazione è a cura delle interfacce montate sul calcolatore, poche).
- Un'interfaccia (qua a lato) si presenta come una memoria RAM: ho un piedino di select /s prodotto da una maschera.
- Le variabili /ior e /iow hanno la stessa funzione di /mr e /mw, rispettivamente. Sono identiche ma non uguali.
- Le variabili che non vanno nella maschera finiscono in ingresso come indirizzi interni se l'interfaccia presenta più di una porta. Nel nostro esempio ne ha due, quindi ci servirà una singola variabile logica.
- **Osservazioni sulle interfacce dal lato bus:**
 - o In una RAM si può leggere e scrivere in una qualunque locazione. Un'interfaccia, invece, può supportare solo operazioni di lettura o solo operazioni di scrittura. Chiaramente uno dei due fili di comando, in questi casi, risulta superfluo. Nella maggior parte dei casi le interfacce presenteranno porte di entrambi i tipi.
 - o Se un'interfaccia presenta una sola porta allora non ho bisogno delle variabili di indirizzo (mi basta solo il select)

IS	/iow	zione
1	-	nessuno
0	1	"
0	0	ciclo lettura
0	1	ciclo scrittura
0	0	non def.



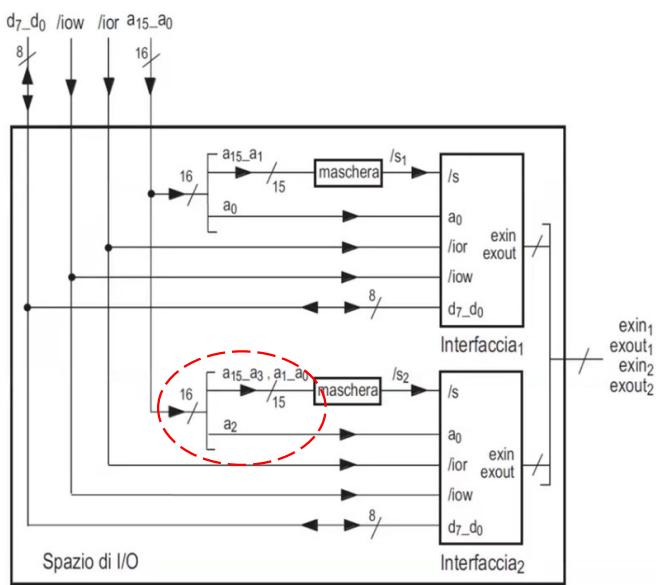
Struttura di un'interfaccia con tavola di verità per ripassare

Osservazioni sulle interfacce dal lato dispositivo:

- o Gli ingressi e le uscite variano da interfaccia a interfaccia e verranno descritti più avanti.
- o **A cosa ci serve avere delle interfacce?**
 - I dispositivi hanno velocità molto diverse tra loro, e spesso sono più lenti del processore (visto che si interfacciano col mondo esterno). Se io collegassi i dispositivi direttamente al bus il processore dovrebbe conoscere le proprietà del dispositivo cosa insana realizzare i processori tenendo conto dei dispositivi esistenti, che cambiano nel tempo in modo rapido)
 - Le modalità di trasferimento dei dati sono molto diverse. Alcuni dispositivi trasferiscono un bit alla volta, altri gruppi di bit.

L'interfaccia permette di avere temporizzazioni omogenee e trasferimento di dati omogeneo.

- Vediamo il nostro spazio di I/O, con due interfacce:
 - o Entrambe hanno due porte (lo vediamo dalla variabile a₀)
 - o Entrambe sono interfacce sia di ingresso che di uscita.
 - o Abbiamo 16 fili (non 24) che entrano nello spazio di I/O. 15 fili vanno nella maschera per generare il select.
 - o Nell'esempio supponiamo di avere le porte della prima interfaccia agli offset 'H03C8, 'H03C9, e le porte della seconda interfaccia agli offset 'H0060, 'H0064. La maschera ci permette di generare i valori di /s₁ ed /s₂



- **Attenzione:** non per forza la cifra meno significativa è quella utilizzata per distinguere le porte. Nella seconda interfaccia utilizziamo la terza cifra meno significativa per indicare la porta desiderata!

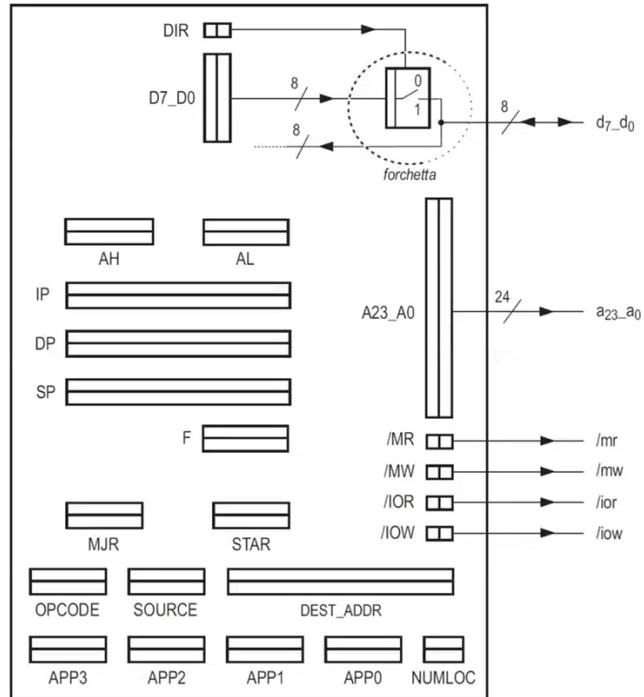
- 'H03C8: 0000 0011 1100 1000
- 'H03C9: 0000 0011 1100 1001
- 'H0060: 0000 0000 0110 0000
- 'H0064: 0000 0000 0110 0100

Cifre significative con cui identifico la porta dell'interfaccia

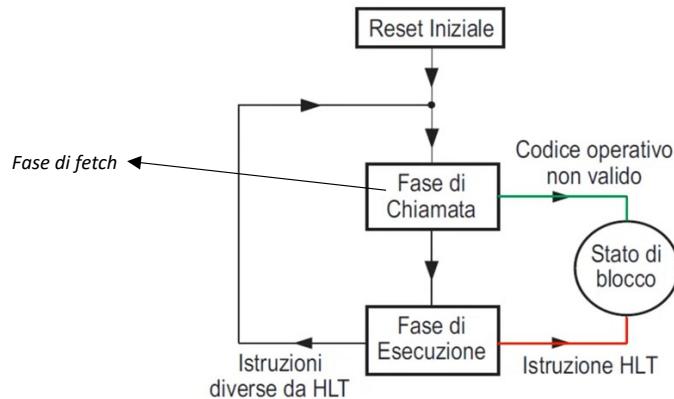
Processore

Il processore è una RSS sincronizzata che presenta un certo numero di registri. Abbiamo:

- Registri a supporto delle uscite, quindi:
 - D7_D0, per i fili di dati
 - A23_A0, per i fili di indirizzo
 - /MR e /MW, per interagire con la memoria principale
 - /IOR e /IOW, per interagire con il sottosistema di I/O
- Un registro DIR per gestire la porta tristate per i fili di dati. Se DIR è uguale a zero la porta tristate è in alta impedenza, altrimenti è in conduzione.
- Il registro STAR con cui indichiamo lo stato interno della rete.
- Un registro dei flag, che conterrà flag come CF, SF, ZF, OF.
- I registri accumulatori AH e AL
- Il registro puntatore DP (indirizzamento di memoria indiretto)
- Il registro puntatore SP per la pila
- Il registro puntatore IP, per ricordarsi ogni volta l'istruzione successiva da eseguire. In alcuni casi potrebbe essere incrementato più volte (dipende dalla dimensione dell'istruzione, che può essere superiore al singolo byte).
- Cinque registri a supporto delle operazioni di lettura e scrittura:
 - APP0, APP1, APP2, APP3, che contengono il valore letto o da scrivere
 - NUMLOC, registro contatore utilizzato nelle operazioni di lettura e scrittura (per capire quando abbiamo finito).
- Registri a supporto dell'esecuzione delle istruzioni:
 - OPCODE, registro contenente gli 8 bit con formato e istruzione.
 - SOURCE e DEST_ADDR, registri contenenti dati sugli operandi. Non è detto che siano usati sempre. Ricordiamoci che del sorgente ci interessa il contenuto, mentre del destinatario l'indirizzo. Più avanti è presente un pdf di Corsini che spiega i vari casi relativamente ai formati.
 - MJR, registro per i salti (spiegato qualche lezione fa)



- **Fasi del processore:**



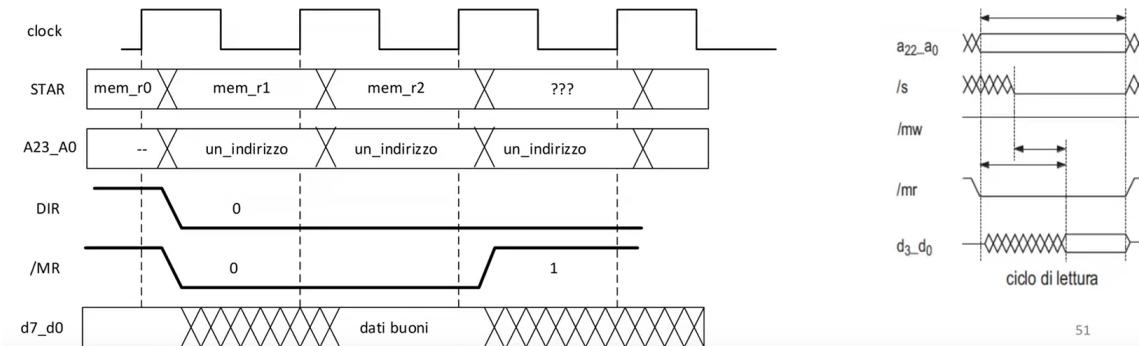
- **Fase di reset:** inizializzazione del contenuto dei registri (F, IP, ma anche altri).
 - IP <- 'HFF0000'
 - F <- 0
 - Tutti i registri che reggono variabili di comando dovranno essere inizializzati in modo coerente: in particolare tutti gli attivi bassi dovranno assumere come valore 1.
 - Dobbiamo anche mettere in alta impedenza la porta tri-state. Segue DIR inizializzato a 0, cambiamo il suo valore solo quando dovremo scrivere sul bus (ricordiamoci che quando la porta tristate del processore si comporta in un modo quella presente dall'altra parte – per esempio quella in memoria principale – dovrà comportarsi in modo opposto).
 - STAR verrà inizializzato col primo stato della fase di fetch.
 - **Fase di fetch:**
 - Preleva un byte dalla memoria, all'indirizzo IP
 - Incrementa IP (punta al byte successivo)
 - Controlla che il bit appena letto corrisponda all'OPCODE di una delle istruzioni note.
 - Il contenuto del byte letto deve essere copiato nel registro OPCODE. Inoltre dobbiamo valutare il formato dell'istruzione (tre bit più significativi). Valutare il formato significa decidere cosa fare (per ogni formato abbiamo una certa procedura)
 - Per alcuni formati dovremo incrementare IP opportunamente (in base al numero di bit da leggere)
 - In alcuni formati dobbiamo procurarci l'indirizzo dell'operando destinatario, e inserirlo in DEST_ADDR. Nel formato F3 l'indirizzo già sta in DP, mi basta copiarlo in DEST_ADDR. Nei formati F6 e F7 devo andarlo a leggere in memoria: incremento di 3 il valore di IP, leggo 3 bit e li sposto nel processore.
 - In altri formati, F0 ed F1, il processore non fa niente in fase di fetch.
 - Ultima cosa è la lettura del contenuto di OPCODE (i cinque bit meno significativi): devo capire qual è l'istruzione da seguire.
 - **Fase di esecuzione**
 - Il processore esegue l'istruzione che ha decodificato.
 - Torna nella fase di fetch a meno che non stia eseguendo l'istruzione di HLT.
 - **Stato di blocco:** raggiunto nei seguenti casi:
 - OPCODE non valido;
 - istruzione HLT.
- Non è possibile uscire da questo stato: l'unico modo è fare reset.

- **Letture e scritture in memoria e spazio di I/O:**

- o Il processore legge in memoria durante la fase di fetch.
- o Durante la fase di esecuzione il processore dovrà leggere e scrivere in memoria (RET, MOV) o nello spazio di I/O (IN, OUT)
- o **Lettura e scrittura in memoria:**
 - Ricordarsi della struttura della RAM statica (multiplexer, demultiplexer, variabili di comando, RC per le variabili di pilotaggio).
 - Ricordarsi la temporizzazione delle RAM statiche in lettura e scrittura
 - **Cosa significano queste cose dal punto di vista del processore?**

Vediamo il ciclo di lettura:

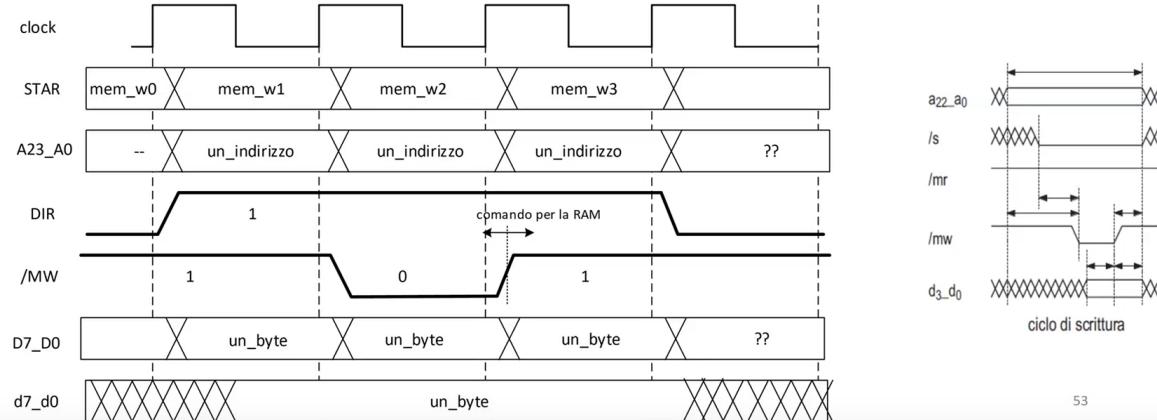
```
mem_r0: begin A23_A0<=un indirizzo; DIR<=0; STAR<=mem_r1; end
mem_r1: begin STAR<=mem_r2; end //stato di wait
mem_r2: begin QUALCHE_REGISTRO<=d7_d0; MR_<=1; ..... ; end
```



- Nel primo stato imposto l'indirizzo uguale a qualcosa, pongo DIR uguale a 0 (finché non ho dati sensati rimane a 0). Pongo MR_a 0 e indico il passaggio alla stato successivo (micro-istruzione a singolo step).
- Il secondo stato è detto stato di wait, utilizzato per dare alla memoria tempo per rispondere. D'ora in poi lo daremo per scontato, in modo tale da ottenere descrizioni più semplici.
- Il terzo stato è quello in cui abbiamo dati affidabili: li campioniamo in quale registro e riportiamo MR_a 1.
- **Letture successive:** nell'ultimo stato non alzo MR_ e pongo un nuovo indirizzo da leggere.
- **Domanda:** posso alzare DIR a 1 nell'ultimo stato assieme a MR_? No, perché le tristate della memoria sono ancora in conduzione e ci mettono più tempo ad alzarsi rispetto alle tristate del processore. Se alzo insieme i due valori creo, per poco tempo, una situazione in cui entrambe le tristate sono attive.

