

# Reti Logiche

Francesco Bonistalli

January 2024

# Indice

<b>1 Note Assembler</b>	<b>2</b>
1.1 Rappresentazione numeri naturali e interi . . . . .	2
1.2 Codifica macchina e mnemonica delle istruzioni . . . . .	3
1.3 Note sulle principali istruzioni . . . . .	4
1.4 Programmare in Assembler . . . . .	7
1.5 Istruzioni che manipolano le stringhe . . . . .	8
1.6 Appendice: programmare in Assembler in modo efficiente . . . . .	9
<b>2 Reti Combinatorie</b>	<b>10</b>
2.1 Le reti logiche come modello astratto di sistemi fisici . . . . .	10
2.2 Generalità sulle Reti Combinatorie . . . . .	11
2.3 Algebra di Boole . . . . .	13
2.4 Reti combinatorie significative . . . . .	14
2.5 Modello strutturale universale per reti combinatorie . . . . .	15
2.6 Sintesi di reti in forma SP a costo minimo . . . . .	16
2.7 Sintesi in formato PS . . . . .	21
2.8 Porte logiche universali . . . . .	25
2.9 Porte tri-state . . . . .	25
2.10 Circuiti di ritardo e formatori di impulsi . . . . .	26
<b>3 Aritmetica dei Calcolatori</b>	<b>28</b>
3.1 Rappresentazione dei numeri naturali . . . . .	28
3.2 Elaborazione di numeri naturali tramite reti combinatorie . . . . .	29
3.3 Rappresentazione dei numeri interi . . . . .	37
3.4 Operazioni su interi in complemento alla radice . . . . .	39
<b>4 Reti Sequenziali</b>	<b>48</b>
4.1 La funzione di memoria e le reti sequenziali asincrone . . . . .	48
4.2 Il linguaggio verilog . . . . .	57
4.3 Reti Sequenziali Sincronizzate . . . . .	57
4.4 Descrizione e sintesi di reti sequenziali sincronizzate complesse . . . . .	64
<b>5 Struttura del calcolatore</b>	<b>67</b>
5.1 Interfacce . . . . .	75

# Capitolo 1

## Note Assembler

### 1.1 Rappresentazione numeri naturali e interi

La ALU è in grado di eseguire operazioni logiche su stringhe di bit e aritmetiche interpretando le stringhe come naturali o interi.

#### 1.1.1 Numeri naturali

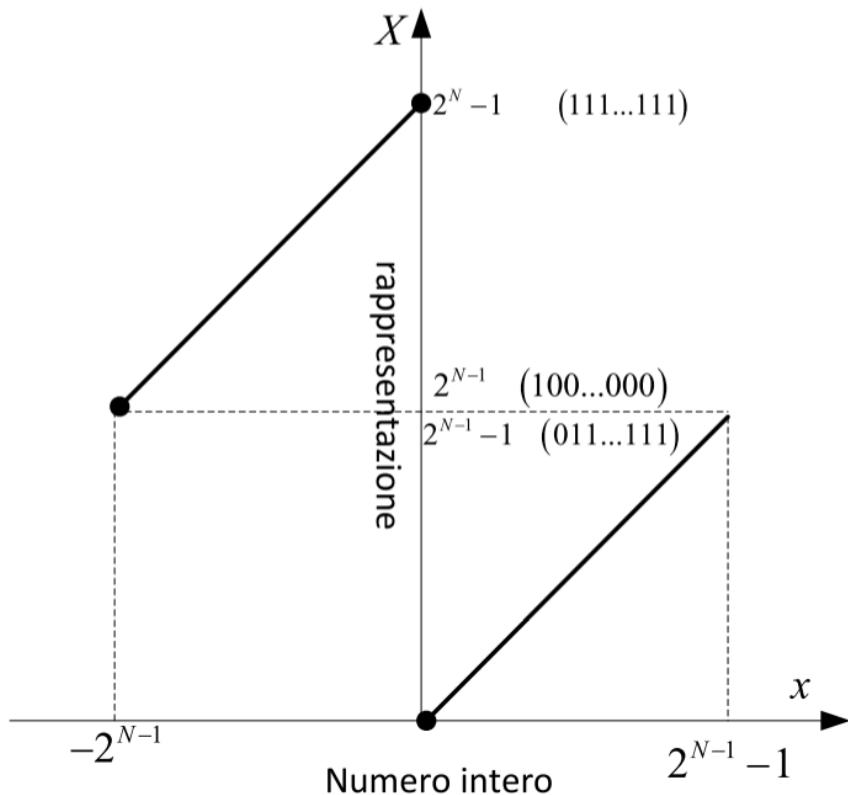
Su  $n$  bit si possono rappresentare numeri naturali nell'intervallo:

$$[0; 2^N - 1]$$

#### 1.1.2 Numeri interi

Su  $n$  bit si possono rappresentare numeri naturali nell'intervallo:

$$[-2^{N-1}; 2^{N-1} - 1]$$



### 1.1.3 Conversioni

Dati X numero naturale e x numero intero:

$$X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases}$$

In modo equivalente:

$$x = \begin{cases} X & X_{N-1} = 0 \\ -(\overline{X} + 1) & X_{N-1} = 1 \end{cases}$$

## 1.2 Codifica macchina e mnemonica delle istruzioni

3 concetti base:

- codifica **macchina**: serie di 0 e 1 che codificano le istruzioni che il processore esegue
- codifica **mnemonica**: modo simbolico di scrivere le serie di 0 e 1
- linguaggio **Assembler**: linguaggio che usa la codifica mnemonica per le istruzioni, completato poi da una serie di sovrastrutture sintattiche (esempio: nome simbolico locazioni di memoria usati nelle istruzioni)

Le istruzioni in linguaggio Assembler vengono tradotte in linguaggio macchina da un programma detto assemblatore

### **1.2.1 Esempio primo programma**

(non rilevante)

#### **Note sulla sintassi**

- gli operandi immediati vanno preceduti dal dollaro
- i registri vanno preceduti dal %
- in una istruzione un numero non preceduto da dollaro è un indirizzo di memoria

#### **Lunghezza degli operandi e suffissi**

Nel formato delle istruzioni è presente un suffisso di lunghezza che può essere:

- B
- W
- L

È obbligatorio metterlo solo nessuno degli operandi è un registro.

## **1.3 Note sulle principali istruzioni**

(Note a complemento del contenuto della dispensa)

### **1.3.1 1**

#### **MOVE**

Non è possibile trasferire dati da una locazione di memoria all'altra senza passare da un registro

### **1.3.2 Istruzioni aritmetiche**

Caratteristiche comuni:

- alcune considerano indifferentemente gli operandi come naturali o come interi (esempio la somma, dato che la somma delle rappresentazioni è uguale alla rappresentazione della somma)(vale anche per la differenza)
- queste istruzioni modificano i flag (nel farlo considerano 2 algoritmi distinti per naturali e interi)
- essendo il programmatore l'unico che sa cosa sta scrivendo è suo compito guardare il flag giusti

## ADD

Dato che la somma sta su  $n + 1$  bit quando si fa la somma per **naturali** se si genera un riporto, e di