Vediamo il 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
```

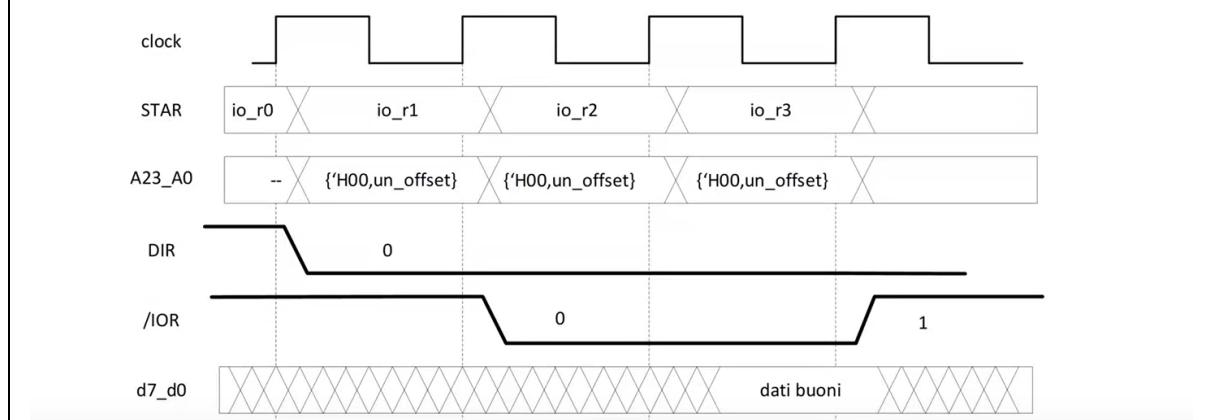


- Operazione distruttiva. Sappiamo che dobbiamo attendere la stabilizzazione di /s e degli indirizzi prima di portare giù /mw.
- I dati, ricordiamo, devono essere corretti a cavallo del fronte di salita.
- Primo stato: setto l'indirizzo, pongo i dati che voglio salvare in ingresso, alzo DIR e indico il passaggio allo stato successivo.
- Secondo e terzo stato: stati di attesa.
- Terzo stato: abbasso DIR, abbiamo scritto.
- **Posso scrivere in D7_D0 nel terzo stato?** No, altrimenti non sarebbe rispettata la regola di temporizzazione principale (dati costanti a cavallo del fronte di salita).
- **Posso portare DIR a 0 direttamente nel secondo stato?** No, altrimenti i dati non rimangono stabili.
- **Posso porre un nuovo indirizzo nel terzo stato?** No, devo aspettare il clock successivo.

○ **Letture nello spazio di I/O:**

Vediamo la lettura

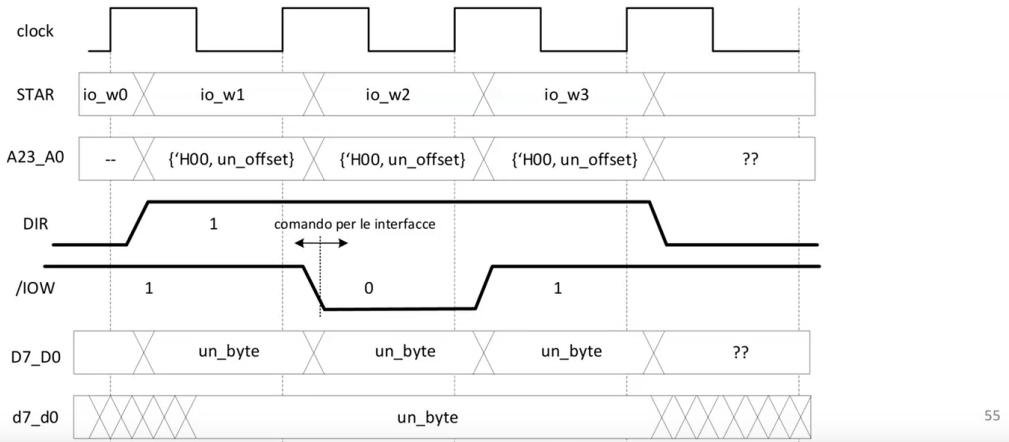
```
io_r0: begin A23_A0<='H00,un_offset}; DIR<=0; STAR<=io_r1; end
io_r1: begin IOR_<=0; STAR<=io_r2; end
io_r2: begin STAR<=io_r3; end //stato di wait
io_r3: begin QUALCHE_REGISTRO<=d7_d0; IOR_<=1; .... ; end
```



- Le letture sono simili, MA NON UGUALI, alle lettura in memoria.
- Differenza: MR_ può essere portato a 0 nello stesso stato in cui setto gli indirizzi. La lettura in memoria è un'operazione non distruttiva, se gli indirizzi ballano non è un problema. Nello spazio di I/O anche la lettura può essere distruttiva: se io leggo da un'interfaccia un dato questa si regola col suo dispositivo per svolgere operazioni di scrittura. Quindi posso avere dati appena letti subito sovrascritti.
- Nell'ultimo stato posso assegnare il valore in uscita (d7_d0) a qualche registro. Alzo IOR_ e sono obbligato a farlo: in caso di nuove lettura siamo obbligati a finire un ciclo e aprirne un altro.

Vediamo la scrittura:

```
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
```



- Nel ciclo di scrittura in memoria l'assegnamento a D7_D0 poteva essere spostato nello stato successivo. Nella scrittura nello spazio di I/O dobbiamo avere i dati già pronti a cavallo del fronte di discesa di IOW_: questo perché alcune interfacce memorizzano sul fronte di discesa, e non su quello di salita. Segue che D7_D0 deve essere assegnato per forza nel primo stato e che non possa essere spostato.

- Accessi per più di un byte alla memoria:

- Il processore deve poter leggere operandi a 2 e 3 byte.
- Fa comodo strutturarsi di micro-sottoprogrammi di lettura/scrittura modulari. Questi sottoprogrammi possono essere riferiti ogni volta che dobbiamo leggere più di un byte.
- Registri utilizzati:**
 - Il registro MJR per salvare il micro-indirizzo di ritorno, cioè lo stato su cui devo tornare dopo aver eseguito il micro-sottoprogramma.
 - I registri APP0, APP1, APP2, APP3 per salvare i byte letti o da scrivere.
 - Il registro NUMLOC come contatore del numero di byte da leggere/scrivere.
- Lettura**
 - I micro-sottoprogrammi sono i seguenti:
 - readB (1 byte)
 - readW (2 byte)
 - readM (3 byte)
 - readL (4 byte, non lo utilizzeremo)
 - I parametri in ingresso sono A23_A0, che contiene il primo indirizzo in memoria (modificato dai micro-sottoprogrammi), DIR a zero e il valore di MJR (dobbiamo indicare

da dove ricominciare dopo aver terminato l'esecuzione del micro-sottoprogramma).

▪ Codice del microprogramma:

// PER L' ESECUZIONE

Dati in ingresso

Sx: **begin** ... A23_A0<=un indirizzo; MJR<=Sx+1; STAR<=readB; **end**
Sx+1: **begin** ... <utilizzo di APP0> **end**

Chiamata di un micro-sottoprogramma

*Utilizzo dei dati letti dal micro-sottoprogramma.
Questi dati sono recuperati dai registri di supporto*

// MICROSOTTOPROGRAMMA PER LETTURE IN MEMORIA

readB: **begin** MR_<=0; DIR<=0; NUMLOC<=1; STAR<=read0; **end**

readW: **begin** MR_<=0; DIR<=0; NUMLOC<=2; STAR<=read0; **end**

readM: **begin** MR_<=0; DIR<=0; NUMLOC<=3; STAR<=read0; **end**

readL: **begin** MR_<=0; DIR<=0; NUMLOC<=4; STAR<=read0; **end**

Inizializzazione delle operazioni di lettura. Abbasso MR_ visto che voglio svolgere l'operazione, indico il numero di locazioni da leggere in NUMLOCK e passo alla prima lettura

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**

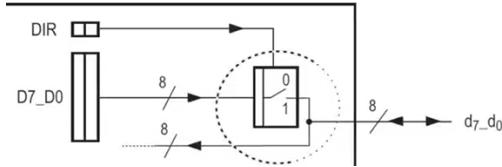
read3: **begin** APP3<=d7_d0; A23_A0<=A23_A0+1; STAR<=read4; **end**

read4: **begin** MR_<=1; STAR<=MJR; **end**

Operazioni di lettura. Salvo il byte letto nell'apposito registro APPx. Dopo aver letto e spostato nel processore mi chiedo se ho svolto il numero di letture necessarie. Se ho finito abbasso MR_ e indico come stato successivo quello posto all'inizio in MJR.

○ Scrittura

- I micro-sottoprogrammi sono i seguenti:
 - writeB (1 byte)
 - writeW (2 byte)
 - writeM (3 byte)
 - writeL (4 byte)
- I parametri in ingresso sono A23_A0, che contiene il primo indirizzo in memoria su cui lavorare (verrà modificato dai micro-sottoprogrammi), DIR a 0 (ci pensa il sottoprogramma ad alzarlo e poi ad abbassarlo), APPj ($j=0 \dots 3$) con i byte da scrivere, MJR (come prima).



Ricordarsi a cosa serve DIR. Se la porta tristate ha 1 la "levetta" è alzata.

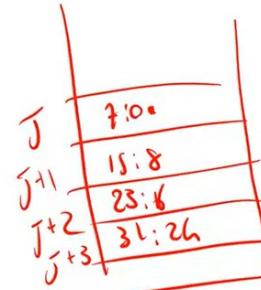
- Codice del microsottoprogramma:

```
// PER L'ESECUZIONE
Sx: begin ... APP1<=dato 16 bit[15:8]; APP0<=dato 16 bit[7:0]; ... end
A23 A0<=un indirizzo; MJR<=Sx+1; STAR<=writeW; end
Sx+1: begin ... <proseguimento del programma dopo la scrittura> end
```

Solite cose fatte prima, cambiano solo gli ingressi e i micro-sottoprogrammi chiamati

```
// MICROSOFTOPROGRAMMA PER SCRITTURE IN MEMORIA
// PIU' NOIOSO, PER SVOLGERE OGNI SINGOLA SCRITTURA SERVE UN CICLO IN PIU'
// Metto il primo indirizzo su cui scrivo, alzo DIR perché devo svolgere operazioni
// di lettura, indico il numero di scritture da fare, infine cambio stato per svolgere
// la prima scrittura.
// In ogni ciclo di scrittura prima abbasso MW_, poi lo rialzo (ogni volta)
```

```
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
```

Ricordare l'ordine delle cifre nei vari bit.

```
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
```

Ogni volta:

- Indico l'indirizzo su cui voglio scrivere (nel caso della prima scrittura è indicato come parametro in ingresso) e i valori da porre sui fili di dati D7_D0.
- Dopo aver posto questi dati, E NON PRIMA, abbasso MW_ a 0.
- Al ciclo successivo rialzo e verifico se ho scritto il numero di byte richiesti.
- Se ho finito vado all'ultimo stato del micro-sottoprogramma: riabbasso la levetta della porta tristate e pongo come stato successivo quello indicato in MJR all'inizio.
- **Promemoria:** negli indirizzi più bassi si pongono le cifre meno significative.

Descrizione in Verilog del processore

Vediamo la descrizione in Verilog del processore. Nel corso della descrizione faremo uso di reti combinatorie in grado di semplificarcisi alcune operazioni. Non ci dedicheremo alla descrizione accurata di queste funzioni: sono estremamente noiose, ma non impossibili da scrivere per noi.

```

//-----
// 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 REGISTRIS
    reg DIR;
    reg [7:0] D7_D0;                                Definisco i registri di supporto alle variabili di uscita e definisco come valore di queste uscite quelli dei corrispondenti registri
    reg [23:0] A23_A0;
    reg MR_,MW_,IOR_,IOW_;
    assign mr_ = MR_;                               Evidenziate in rosso due righe con sintassi nuova: con inout indichiamo fili che possono fungere sia come ingressi che come uscite, con 'HZZ indichiamo l'alta impedenza (in questo caso se DIR == 0)
    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; Al più lavoriamo su 4 byte, quindi bastano 3 bit.
    reg [7:0] AL,AH,F,OPCODE,SOURCE,APP3,APP2,APP1,APP0;
    reg [23:0] DP,IP,SP,DEST_ADDR; I registri a 24 bit contengono indirizzi

    // REGISTRO DI STATO, REGISTRO MJR E CODIFICA DEGLI STATI INTERNI
    reg [6:0] STAR,MJR;                            Con 7 bit rappresentiamo 86 stati interni diversi!
    parameter fetch0=0, .... write11=86;

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

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

    function [7:0] alu_result;
        input [7:0] opcode,operand1,operando2;
        ... ...
    endfunction

    function [3:0] alu_flag;
        input [7:0] opcode,operand1,operando2;
        ... ...
    endfunction

```

Prende in ingresso un byte e restituisce 1 se quel byte è l'OPCODE di un'istruzione nota, 0 altrimenti

Prende in ingresso 7 bit, interpretato come OPCODE valido, restituisce la codifica del primo stato interno dell'esecuzione dell'istruzione relativa.

Simulo lo ALU interno al processore. Interpretro i 3 byte passati come un OPCODE, un operando sorgente e un operando destinatario. Restituisco il risultato su 8 bit dell'elaborazione svolta. Posso porre come OPCODE add, sub, and, or ...

Prendo in ingresso gli stessi byte di alu_result e aggiorno in flag rendendoli consistenti. Restituisco 4 bit che consistono nei 4 flag significativi (quelli che ci interessano di più) del registro F.

```

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

// 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
  Inizializzazione di registri importanti: primo indirizzo di istruzione, reset dei flag, porta tristate in alta impedenza, attivi bassi alzati, prima istruzione di fetch come stato successivo.

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

//-----
// FASE DI FETCH (CHIAMATA)
Ricordarsi le fasi:
  - LEGGI GLI OPCODE
  - PROCURATI GLI OPERANDI
  - ESECUZIONE DELL'OPERAZIONE CON OPCODE VALIDO
// Pongo come indirizzo quello presente nell'IP e lo incremento (ricordare, esecuzione in parallelo, non in sequenza)
fetch0: begin A23_A0<=IP; IP<=IP+1; MJR<=fetch1; STAR<=readB; end
  Operazione di lettura di un byte.
fetch1: begin OPCODE<=APP0; STAR<=fetch2; end
  Istruzione di ritorno. Passiamo a fetch1 dopo la lettura.
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;
  Se l'OPCODE è valido passo a fetch3, altrimenti ad nvi
end
  Porto il byte letto in OPCODE e passo allo stato successivo
  Nessun problema a eseguirlo nello stesso stato, valid_fetch non dipende da MJR.

// SALTO AL PRIMO PASSO SPECIFICO DELLA FASE DI FETCH (Lo abbiamo determinato prima)
fetch3: begin STAR<=MJR; end

```

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 (010)

- Categoria in cui rientrano le istruzioni dove l'operando sorgente si trova in memoria ed è indirizzato tramite DP.
- Il sorgente deve essere ripescato in memoria. Dovrà fare una seconda lettura in memoria per portare l'operando sorgente dentro il processore.

POP DP	00010000
RET	00010001

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

○ Formato F3 (011)

- Categoria in cui rientrano le istruzioni dove l'operando destinatario si trova in memoria ed è indirizzato usando DP (solo le MOV)
- Codifico su un unico byte l'istruzione. La fase di fetch consiste nel non fare niente: il contenuto da spostare è già presente nel processore, stessa cosa l'indirizzo da raggiungere.
- La scrittura del destinatario avviene in fase di esecuzione.

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

○ Formato F4 (100)

- Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo immediato, e sta su 8 bit.
- L'istruzione è lunga due byte: il primo contiene l'istruzione, il secondo l'operando indirizzato in modo immediato. La fase di fetch consiste nel fare due letture consecutive.

```
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 (101)

- Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo diretto. Ciò pongo direttamente l'indirizzo del sorgente.
- In fase di Fetch l'operando sorgente deve essere riportato nel processore.
- L'operazione sarà lunga 4 byte: uno di opcode e tre di indirizzo di memoria (24 bit per poter rappresentare qualunque indirizzo). Seguono tre cicli di lettura consecutivi a partire da IP. Ciò non basta: devo fare un'altra lettura all'indirizzo trovato: a quel punto ho raggiunto l'operando sorgente e posso porlo nel processore.

```
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
```

○ Formato F6 (110)

- Categoria in cui rientrano le istruzioni dove l'operando destinatario è in memoria, indirizzato in modo diretto.
- Il processore dovrà leggere 4 byte in memoria: uno per l'opcode, tre per l'indirizzo del destinatario.
- La scrittura del destinatario avviene in fase di esecuzione.

```
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
```

○ Formato F7 (111)

- Uguale al precedente, raggruppa le istruzioni di controllo (CALL, JMP, Jcon) in cui ho un indirizzo di salto.
- Utilizzo un byte per l'opcode, altri tre per l'indirizzo. In fetch abbiamo la lettura di 4 byte consecutivi, a partire da IP.

```
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
// Eseguo se l'OPCODE non è valido (rivedere fetch2)
nvi: begin STAR<=nvi; end
// TERMINAZIONE REGOLARE CON PASSAGGIO ALLA FASE DI ESECUZIONE
// Individuo il primo passo da eseguire per la fase di esecuzione.
// Osservazione per bimbi spastici: perché il salto lo faccio un passo dopo?
Ricordiamo che le istruzioni sono eseguite in parallelo, non in sequenza!
fetchEnd: begin MJR<=first_execution_state(OPCODE); STAR<=fetchEnd1; end
fetchEnd1: begin STAR<=MJR; end

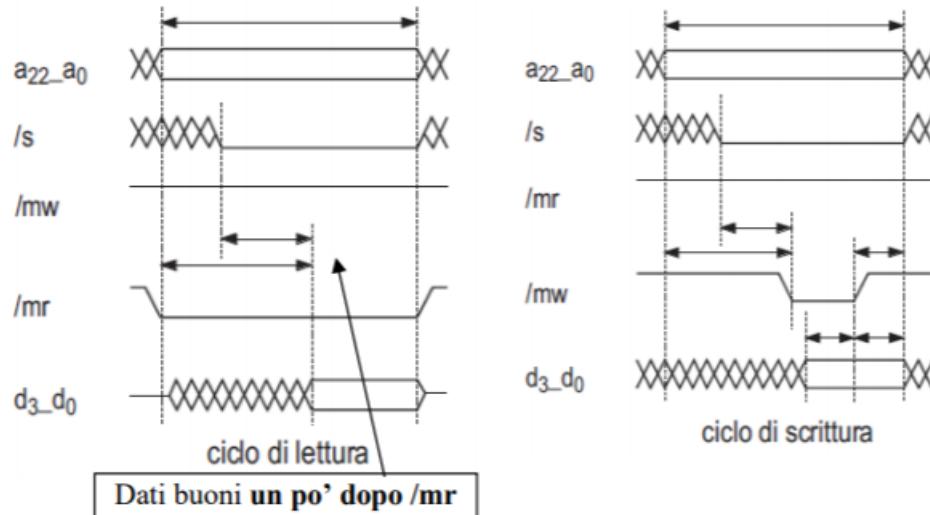
```

All'uscita dalla fase di fetch avrà:

- L'OPCODE che contiene il codice operativo dell'istruzione
- Se l'istruzione ha un operando sorgente immediato o in memoria questo è in SOURCE.
- Se l'istruzione ha un operando destinatario in memoria il suo indirizzo sta in DEST_ADDR.
- IP è stato incrementato e punta alla prossima istruzione da prelevare (ESECUZIONE IN PARALLELO, NON IN SEQUENZA).

3.1 Ricapitoliamo sulle operazioni di lettura e scrittura...

3.1.1 Memoria principale



- Quando scriviamo il codice Verilog dobbiamo tenere conto di quanto già visto nello spiegare la temporizzazione di scrittura e lettura in una memoria RAM statica:

– Lettura:

- * Operazione non distruttiva. mr può essere abbassato fin da subito.
- * L'indirizzo può essere indicato fin da subito, ma anche dopo: bisogna tenere conto della necessità di aspettare un po' prima che i dati restituiti sui fili di dati si stabilizzino (considerare la struttura della memoria RAM statica). Ricordarsi che la select non si stabilizza immediatamente.

– Scrittura:

- * Operazione distruttiva. mw non può essere abbassato subito.
- * Possiamo indicare fin da subito l'indirizzo, ma mw può essere abbassato solo dopo che si è stabilito l'indirizzo (se l'indirizzo non si è stabilito si corre il rischio di scrivere in posti sbagliati).
- * Non sono importanti i valori posti sui fili di dati all'inizio, ma quelli alla fine. Per evitare situazioni imprevedibili i dati devono rimanere costanti a cavallo del fronte di salita di mw.

- Quindi:

– Lettura:

```
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; end
```

- * **Primo stato:** si pone l'indirizzo dove leggere, si pone la porta tristate in alta impedenza (ricordiamoci che dobbiamo fare l'opposto di quanto avviene in memoria principale, se in memoria si ha un'operazione di lettura allora avrà la porta tristate in conduzione), si abbassa subito MR.

- * ... si attende quanto necessario affinchè si stabilizzino i dati. Abbiamo detto che supponiamo che la rete è subito pronta per evitare di allungare le descrizioni
- * **Terzo stato:** si memorizza il contenuto in ingresso nel processore da qualche parte. A questo punto:
 - se abbiamo finito alziamo mr;
 - altrimenti poniamo subito un nuovo indirizzo.

– **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
```

- * **Primo stato:** indichiamo l'indirizzo, i dati da scrivere nell'indirizzo, si pone la porta tristate in conduzione (stesso discorso di prima). Contrariamente a prima NON POSSIAMO abbassare subito mw.
- * ... si attende quanto necessario affinchè si stabilizzi l'indirizzo.
- * **Secondo stato:** adesso possiamo abbassare mw. L'indirizzo si è stabilizzato e non corriamo il rischio di scrivere in posti sbagliati
- * **Terzo stato:** alziamo mw. I dati a cavallo di questo fronte di salita sono quelli che saranno memorizzati.
- * **Quarto stato:** Riporto DIR a 0, quindi pongo la porta in alta impedenza. Si osservi che non è possibile fare questo prima, visto che dobbiamo mantenere i dati stabili sul fronte di salita. Dobbiamo rimettere la porta tristate ogni volta in alta impedenza, anche nel caso in cui si voglia svolgere più operazioni di scrittura: segue un numero di stati maggiori (vedere il sottoprogramma per operazioni di scrittura su più byte).

3.1.2 Sottosistema di I/O

- Presenti differenze non banali rispetto alle operazioni in memoria. Il numero di bit è minore (solo 16 bit), si utilizzano variabili di pilotaggio diverse (ior e iow, non mr e mw).

• **Lettura:**

```
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
```

- **Primo stato:** pongo l'indirizzo della locazione da leggere, pongo la porta tristate in alta impedenza (solito discorso di prima).
- **Secondo stato:** abbasso ior. L'operazione di lettura è distruttiva, poichè può provocare nelle interfacce conseguenti operazioni di scrittura (potrei avere dati appena letti subito sovrascritti).
- ... attendo

- **Quarto stato:** memorizzo i dati appena ricevuti in un registro e alzo ior. Contrariamente all’operazione di lettura non possiamo mettere subito indirizzi (ricordiamo che la lettura in I/O è distruttiva)

- **Scrittura:**

```

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

```

- **Primo stato:** indichiamo l’indirizzo, i dati da scrivere nell’indirizzo, poniamo la porta tristate in conduzione (soliti motivi di prima). Contrariamente a prima i dati devono essere già pronti sul fronte di discesa di iow: questo perchè molte interfacce memorizzano sul fronte di discesa di iow, e non sul fronte di salita.
- **Secondo stato:** abbassiamo iow. Ricordiamo che la scrittura è distruttiva, quindi la cosa va fatta solo dopo la stabilizzazione dell’indirizzo dove scrivere.
- **Terzo stato:** alzo iow. Valgono gli stessi discorsi di prima (soprattutto se l’interfaccia scrive in caso di fronte di salita).
- **Quarto stato:** pongo la porta tristate in alta impedenza, visto che abbiamo finito. Non possiamo farlo prima.

3.2 Ricapitoliamo sui sottoprogrammi per la lettura/scrittura

- Si consideri che ogni volta dobbiamo indicare
 - l’indirizzo dove scrivere o leggere,
 - lo stato interno dove ritornare dopo aver eseguito il sottoprogramma (con il registro MJR), e
 - il numero di byte da leggere (che poniamo in modo implicito indicando il primo stato di partenza del sottoprogramma)
- **Lettura** (parametro di ingresso l’indirizzo):
 - **Primo stato:** pongo la porta tristate in alta impedenza (registro DIR), abbasso mr, indico il numero di locazioni da leggere (registro NUMLOC)
 - **Stati successivi:** ho, per ogni step possibile, uno stato interno. In ciascuno di essi
 - * memorizzo il valore letto nel relativo registro di supporto,
 - * incremento il registro contenente l’indirizzo,
 - * decremento il registro NUMLOC (che funge da contatore), e
 - * determino il passaggio allo step successivo o allo stato finale del sottoprogramma sulla base del valore non decrementato di NUMLOC.
 - **Stato finale:** alzo mr e indico come nuovo stato interno quello memorizzato nel registro MJR.

- **Scrittura** (parametro di ingresso l'indirizzo e i valori da memorizzare, posti nei registri APPx):
 - **Primo stato:** pongo la porta tristate in conduzione (registro DIR), indico il numero di locazioni da leggere (registro NUMLOC), indico come valore dei fili di dati quello del primo dei registri di supporto (APP0). Non posso abbassare subito mw, visto che devo avere valori stabili prima del fronte di discesa.
 - **Stati successivi:** ogni step di scrittura di un byte non può essere gestito con un solo stato interno (contrariamente alle operazioni di lettura).
 - * **Primo stato step:** abbasso mw.
 - * **Secondo stato step** alzo mw, e attraverso il numero di locazioni (non ancora decrementato) verifico se ho finito. Non posso modificare subito i fili di dati e i fili di indirizzo: i dati, RIBADIAMO, devono essere costanti a cavallo del fronte di salita.
 - * **Terzo stato step:** se siamo a questo punto significa che dobbiamo scrivere un altro byte. Modifco i fili di dati indicando il successivo registro APPx, incremento il registro con l'indirizzo e decremento NUMLOC.
 - **Stato finale:** pongo la porta tristate in alta impedenza (registro DIR) e indico come nuovo stato interno quello memorizzato nel registro MJR.

Processore SEP8

Contenuto di alcuni registri alla fine della fase di chiamata

(F0, F1, F2, F3, F4, F5, F6, F7 sono i formati delle istruzioni di detto processore)

	OPCODE	SOURCE	DEST_ADDR	
F0, F1	codice operativo	non significativo	non significativo	
F2, F4, F5	codice operativo	operando sorgente	non significativo ¹	Indirizzamento con registro puntatore per l'operando sorgente Indirizzamento immediato per l'operando sorgente Indirizzamento diretto per l'operando sorgente
F3, F6	codice operativo	non significativo ²	Indirizzo in memoria dell'operando destinatario	Indirizzamento con registro puntatore dell'operando destinatario Indirizzamento diretto per l'operando destinatario
F7	codice operativo	non significativo	Indirizzo in memoria dell'istruzione a cui saltare	Istruzioni di salto Istruzioni di chiamata dei sottoprogrammi

¹ l'operando destinatario sarà immesso nel registro AL o nel registro AH

² l'operando sorgente è nel registro AL o nel registro AH

3.4 Ricapitoliamo sugli stati della fase di fetch

Si tenga conto che al termine della fase di fetch avremo

- l'OPCODE nel registro OPCODE
- L'operando sorgente nel registro SOURCE, se il sorgente è immediato o in memoria (e non è un registro)
- L'operando destinatario nel registro DEST_ADDR, se il destinatario è in memoria (e non è un registro)

Step

- **fetch0:** Prendo l'indirizzo memorizzato nell'IP, incremento l'indirizzo memorizzato nell'IP (la cosa avviene in parallelo), indico col registro MJR che lo stato da richiamare più avanti è fetch1, avvio un'operazione di lettura del contenuto all'indirizzo appena estratto. Abbiamo ottenuto l'OPCODE.
- **fetch1:** Memorizzo nel registro OPCODE il risultato dell'operazione di lettura (un byte memorizzato nel registro di supporto APP0).
- **fetch2 e fetch3:** Utilizzo il registro MJR per gestire un salto a tante vie. Devo capire come ho gestito gli operandi, quindi quale formato ho adottato.
 - Gestisco il formato adottato:
 - **F0:** si salta diretto a fetchEnd, abbiamo già gli operandi nel processore.
 - **F1:** si salta diretto a fetchEnd, rimandiamo la gestione degli operandi alla fase di esecuzione.
 - **F2:** eseguiamo un'operazione di lettura di un byte nell'indirizzo puntato dal registro DP. Si memorizza il risultato della lettura (posto in APP0) nel registro SOURCE.
 - **F3:** del destinatario non ci interessa il contenuto, ma solo l'indirizzo. Mi limito a copiare l'indirizzo puntato da DP nel registro DEST_ADDR.
 - **F4:** l'operando sorgente è posto nel byte immediatamente successivo a quello dell'OPCODE. Svolgo un'operazione di lettura di un byte nella locazione successiva, e pongo il risultato della lettura nel registro SOURCE. Ovviamente incrementiamo IP una volta in più (ricordarsi che le istruzioni sono eseguite in parallelo, non in sequenza).
 - **F5:** l'operando sorgente è un indirizzo memorizzato su 3 byte successivi. Svolgo un'operazione di lettura su 3 byte (readM) per recuperare questo indirizzo. Dopo aver recuperato l'indirizzo (memorizzato in APP2, APP1, APP0) svolgo un'ulteriore operazione di lettura (su un byte) per recuperare il contenuto della locazione puntata dall'indirizzo. Pongo il risultato di quest'ultima lettura nel registro SOURCE.
 - **F6:** svolgo un'operazione di lettura su 3 byte, come prima, per recuperare l'indirizzo posto in modo diretto. Il risultato della lettura viene posto in DEST_ADDR (ricordarsi che del destinatario ci interessa solo l'indirizzo, non il contenuto).
 - **F7:** le istruzioni di salto presentano solo l'operando destinatario. Svolgo un'operazione di lettura su 3 byte per recuperare l'indirizzo, come prima, e pongo come contenuto di DEST_ADDR l'indirizzo appena estratto.
- **fetchEnd e fetchEnd1:** conclusione della fase di fetch. Memorizzo nel registro MJR il risultato di una rete combinatoria. Questa restituisce il prossimo stato a cui saltare. Con questo salto concluso si passa alla fase di esecuzione dell'istruzione.

Capitolo 4

Mercoledì 02/12/2020

Continuiamo l'analisi della descrizione Verilog con la fase di esecuzione. La cosa sarà semplice:

- parte della complessità è stata già gestita in fase di fetch;
- la maggior parte delle istruzioni complesse sono implementate tramite reti combinatorie, quindi si riducono a un semplice assegnamento a registro.

```
//----- istruzione NOP -----
// Istruzione che non fa niente. Ho un passo in più, ritorno subito al primo step di fetch.
nop: begin STAR<=fetch0; end

//----- istruzione HLT -----
// Loop infinito da cui si esce solo premendo reset
hlt: begin STAR<=hlt; end

//----- istruzione MOV AL,AH -----
// Si assegna al registro AH il contenuto del registro AL
ALtoAH: begin AH<=AL; STAR<=fetch0; end

//----- istruzione MOV AH,AL -----
// Si assegna al registro AL il contenuto del registro AH
AHToAL: begin AL<=AH; STAR<=fetch0; end

//----- istruzione INC DP -----
// Incremento il contenuto del registro DP
incDP: begin DP<=DP+1; STAR<=fetch0; end

//----- istruzioni MOV (DP),AL -----
// MOV $operando,AL
// MOV indirizzo,AL
// Assegno al registro AL il contenuto del registro SOURCE (ricordiamo quanto detto nella scorsa lezione)
ldAL: begin AL<=SOURCE; STAR<=fetch0; end

//----- istruzioni MOV (DP),AH -----
// MOV $operando,AH
// MOV indirizzo,AH
// Assegno al registro AH il contenuto del registro SOURCE (uguale a prima, registro destinatario diverso)
ldAH: begin AH<=SOURCE; STAR<=fetch0; end

//----- istruzioni MOV AL,(DP) -----
// MOV AL,indirizzo
// Indirizzo dell'operando destinatario nel registro DEST_ADDR. Contrariamente a prima dobbiamo svolgere
// un'operazione di scrittura al di fuori del processore.
storeAL: begin A23_A0<=DEST_ADDR; APP0<=AL; MJR<=fetch0; STAR<=writeB; end

//----- istruzioni MOV AH,(DP) -----
// MOV AH,indirizzo
// Stessa cosa di prima, cambia solo il valore assegnato al registro APP0
storeAH: begin A23_A0<=DEST_ADDR; APP0<=AH; MJR<=fetch0; STAR<=writeB; end
```

- Raggruppa tutte le istruzioni che non possono essere classificate nei precedenti formati
 - Istruzioni di I/O
 - operando indirizzo a 16 bit, sorgente (IN) o destinatario (OUT)
 - MOV con uno dei registri a 24 bit (DP o SP)
 - Operando a 24 bit sorgente, sia immediato che diretto, o destinatario
- Le azioni per procurarsi gli operandi sono diverse da un'istruzione all'altra
 - Meglio fare una fase di fetch «scarna» in cui prelievo solo l'opcode
 - Gestiremo gli operandi successivamente in fase di esecuzione
- Poco pulito dal punto di vista concettuale, ma molto più semplice

73

```

----- istruzione MOV $operando,SP -----
// L'operando SP è a 24bit. IP è stato incrementato di 1 quando arriviamo qua. Incremento di tre andando al primo
bit che non mi interessa. Svolgo un'operazione di lettura readM (3 byte). Come indirizzo da leggere ho
l'Instruction Pointer, come MJR lo stato successivo in cui utilizzo i valori letti. L'unione dei tre registri modificati
con l'operazione di lettura consiste nel valore dell'SP (In ordine decrescente, ricordiamo dove stanno le cifre più
significative e quelle meno significative).
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 -----
// Anche in questo caso abbiamo un operando (quello sorgente) a 24 bit. Facciamo le stesse cose di prima, cambia
solo l'assegnamento a DP invece che ad SP.
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 -----
// Devo leggere l'indirizzo, quindi svolgo una lettura di 3 byte. Dopo aver estratto l'indirizzo svolgo un'ulteriore
operazione di lettura per leggere il contenuto dell'indirizzo. Pongo il contenuto dell'indirizzo come nuovo valore
del registro 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

----- istruzione MOV DP,indirizzo -----
// Mi devo procurare l'indirizzo (l'operando destinatario) con un'operazione di lettura su 3 byte. Successivamente
svolgo un'operazione di scrittura sull'indirizzo trovato. Questa istruzione, suggeriva qualcuno, potrebbe essere
accorpata ad F6: mi sbarazzo del primo step e nel secondo assegno DEST_ADDR invece di {APP2,APP1,APP0}
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 -----
// Per prima cosa devo procurarmi l'offset: svolgiamo un'operazione di lettura su 2 byte a partire dall'IP. Dopo la
lettura prendo i primi 16 bit (gli unici che mi interessano). A questo punto devo fare una lettura nello spazio di I/O
in: begin A23_A0<=IP; IP<=IP+2; MJR<=in1; STAR<=readW; end

// Preparo l'indirizzo, allo step successivo abbasso IOR_(DOPO), nello step finale prendo i dati letti e li metto nel registro. Alzo
IOR_visto che ho finito l'operazione.
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 -----
// Leggo l'offset e svolgo operazione di lettura. Ricordarsi che non possiamo abbassare DIR e alzare IOW_ in
contemporanea (altrimenti abbiamo problemi con la porta tristate). Il comando di scrittura nello spazio di I/O è il
fronte di discesa.
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

----- 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
SHR AL
SHL AL

// Tutte le istruzioni elencate possono essere
collassate nel seguente codice. Si assegna ad AL il risultato di un'operazione
combinatoria dove indico OPCODE, SOURCE, AL. Dall'OPCODE capisco quale operazione
dovrà essere svolta, gli altri due consistono negli operandi. Le operazioni possibili
sono ADD, SUB, AND, OR, CMP, NOT, SHL, SHR: segue un multiplexer a otto vie con
variabili di comando generate dall'OPCODE (Rete combinatoria dipendente
dall'implementazione scelta dell'OPCODE).

Dobbiamo tener conto anche dell'aggiornamento dei flag: ricordiamo che il registro
dei flag consiste in un insieme di 8 bit di cui i primi 4 importanti. Attraverso
l'assegnamento non bloccante aggiorniamo i flag nelle posizioni 0,1,2,3
(OF,SF,ZF,CF): facciamo in questo modo visto che gli altri flag, in queste
operazioni, sono insignificanti.

aluAL: begin
    AL <= alu_result(OPCODE, SOURCE, AL);
    F<={F[7:4], alu_flag(OPCODE, SOURCE, AL)};
    STAR<=fetch0;
end

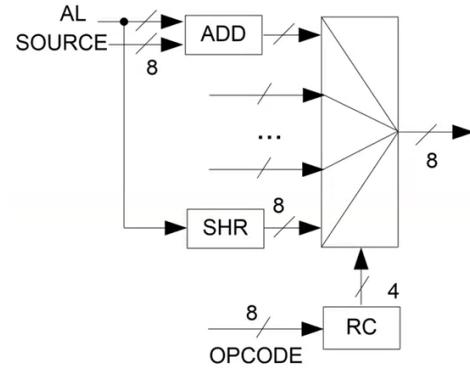
//----- istruzioni ADD (DP),AH --
    ADD $operando,AH
    ADD indirizzo,AH
    SUB (DP),AH
    SUB $operando,AH
    SUB indirizzo,AH
    AND (DP),AH
    AND $operando,AH
    AND indirizzo,AH
    OR (DP),AH
    OR $operando,AH
    OR indirizzo,AH
    CMP (DP),AH
    CMP $operando,AH
    CMP indirizzo,AH
    NOT AH
    SHL AH
    SHR AH

// In modo del tutto speculare facciamo le stesse cose di prima con AH operando
// destinatario.

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
JNE indirizzo
JNO indirizzo
JNS indirizzo
JNZ indirizzo
JS indirizzo
JO indirizzo
JZ indirizzo

// Tutte le istruzioni di salto consistono nell'impostare un nuovo valore dell'Instruction pointer. La fase di fetch mi ha portando l'indirizzo nel DEST_ADDR.

Attraverso l'OPCODE capisco se la condizione di JUMP è vera (ovviamente se ho la JMP la condizione sarà sempre vera), quindi se devo fare il salto.

jmp: begin
IP<=(jmp_condition(OPCODE,F)==1)?DEST_ADDR:IP;
STAR<=fetch0;
end

----- istruzione PUSH AL -----
// La push è una scrittura in memoria all'indirizzo del nuovo top della pila. Se SP punta al top della pila allora decremento per andare alla posizione successiva.

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

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

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

----- istruzione POP AL -----
// Letture in memoria all'indirizzo puntato da SP. Ricordiamoci che dobbiamo incrementare in modo da rendere consistenti le letture successive: questo incremento dipende dal numero di byte letti.

popAL: begin
A23_A0<=SP;
SP<=SP+1;

```

Il decremento dipende dalla dimensione degli elementi di una pila (quanti byte?)

```

MJR<=popAL1;
STAR<=readB;
end
popAL1: begin AL<=APP0; STAR<=fetch0; end

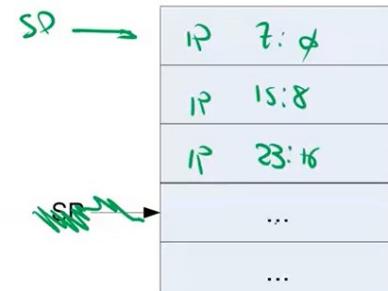
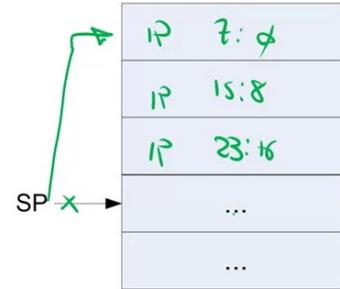
//----- istruzione POP AH -----
// Ibidem
popAH: begin
A23_A0<=SP;
SP<=SP+1;
MJR<=popAH1;
STAR<=readB;
end
popAH1: begin AH<=APP0; STAR<=fetch0; end

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

//----- istruzione CALL indirizzo -----
// La CALL è una JMP preceduta da una scrittura in memoria, se volete.
Nella pila devo salvare 3 byte di IP. Quindi scrivo 3 byte in memoria
all'indirizzo SP-3. Assegno ad IP il DESTINATION ADDRESS.
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 -----
// Si ha una lettura in memoria della pila e la sostituzione di IP con
quanto letto.
ret: begin
A23_A0<=SP;
SP<=SP+3;
MJR<=ret1;
STAR<=readM;
end
ret1: begin IP<={APP2,APP1,APP0}; STAR<=fetch0;
end

```

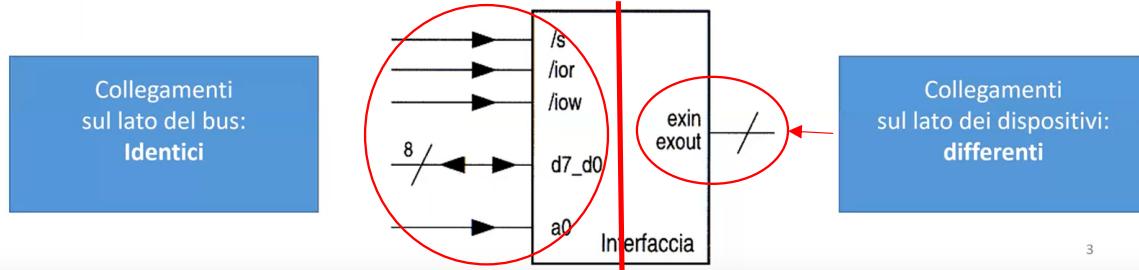


- Conclusioni:

- In ogni stato della descrizione appena visto sono presenti al più micro-salti a due vie.
- Se vogliamo essere precisi la maggior parte dei micro-salti sono a una solo via (micro-salti incondizionati). I micro-salti a due sono presenti quasi esclusivamente nelle sottoliste di lettura/scrittura in memoria.
- Il processore è sintetizzabile con il metodo di scomposizione *Parte operativa – Parte controllo* visto a lezione. Le reti combinatorie contengono solo cose già viste a lezione. Non sono difficili da sintetizzare, il problema è solo la noia.
- **Riflessione:** in soli due mesi di corso abbiamo acquisito la capacità di descrivere e sintetizzare dell'hardware capace di eseguire programmi software arbitrariamente complessi.

Interfacce

- A questo punto ci mancano le varie interfacce che completano il calcolatore.
- Vedremo interfacce di tre tipi:
 - o interfacce parallele, quelle che colloquiano con dispositivi ai quali si invia un byte alla volta;
 - o interfacce seriali, quelle che colloquiano con dispositivi con i quali si scambia un bit alla volta;
 - o quelle per la conversione A-D e D-A, che trasformano gruppi di bit in tensioni e viceversa.
 Utilizzeremo la tensione come grandezza analogica.
- Le interfacce sono dispositivi collocati tra il bus e i dispositivi.

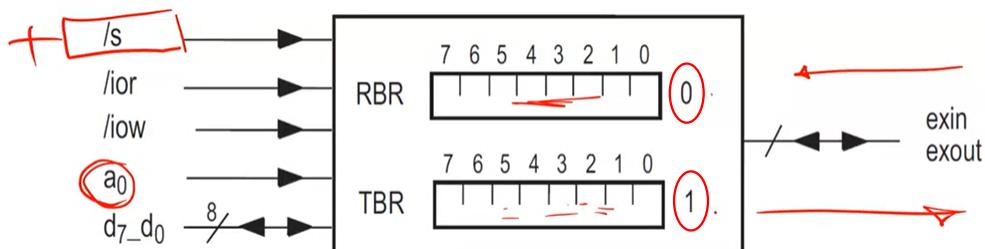


3

- o Per capire di quale interfaccia si parla dobbiamo guardare il lato dei dispositivi, diverso da interfaccia a interfaccia.
- o Dal lato del processore le interfacce sono tutte uguali e si presentano come piccole memorie.
- o Chiaramente se ho un solo filo di indirizzo avrò solo due porte: una di ingresso e una di uscita.
- o I fili di indirizzo rimanenti passano in una maschera che genera il /s.

Descrizione dell'interfaccia a livello funzionale:

- Chi usa l'interfaccia (un sistemista per il montaggio, o un programmatore) come la deve usare?



- o Nell'interfaccia sono presenti due registri:
 - RBR (Receive Buffer Register), dove salvo i dati scritti dal dispositivo esterno;
 - TBR (Transmit Buffer Register), dove salvo i dati da mandare al dispositivo esterno.
- o L'indirizzo interno a_0 mi permette di indicare quale registro mi interessa (0 per RBR e 1 per TBR).
- o La lettura del registro RBR consiste nell'istruzione Assembler IN
IN offset_RBR, %AL
- o La scrittura nel registro TBR consiste nell'istruzione Assembler OUT
OUT %AL, offset_TBR
- o **Problema:** sincronizzazione tra processore e dispositivo. Il processore non ha modo di sincronizzarsi con i dispositivi (anche perché li vede solo come memorie)
 - Supponiamo che un programma contenga le seguenti istruzioni
IN offset_RBR, %AL
...
IN offset_RBR, %AL
Nessuno può garantirmi che tra le due IN il dispositivo sia stato in grado di produrre un dato nuovo. Il secondo dato potrebbe non essere significativo.
 - Dualmente...
OUT %AL, offset_TBR
...
OUT %AL, offset_TBR

Nessuno può garantire che tra le due OUT il dispositivo abbia processato il dato.
Per risolvere questo problema dobbiamo dotarci di registri di stato per implementare un handshake tra processore e dispositivi. I due registri sono i seguenti:

- RSR (*Receive Status Register*),
- TSR (*Transmit Status Register*).

Molto spesso collassati in un unico registro *RTSR*. Di ciascun registro è significativo un solo bit, rispettivamente i seguenti:

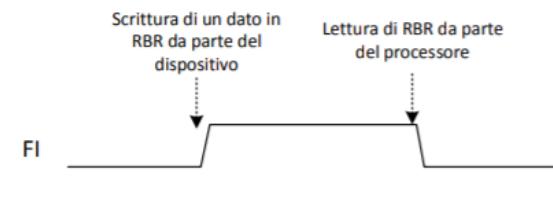
- *Flag di buffer ingresso pieno* (FI)
- *Flag di buffer di uscita vuoto* (FO)

I flag sono gestiti dall'interfaccia che li setta e resetta quando capta certi eventi.

○ **Registro FI:**

- Il flag è inizialmente a 0.
- L'interfaccia lo mette a 1 quando il dispositivo scrive un dato in RBR, segnalando la presenza di un nuovo dato
- Quando il processore accede in lettura al registro RBR, l'interfaccia porta a 0 il flag FI.
- Un sottoprogramma Assembler che legge dati nuovi da un'interfaccia con handshake, mettendoli 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
```



○ **Registro FO:**

- Il flag inizialmente è a 1.
- L'interfaccia lo mette a 0 quando il processore scrive un dato in TBR (istruzione OUT), segnalando che il dispositivo non ha ancora processato.
- Quando il dispositivo (con i suoi tempi) accede al registro TBR e legge il dato, l'interfaccia porta nuovamente a 1 il flag FO.
- Un sottoprogramma Assembler che scrive il contenuto di AL dentro TBR in un'interfaccia con handshake 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
```



Ma tutto questo è inefficiente? Per nulla: l'idea è che il processore rimanga in attesa, cioè che cicli finché il dispositivo esterno non sarà pronto. Il processore perde tempo, considerando la lentezza delle reti che comunicano con lui.

Molto meglio se il processore può andare avanti per conto proprio, con l'interfaccia che segnala al processore quando è pronto. A quel punto il processore interrompe il lavoro che sta facendo.

Direct memory access (DMA): il processore demanda ad un'altra unità (il DMA controller) il compito di trasferire dati tra la memoria e le interfacce, mentre va avanti con le sue elaborazioni. Il processore riferisce al DMA l'area di memoria e l'interfaccia con cui lavorare.

Capitolo 5

Giovedì 03/12/2020

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

letv13
 A23_A0 L = - - - i
 ↓
 10R_L = p

- **Perché non possiamo farlo?**

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

Interfacce parallele

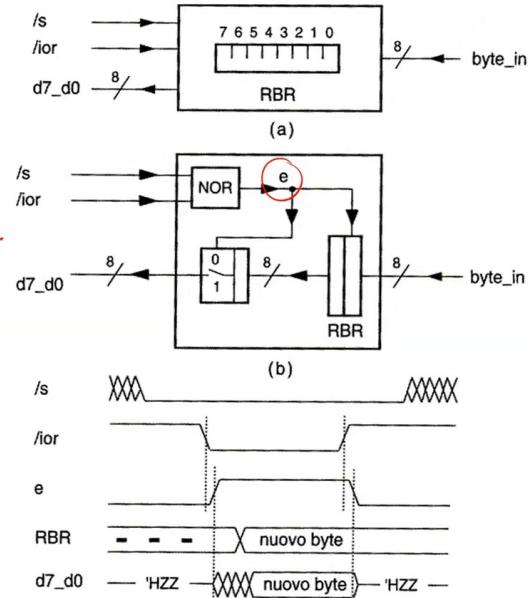
Interfaccia parallela di ingresso senza handshake

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

- **Come è fatta l'interfaccia internamente?**

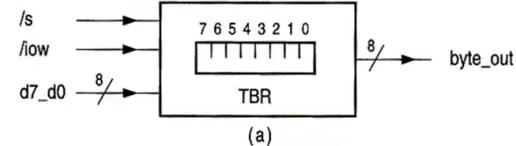
- o Abbiamo un registro RBR che salva i dati in ingresso.
- o Il clock deve essere fornito sulla base degli accessi fatti all'interfaccia. Quando il processore vuole accedere il registro campiona!
- o Il comando di memorizzazione non sarà dato col solito generatore di impulsi: esprimiamo un comando di memorizzazione quando il processore vuole accedere a questa interfaccia.
- o La cosa può essere ottenuta attraverso una porta NOR che ha in ingresso /s ed /ior: restituisco 1 soltanto se entrambe sono abbassate a zero. Se esprimo 1 le porta tristate viene abbassate e il dato in ingresso campionato.
- o **Temporizzazione:**
 - Si stabilizzano gli indirizzi
 - Abbasso /ior a zero: con un leggero ritardo dovuto alla porta NOR le tri-state vanno in conduzione e il registro RBR campiona il nuovo byte.
 - Dopo un tempo di propagazione RBR presenterà il nuovo byte.
 - Non appena /ior ritorna ad 1 le tristate vanno in alta impedenza.
- o **Descrizione Verilog** (un po' sintesi-oriented, cit.):


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



Interfaccia parallela di uscita senza handshake

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



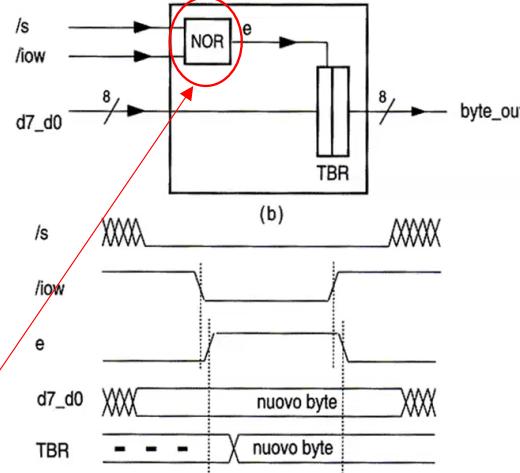
- All'interno dell'interfaccia ho una porta NOR che esprime il clock per il registro TBR.
- Non sono necessarie porte tri-state, ovviamente.
- Ricordiamo che il comando di memorizzazione è il fronte di discesa e non quello di salita.
- I dati devono essere pronti prima del fronte di discesa.
- All'arrivo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

- **Perché non posso fare un'interfaccia che memorizza sul fronte di salita?**

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

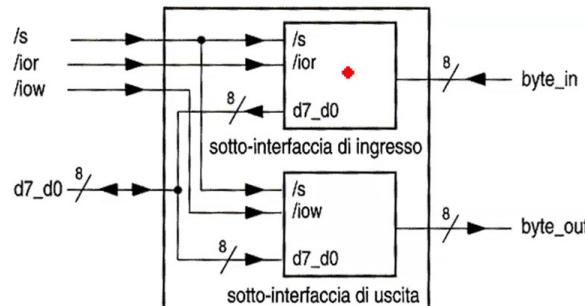
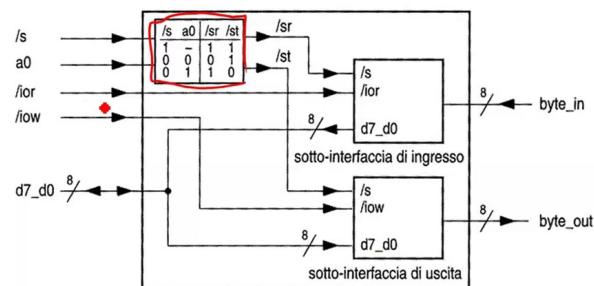
- **Descrizione in Verilog:**

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



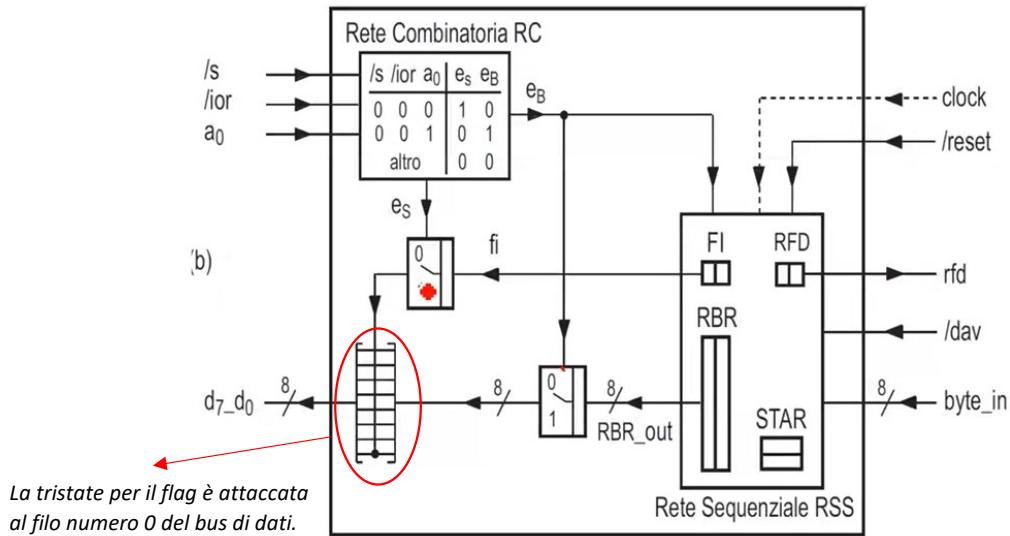
Montaggio di interfacce parallele di ingresso e uscita

- Vogliamo realizzare un'interfaccia sia di ingresso che di uscita.
- **Primo metodo:**
 - o Poniamo la porta di ingresso all'indirizzo pari (0) e quella di uscita all'indirizzo dispari (1). Serve un minimo di logica combinatoria per produrre due select. I meccanismi utilizzati sono i soliti.
- **Secondo metodo:**
 - o Le due porte sono allo stesso offset.
 - o Il programmatore le riferisce con lo stesso indirizzo nello spazio di I/O, e l'accesso dipende dal tipo di accesso: è immediato dalle regole di pilotaggio che solo una delle due sotto-interfacce sarà abilitata a fare qualcosa.



Interfaccia parallela di ingresso con handshake

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



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

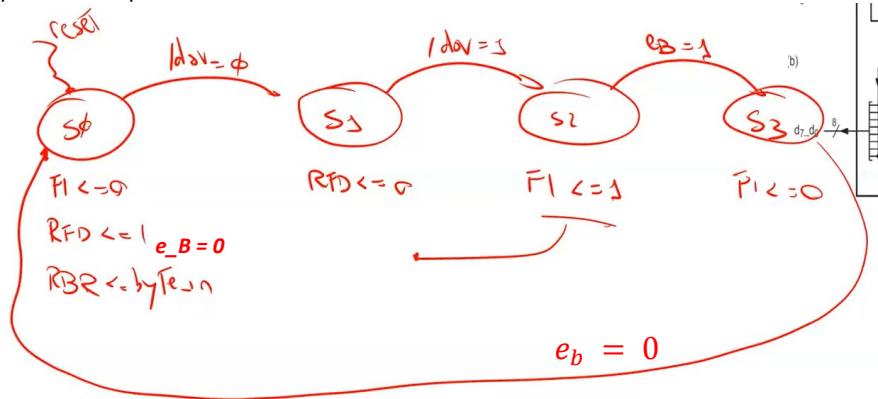
Per quanto riguarda la disposizione delle porte tristate:

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

Sintetizziamo la RSS:

- Di quali registri ho bisogno? Quelli già detti, oltre al registro STAR per tenere conto della sequenza di stati.
- **Condizioni di reset:**
 - I valori per gestire l'handshake sono quelli già noti (rfd e /dav uguali ad 1)
 - Il flag di ingresso pieno va a 0 perché nessuno, al momento del reset, ha scritto qualcosa.
 - Il contenuto di RBR non è significativo e quindi non necessita di essere inizializzato.
 - Il registro STAR avrà come valore iniziale quello relativo al primo stato interno.
- **Descrizione delle operazioni negli stati** (immagine a pagina dopo):
 - **S0:** stato di riposo in cui attendiamo che il produttore (il dispositivo) fornisca un dato (cioè attendo che /dav venga abbassato). Ogni volta aggiorno il valore del registro RBR, in modo tale da averlo già pronto nello stato S1.
 - **S1:** Metto rfd a 0 per indicare che il consumatore (cioè l'interfaccia) ha ricevuto i dati e li sta elaborando. Attendo che /dav venga nuovamente alzato (gestione dell'handshake, necessario).
 - **S2:** metto il flag di ingresso pieno uguale ad 1. Rimango in questo stato finchè il processore non svolgerà un'operazione di lettura nel buffer RBR (cioè finchè non avrò $e_b = 1$).

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



- **Descrizione in Verilog:**

```

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

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

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

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

```

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

```

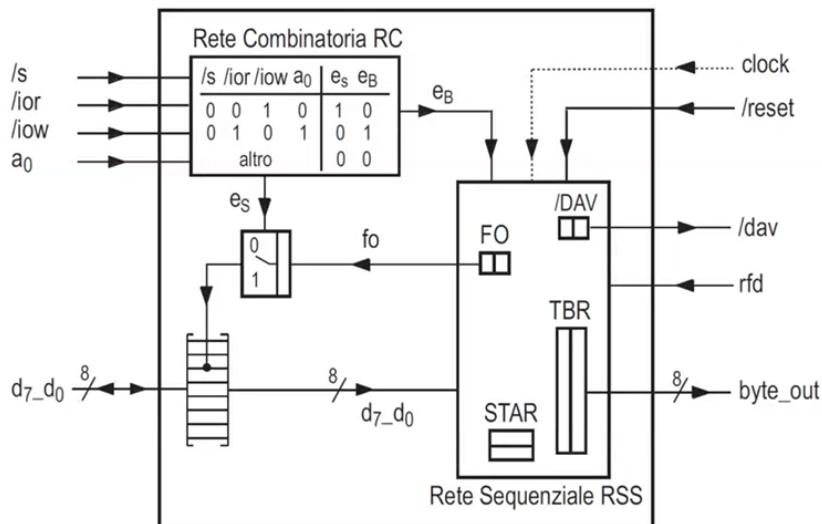
    //Ritorno allo stato interno iniziale quando il processore
    //termina la fase di esecuzione dell'istruzione IN RBR_offset,AL
    S3: begin STAR<=(eB==1)?S3:S0; end
endcase
endmodule

```

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

Interfaccia parallela di uscita con handshake

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



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

Per quanto riguarda la disposizione delle porte tristate:

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

Descrizione:

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

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

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

```

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

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

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

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

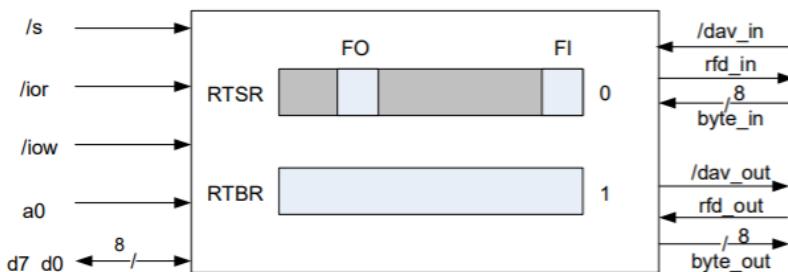
```

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

Interfaccia parallela di ingresso-uscita

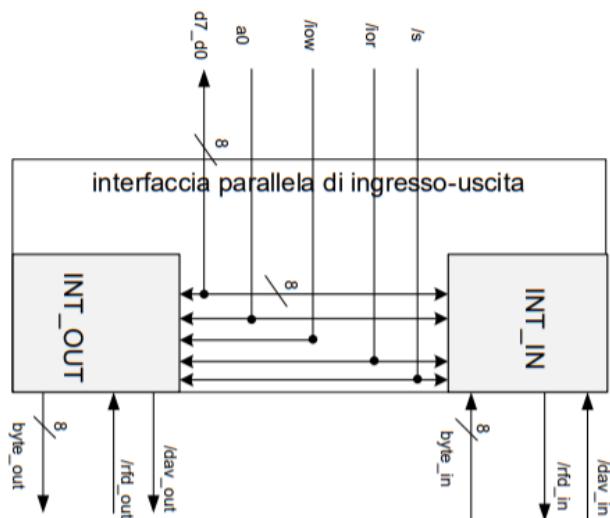
Le due interfacce con handshake appena introdotte possono essere connesse in un'unica interfaccia parallela con handshake d'ingresso-uscita. Si consideri che:

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



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

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

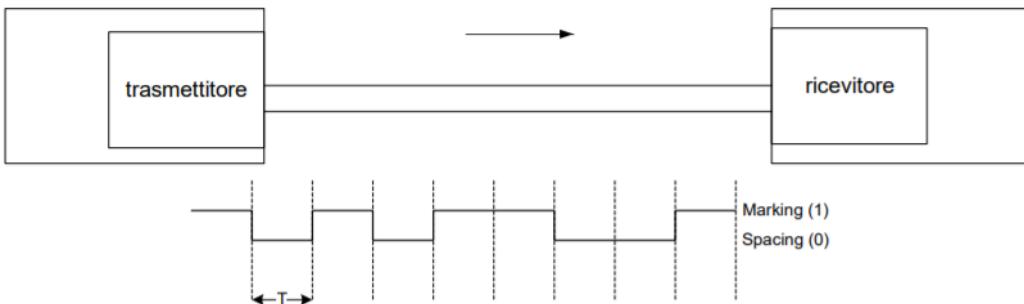


Interfacce seriali

- Un'interfaccia seriale è un'interfaccia dove la trasmissione dei singoli bit avviene in modo seriale. Con "in modo seriale" si intende la trasmissione di un bit alla volta, dal meno significativo, al più significativo.
L'interfaccia:
 - o riceve byte dal bus e trasmette all'esterno sequenze di bit (serializzazione);
 - o riceve dall'esterno sequenze di bit e presenta byte al processore componendo quelle sequenze di bit in un registro (de-serializzazione).
- Le interfacce viste fino ad ora sono, teoricamente, seriali, in quanto permettono di trasmettere una serie di byte. La differenza sta nel fatto che qua serializziamo il singolo byte.
- Un PC, di norma (oggi sempre meno), presenta almeno una porta seriale. La cosa è utilizzata per modem e un tempo pure per i mouse (pensiamo ai mouse di una volta, quelli con la pallina, che presentano un connettore non-USB).
- Perché esistono le interfacce seriali: un tempo i calcolatori consistevano in grossi elaboratori (cervello) che venivano connessi a terminali stupidi (mani). La connessione tra terminale stupido ed elaboratore intelligente avveniva, appunto, mediante linee seriali.

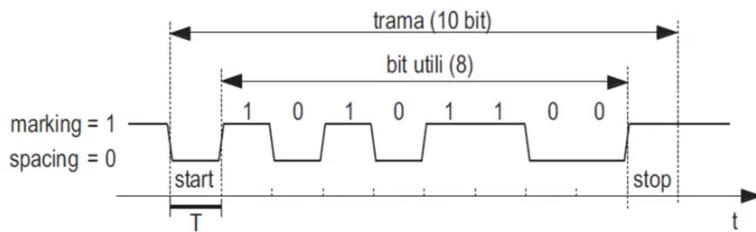
Comunicazione seriale

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



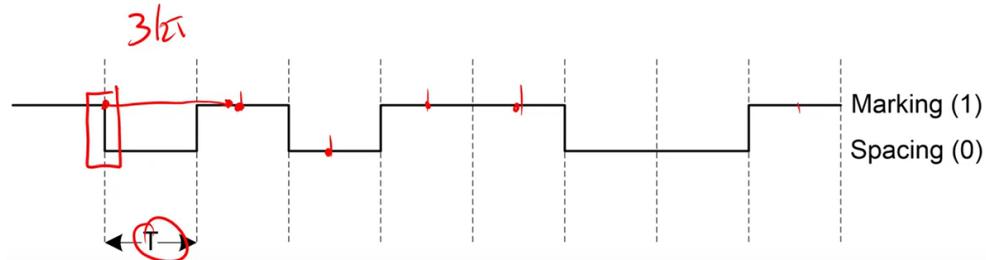
- Il mezzo trasmissivo consiste in un insieme di due linee:
 1. Una linea di riferimento, detta linea di massa
 2. Una linea che porta una tensione riferita a massa. Su questa linea sono leciti due valori:
 - *marking*, cioè 1 logico;
 - *spacing*, cioè 0 logico.
- La trasmissione di un bit consiste nel tenere la seconda linea in uno stato di *marking* o *spacing* per un determinato tempo T, detto **tempo di bit**.
- Un insieme di bit scambiati costituisce una trama, detta in inglese frame.
- **Problema fondamentale**: abbiamo detto che la comunicazione si ha mediante due fili, allora come possiamo sincronizzare trasmettitore e ricevitore? Chiaramente non potremo condividere un clock (significa avere un filo in più), né aggiungere delle linee dedicate all'*handshake* (due fili in più).
- **Soluzione al problema fondamentale**:
 - o Le due parti devono essere concordi sul tempo di bit e sul numero di bit che caratterizza ogni trama (tipicamente si ha da 5 a 8 bit per trama);
 - o Le trame devono essere rese riconoscibili, cioè abbiamo bisogno di elementi che permettono di capire quando una trama inizia e quando finisce. Abbiamo stabilito che:
 - La linea sta, normalmente, in uno stato di marking;
 - Per iniziare una nuova trama poniamo la linea in uno stato di spacing. Questo significa che ogni trama presenterà un bit iniziale, non informativo del contenuto della trama, detto bit di start.

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



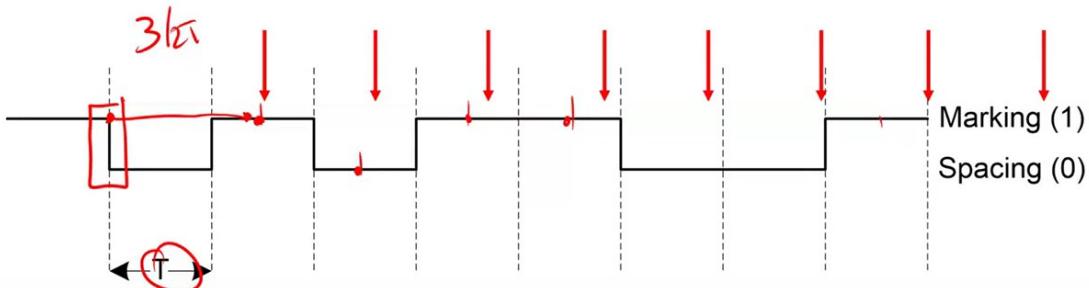
Esempio di trama. Il trasmettitore fa il primo passo ponendo il bit di start. Sia il trasmettitore che il ricevitore conoscono il tempo di bit.

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



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

- Cosa avviene in realtà?



- Supponiamo che il clock del ricevitore sia un po' più lento: a causa di fenomeni fisici e della discrepanza dei clock si accumulano ritardi e nel giro di poco esco dal bit, quindi campiono in modo sbagliato.
- Quale compromesso posso adottare? Se il clock del ricevitore misura un tempo $T + \Delta T$ (somma tempo di bit e ritardo), per poter decodificare n bit è necessario che

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

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

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

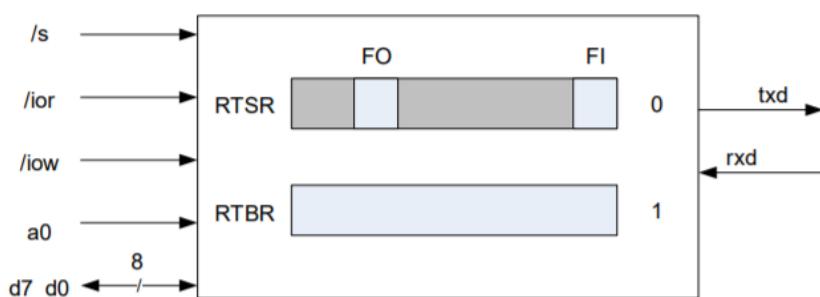
cioè l'errore relativo tollerabile è inversamente proporzionale al numero di bit che devono essere trasmessi tra due segnali di sincronizzazione.

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

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

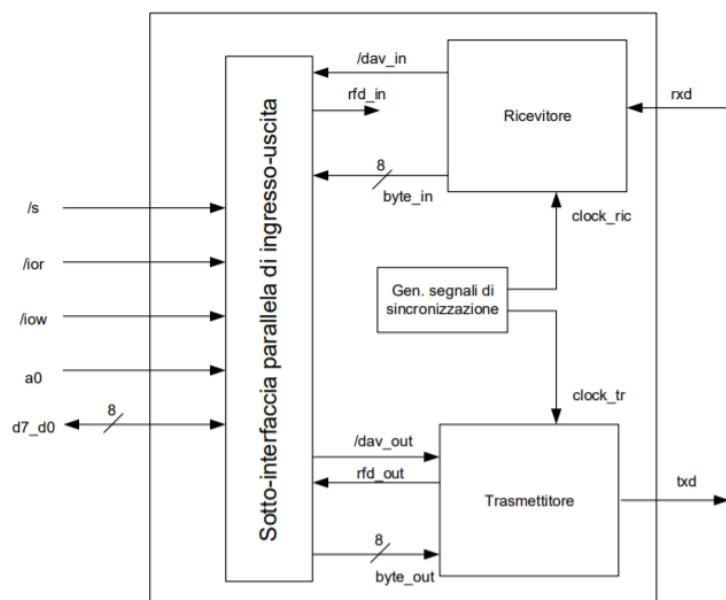
Visione funzionale e struttura interna dell'interfaccia

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



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

- Spogliamo ulteriormente l'interfaccia

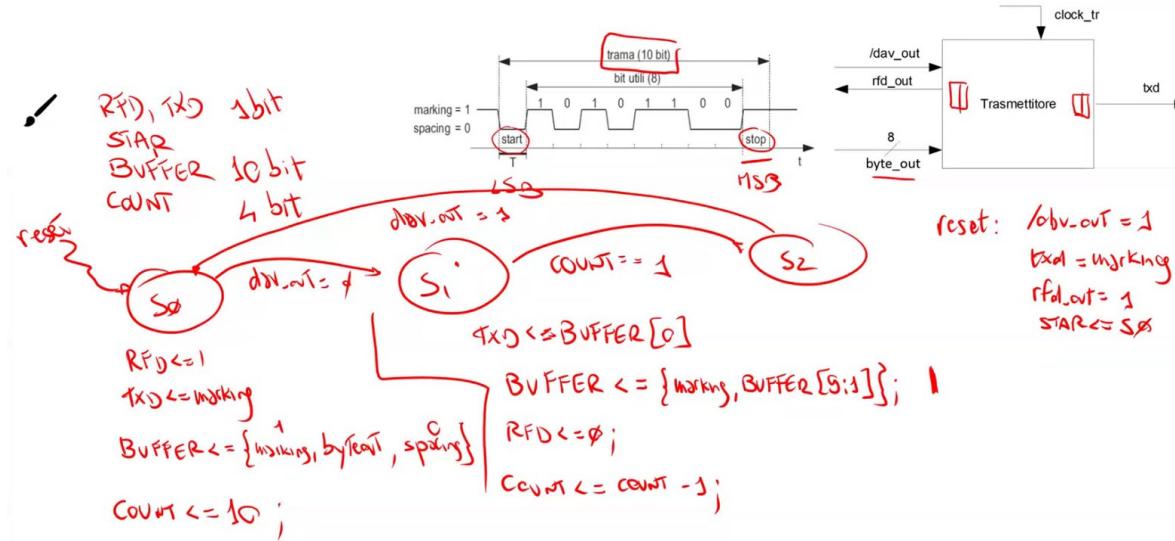


Individuiamo una RSS Ricevitore e una RSS Trasmettitore. Si osservi che:

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

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

Descrizione del trasmettitore



- L'interfaccia:
 - o Accetta un nuovo byte dalla sotto-interfaccia parallela di uscita, con la quale ha un *handshake*;
 - o Trasmette tutti i bit di quel byte sul mezzo trasmissivo tramite il filo *txd*.
- Di quali registri abbiamo bisogno?
 - o Registro di stato STAR
 - o Registri a supporto delle uscite: RFD, TXD (ciascuno di 1bit)
 - o Registro BUFFER dove memorizzare l'intera sequenza di bit (10 bit, incluso il bit di start e il bit di fine)
 - o Registro COUNT, quattro bit (10) necessari per contare il numero di bit da trasmettere.
- Ipotesi al reset:
 - o /dav_out e rfd_out presentano i valori tipici dell'handshake al momento del reset (entrambi 1)
 - o txd è uguale ad uno, quindi è in uno stato di marking
 - o STAR è uguale ad S0, come al solito si pone come valore iniziale quello del primo valore interno.
- Ipotesi sul clock: un clock equivale al tempo di bit T.

```

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

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

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

  always @ (reset_==0) #1 begin RFD<=1; TXD<=mark; STAR<=S0; end
  always @ (posedge clock) if (reset_==1) #3
    casex (STAR)
      S0: begin RFD<=1; COUNT<=10; TXD<=mark;
            BUFFER<={stop_bit,byte_out,start_bit};
      end
      S1: begin
            
```

```

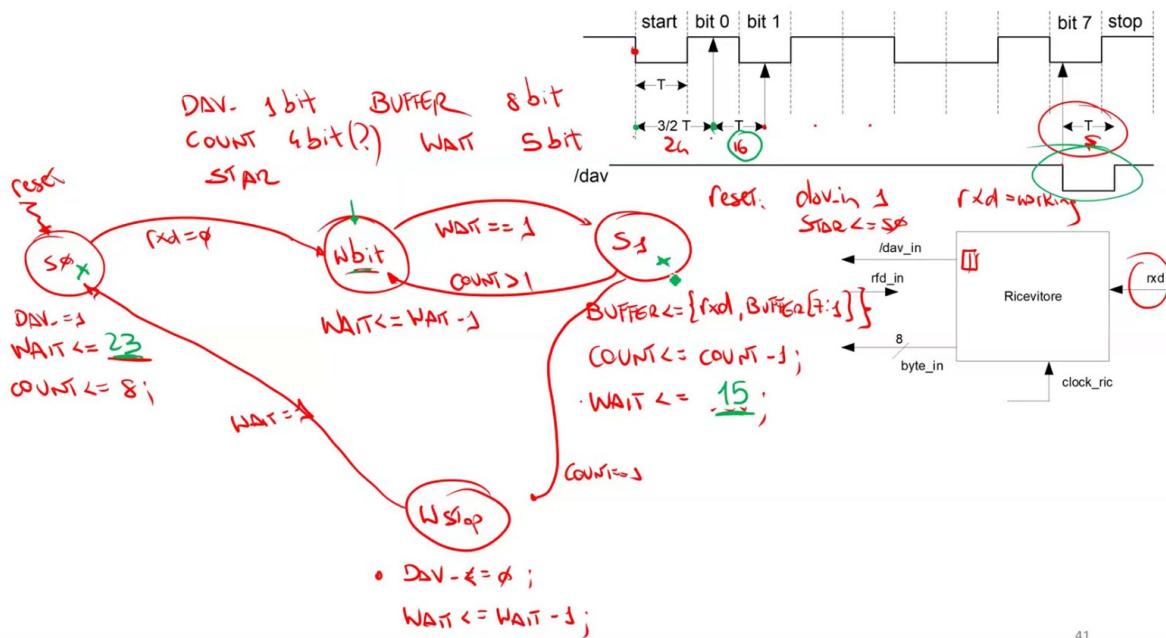
        STAR<=(dav_out==1)?S0:S1; end
S1: begin RFD<=0; TXD<=BUFFER[0]; BUFFER<={mark, BUFFER[9:1]};
      COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
S2: begin STAR<=(dav_out==0)?S2:S0; end
endcase
endmodule

```

- Spiegazione degli stati:

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

Descrizione del ricevitore



- Il ricevitore:

- Non è collegato al filo rfd_in. Come già anticipato il ricevitore non è in grado di controllare il flusso dei dati in ingresso: segue che non ha nessun senso gestire un handshake completo. Se la sottointerfaccia parallela non è in grado di accettare un nuovo dato è problema suo: il dato verrà sovrascritto da una nuova trama di bit.
- Il ricevitore capta l'inizio di una trama col fronte di discesa da marking a spacing. Campiona il primo bit dopo $3/2$ del tempo di bit da quando vede l'inizio della trama. Successivamente campiona ciascun successivo bit dopo un tempo di bit.

- Ipotesi sul clock:

- Il clock del ricevitore deve essere diverso da quello del trasmettitore: questo perché dobbiamo riuscire a campionare i bit (per quanto possibile) a metà del tempo di bit. Non è possibile fare ciò con lo stesso clock del trasmettitore.
- **Come imposto il clock?** Il periodo di clock è la minima finestra temporale su cui guardo gli eventi, il ricevitore deve individuare un certo evento nella maniera più veloce possibile: segue che più il clock è veloce, più saremo precisi. Supponiamo che il clock abbia frequenza **16x**, cioè che sia sedici volte più veloce del clock del trasmettitore.

- **Di quali registri abbiamo bisogno?**
 - o Registro DAV_a 1 bit, che supporta l'uscita /dav_in
 - o Il registro BUFFER dove la parte di trama ricevuta fino a questo momento. Mi bastano solo 8 bit.
 - o Il registro COUNT, con cui tengo conto del numero di bit letti.
 - o Il registro WAIT, per contare i clock. Utilizzo questo registro per attendere:
 - 16 cicli di clock tra un bit utile e il successivo, e alla fine (quando finisco nel bit di stop). In questo ultimo caso potrei attendere solo 8 clock, ma considerando le discrepanze fisiche nei due clock non mi conviene.
 - 24 cicli di clock dopo il fronte di discesa del bit di start
- **Ipotesi al reset:**
 - o /dav_in è a 1 (come nell'handshake)
 - o byte_in non è significativo poiché è protetto dall'handshake
 - o rxd è in marking
- **Spiegazione degli stati:**
 - o **S0**: in questo stato ho /dav a 1. Passo allo stato successivo quando rxd va in spacing, e rimango in quello stato per un numero di clock pari a 23 (cioè 3/2 del tempo di bit). Pongo WAIT uguale a 23 e non a 24 poiché perdo già un clock in S0 prima di passare allo stato Wbit.
 - o **Wbit**: se mi trovo in questo stato significa che devo attendere prima di fare il prossimo campionamento. Abbiamo definito, in S0 e in S1, un valore di WAIT. WAIT viene decrementato e mi sposto (o ritorno) in S1 quando ho WAIT uguale ad 1.
 - o **S1**: imposto il valore di BUFFER. Devo fare in modo che i primi valori inseriti si trovino sui posti meno significativi: faccio ciò inserendo ogni nuovo bit (preso dal filo rxd) come MSD (shifts sbarazzandomi della LSD). Decremento COUNT, ed ogni volta ritorno in Wbit (tranne quando avrò COUNT == 1). Memorizzo in WAIT 15, e non 16, per la stessa motivazione vista in S0. Quando COUNT sarà uguale ad 1 mi sposto in Wstop.
 - o **Wstop**: porto /dav a 0, abbiamo finito. Prima di ritornare in S0 devo attendere 16 clock (il solito tempo di bit). A tale scopo recupero il valore del registro WAIT, impostato in S1. Decremento WAIT, e ritorno in S0 quando avrò WAIT == 1.

```

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

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

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

```

Capitolo 6

Martedì 09/12/2020

6.1 Conversione analogico/digitale e digitale/analogica

Finora ci siamo limitati a osservare interfacce che consentono a due calcolatori di dialogare tra loro. Non dobbiamo farci ingannare dall'apparenza che i bit ci siano per magia: nel mondo esterno al calcolatore l'informazione è associata a grandezze analogiche, cioè grandezze che cambiano continuamente nel tempo (in contrapposizione alle stringhe di bit presenti nei calcolatori, che variano in modo discreto). Abbiamo la necessità di svolgere le seguenti conversioni:

- **Conversione A/D:** conversione di grandezze analogiche in stringhe di bit (comunicazione del mondo esterno col calcolatore)
- **Conversione D/A:** conversione di stringhe di bit in grandezze analogiche (comunicazione del calcolatore col mondo esterno).

La grandezza analogica da noi considerata è la tensione.

Obiettivo

Convertire una tensione v in un numero x (naturale o intero), in base 2 e rappresentato su N bit, e viceversa

Dove N è solitamente 8 o 16. La tensione sta su una scala di FSR volts (*Full-scale Range*): tipicamente abbiamo $\text{FSR} \in [5; 30]$.

Interpretazione del numero o della tensione Distinguiamo due tipi di conversioni:

- **Conversione unipolare:** $v \in [0; \text{FSR}]$ (tensioni positive), $x \in [0; 2^N - 1]$ (numeri naturali)
- **Conversione bipolare:** $v \in [-\frac{\text{FSR}}{2}; +\frac{\text{FSR}}{2}]$ (tensioni negative e positive),
 $x \in [-2^{N-1}; 2^{N-1} - 1]$ (numeri naturali)

Mondo ideale Definiamo il rapporto tra le due scale (tensione e numero)

$$K = \frac{\text{FSR}}{2^N}$$

in una conversione ideale dovrei ottenere

$$v = K \cdot x$$

Mondo non ideale Nella realtà questa cosa non è possibile poichè abbiamo un errore di conversione

$$|v - k \cdot x| \leq \text{err}$$

Questo errore è dovuto ai seguenti fattori:

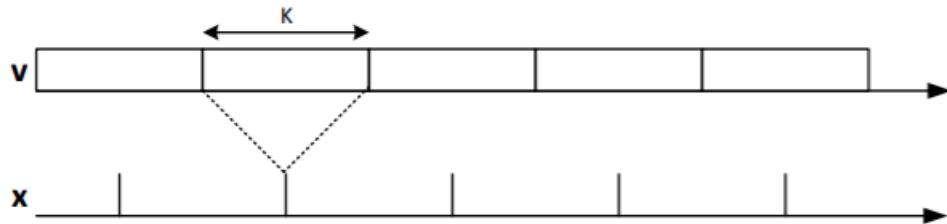
1. **Imprecisioni circuitali**, cioè componenti di un circuito che non si comportano in maniera ideale. Questo errore, difficilmente eliminabile, è presente sia nella conversione D/A che in quella A/D. Si parla di **errore di non linearità**.
2. **Arrotondamento (o Errore di quantizzazione)**. Si tenga conto nella formula $v = K \cdot x$ che $v \in \mathbb{R}$ e $x \in \mathbb{Z}$: questo significa che nella conversione A/D, cioè nel passaggio da tensione a numero, si avrà perdita di informazione (conversione da numero reale a numero intero).

Riflettiamo sull'errore di linearità L'errore di linearità deve essere più piccolo di $\frac{K}{2}$. La presenza dell'errore ci porta a dire che la tensione appartiene a questo intervallo

$$v \in [K \cdot x - \text{err}; K \cdot x + \text{err}]$$

avere un errore che non rispetta la condizione detta significa avere intervalli parzialmente sovrapposti, quindi convertire numeri più grandi in tensioni più piccole e viceversa.

Riflettiamo sull'errore di quantizzazione Supponiamo di dividere la scala FSR in 2^N intervalli uguali a K , e di convertire tutto l'intervallo nello stesso numero.



La conversione da v a x sarà esatta per la tensione al centro dell'intervallo, ed errata di $\pm \frac{K}{2}$ per le tensioni agli estremi.

Conclusioni

- Nella conversione D/A dobbiamo avere $\text{err} < \frac{K}{2}$ (errore di non linearità)
- Nella conversione A/D dobbiamo avere $\text{err} < \frac{K}{2} + \frac{K}{2} = K$ (errore di non linearità ed errore di quantizzazione)

Esempi di conversioni Presenti degli esempi di conversione a pagina 53 della dispensa sulla struttura del calcolatore.

Tempi di risposta delle conversioni

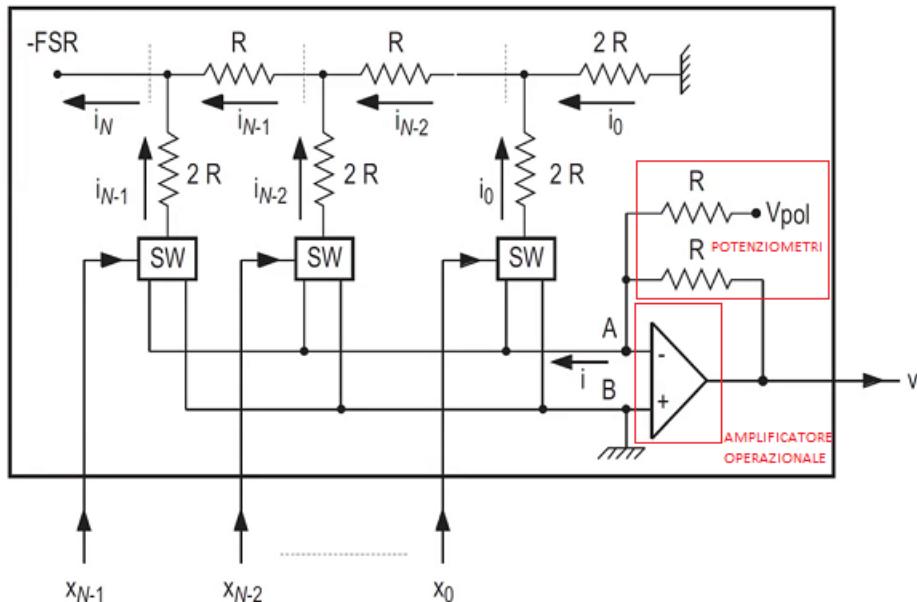
- I convertitori D/A sono circuiti estremamente semplici. Sono velocissimi (pochi ns).
- I convertitori A/D hanno tempi di risposta variabili: sono circuiti sequenziali dove posso incontrare architetture diverse. I convertitori da noi considerati hanno tempi di risposta di qualche centinaio di ns.

Osservazione sulla conversione bipolare I convertitori bipolarari rappresentano i numeri interi con rappresentazione in traslazione. Il numero intero x è rappresentato dal naturale

$$X = x + 2^{N-1}$$

la tensione negativa di fondo scala, che corrisponde al numero intero negativo più piccolo, verrà convertita nel naturale 0. Ricordarsi che la conversione da traslazione a complemento a 2, e viceversa, si ottiene complementando il MSB.

6.1.1 Convertitore D/A



La tensione v in uscita dovrà essere direttamente proporzionale al numero posto in ingresso (già da questo si capisce che la rappresentazione avviene per traslazione). Facciamo l'assunzione che tutte le linee verticali siano connesse a massa: provengono tutte da switch, che presentano in ingresso la linea A e la linea B. La linea B è esplicitamente connessa a massa, la linea A lo verificheremo prossimamente.

- Ogni ramo verticale presenta la stessa corrente che scorre nel ramo orizzontale alla sua destra.
- La resistenza vista a destra di ogni tratteggio è pari ad R .
- Ottengo due resistenze in serie che sommate mi danno $R + R = 2R$.

- La linea verticale immediatamente vicina al ramo orizzontale considerato presenta sempre come resistenza $2R$, quindi otteniamo in ogni filo verticale lo stesso risultato di prima.
- Per quanto riguarda la corrente abbiamo, nel tratteggio vicino alla linea verticale più a destra

$$i_p = i_0 + i_0 = 2 \cdot i_k$$

- Sommando le varie correnti otteniamo, alla fine,

$$i_N = 2^N \cdot i_0$$

Osservando che $i_N = \frac{FSR}{R}$ troviamo

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

quindi

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

dobbiamo ancora verificare che i rami verticali siano collegati a massa.

Vediamo gli switch Gli switch sono guidati dai bit della rappresentazione del numero da convertire. Abbiamo tanti switch quante le cifre del numero.

- Se $x_j = 0$ l'interruttore è connessa alla linea B, quindi a massa.
- Se $x_j = 1$ l'interruttore è connesso alla linea A, che non sappiamo quanto vale.

Amplificatore operazionale Componente connessa all'alimentazione, la corrente non proviene dagli ingressi.

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

con $\alpha \gg 1$, la tensione dipende dalle differenze di tensione agli ingressi dell'amplificatore. tenendo conto che il filo B è connesso a massa possiamo dire

$$V^{\text{out}} = -\alpha \cdot V^-$$

Consideriamo anche l'altro ramo, quello con resistenza R e corrente i_a

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

unisco il tutto e ottengo

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

quindi

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

questo mi permette di dire $V^- \approx 0$. ■

Quanto vale la corrente che esce da A e va verso sinistra? Tenendo conto di quanto detto sullo switch affermiamo che

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

ci siamo ricordati che $i_1 = 2 \cdot i_0$, e così via fino ad arrivare a $i_{N-1} = 2^{N-1} \cdot i_0$. La sommatoria ottenuta è la rappresentazione posizionale in base 2 di un numero naturale X . Ricordandomi che $i_0 = \frac{K}{R}$ ottengo

$$i = \frac{K}{R} \cdot \sum_{i=0}^{N-1} 2^i \cdot x_i = \frac{K}{R} \cdot X$$

segue che gli switch alterano la corrente che scorre da A verso sinistra.

Come faccio uscire la tensione giusta dal convertitore? Scriviamo le equazioni di bilancio della corrente al nodo A

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

Mi sbarazzo di R trovando

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

quindi (raccolgo rispetto a K)

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

otteniamo che V è proporzionale rispetto ad X . Segue che con

- $V_{pol} = 0$ avrò una conversione unipolare $V = K \cdot X$, mentre con
- $V_{pol} = \frac{FSR}{2}$ avrò una conversione bipolare

$$V = K \cdot (X - 2^{N-1})$$

cioè il numero rappresentato in traslazione in base 2.

Il circuito è in sostanza combinatorio.

Problema 1: Situazione precedente nella realtà Quanto spiegato poco fa è valido solo in un modello ideale. Nella realtà dobbiamo tener conto di errori. Riprendiamo l'equazione di bilancio e teniamo conto della cosa

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

otteniamo

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

osserviamo il rapporto linea tra tensione e ingresso X (sappiamo che esiste una proporzionalità diretta tra tensione in uscita e stringa di bit X posta in ingresso)

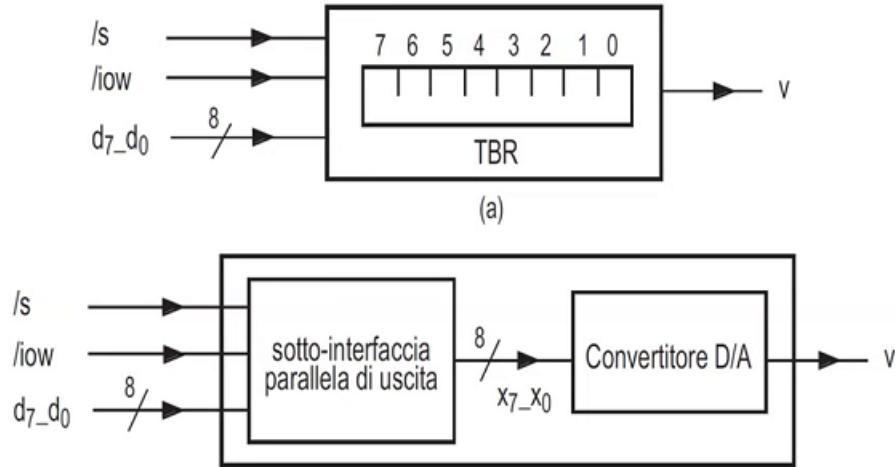
- Se altero γ_2 altero la pendenza della retta.
- Se altero γ_1 traslo la retta.

A tutto questo segue la necessità di dover tarare il convertitore prima di svolgere questa cosa. Le due resistenze vicine all'amplificatore operazionale sono dette *potenziometri*, cioè resistenze variabili che possono essere tarate.

Problema 2: Variazioni a frequenza elevata Supponiamo di voler passare da $X = 01111111$ alla sua complementata $X = 10000000$. Sappiamo fin dall'inizio che la rete non percepisce mai, in modo istantaneo e parallelo, la variazione di tutti i bit. Seguono stati intermedi. Questo comporta variazioni ad alta frequenza, un qualcosa che deve essere evitato. La soluzione è porre un filtro passa-basso, che taglia le variazioni a frequenza troppo elevata.

6.1.1.1 Interfaccia per la conversione D/A

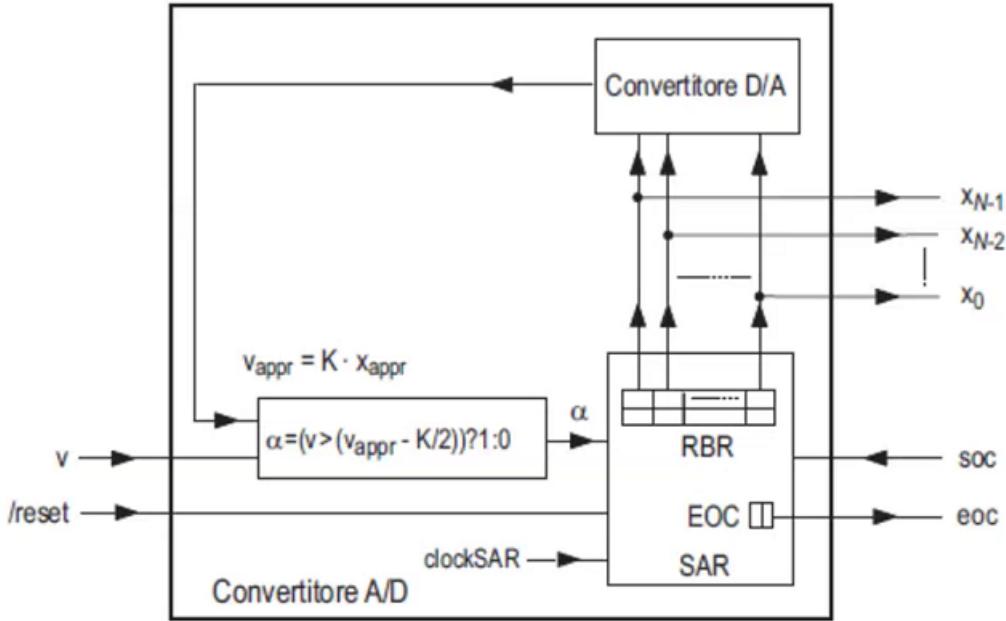
Attenzione Non confondere il convertitore con l'interfaccia!



Abbiamo un'interfaccia parallela senza handshake, con un unico registro TBR (posto in una sotto interfaccia parallela di uscita) e un convertitore D/A.

Velocità Il convertitore D/A è molto più veloce del processore, quindi non servono ulteriori cose nell'interfaccia.

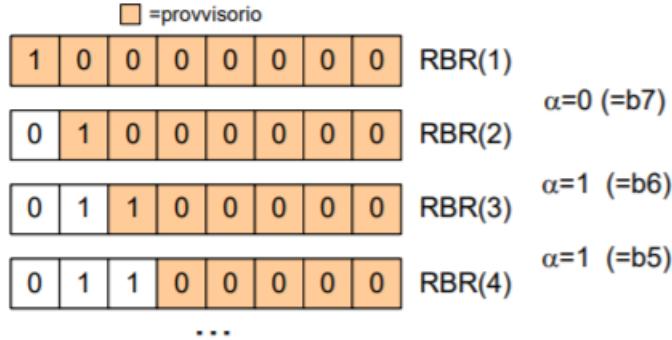
6.1.2 Convertitore A/D



Il convertitore che andiamo a descrivere è detto **convertitore A/D ad approssimazioni successive** ad 8 bit. Il cuore del convertitore è la RSS detta SAR (*Successive Approximation Register*).

- All'interno del convertitore A/D è presente un convertitore D/A (chiaramente dovrò utilizzare un convertitore bipolare o unipolare in base alle circostanze) che utilizzerò per generare una tensione da confrontare, mediante comparatore analogico, con la tensione in ingresso.
- Utilizzeremo un'handshake soc/eoc, considerando la mole di lavoro.
- Il convertitore esegue una **ricerca logaritmica** (detta bisezione, o ricerca binaria).
 - Ho una tensione v analogica in ingresso, mantenuta costante nel corso dell'esecuzione dell'algoritmo mediante latch analogico (non avrebbe senso farla ballare durante i confronti, osservazione per bimbi spastici).
 - Quando il convertitore percepisce $soc = 1$ inizia a “sparare” numeri nel convertitore D/A. Si pone come primo numero quello al centro dell'intervallo di rappresentabilità (100...00). Questo numero è valido sia con conversione unipolare che con conversione bipolare:
 - Ogni tensione ottenuta viene confrontata con la tensione in ingresso (resa costante, ricordiamo) con un comparatore di tensione.
 - Se la tensione generata è più grande della tensione in ingresso allora avremo $\alpha = 1$, quindi la cifra considerata in una certa iterazione uguale ad 1.
 - Continuo a ripetere le operazioni precedenti, quindi a svolgere confronti tra tensioni.
 - Il numero di passi da svolgere è pari al numero di bit: ogni confronto, in base 2, mi permette di individuare un bit. Li determiniamo dalla cifra più significativa a quella meno significativa.

Esempio



Vediamo l'esempio qua sopra. Ogni volta:

- Eseguo il confronto fra tensioni
- Determino se settare o resettare la cifra. La cifra meno significativa immediatamente successiva viene impostata uguale ad 1.

Descrizione Verilog Descriviamo la RSS che permette di gestire l'handshake e il contenuto del buffer che ogni volta sarà “sparato” verso il convertitore D/A.

```
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 consideri anche la seguente versione, semplificata nel codice e nel numero di stati interni grazie all'introduzione di una function:

```

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

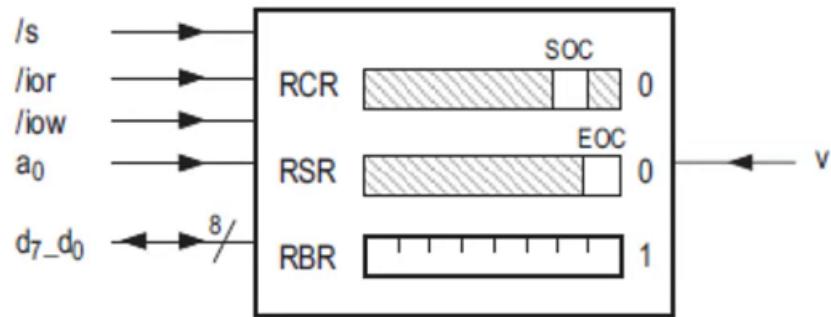
    function [7:0] nuovobyte;
      input [7:0] vecchiobyte;
      input alpha;
      input [2:0] posizione;
      casex (posizione)
        7: nuovobyte={ alpha,'B1000000};
        6: nuovobyte={vecchiobyte[7], alpha,'B100000};
        5: nuovobyte={vecchiobyte[7:6],alpha,'B10000};
        4: nuovobyte={vecchiobyte[7:5],alpha,'B1000};
        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

```

- Ogni volta, sfruttando l'ingresso alpha generato dal comparatore, si determina un bit della sequenza di cifre.
- La sequenza memorizzata inizialmente nel RBR è provvisoria, quando alzeremo eoc il processore chiederà una lettura del buffer trovando la sequenza definitiva.

6.1.2.1 Interfaccia di conversione A/D

Ricordiamo di nuovo L'interfaccia non è il convertitore.

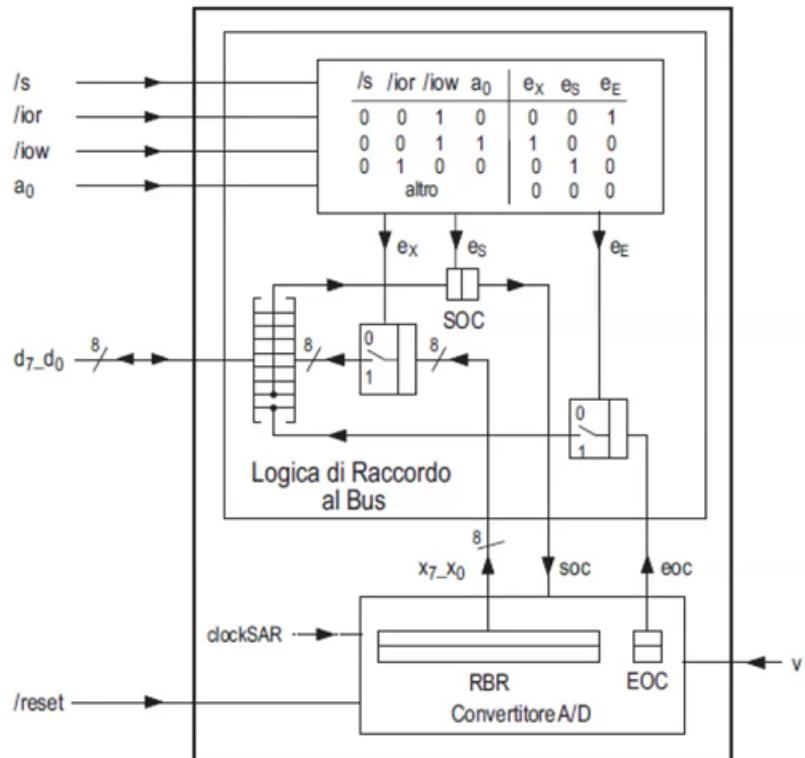


Abbiamo un'interfaccia parallela con handshake soc/eoc. Presenta i seguenti registri:

- Il registro di Buffer dove finisce il risultato (RBR, *Receive Buffer Registry*)
- Un registro di stato per leggere il valore di eoc (RSR, *Receive Status Registry*)
- Un registro di controllo per poter aggiornare il valore di soc (RCR, *Receive Control Registry*).

Gli ultimi due indirizzi sono mappati allo stesso indirizzo logico (si ottiene il registro RSCR, *Receive Status and Control Registry*). Non è un problema perché i bit non si sovrappongono: inoltre la rete sa cosa restituire in base all'operazione indicata (lettura o scrittura).

Spogliamo l'interfaccia



- Le operazioni possibili sono:
 - lettura di eoc
 - letture di RBR
 - scrittura di soc
- Abbiamo una rete combinatoria che permette di gestire le porte tristate (e il registro SOC). Osserviamola:
 - Quando le variabili di pilotaggio non indicano una particolare operazione tutte le porte sono in alta impedenza.
 - Quando voglio leggere eoc avrò $e_E = 1$, quindi la porta relativa in conduzione. L'utente troverà eoc sul filo in posizione 0 (nei fili di dati).
 - Quando voglio leggere RBR avrò $e_x = 1$, enabler di tante porte tristate quante il numero di bit del registro (8). Tutte queste porte sono in conduzione, le rimanenti in alta impedenza. L'utente troverà in uscita, su tutti i fili di dati, il valore del registro RBR.
 - Quando voglio scrivere su soc avrò $e_S = 1$. Questa variabile è il clock per il registro SOC: se $e_s = 1$ il valore del registro impostato sarà quello posto in posizione 1 (nei fili di dati). Segue un nuovo ingresso nella SAR e quindi l'inizio di una nuova operazione di conversione, se necessario.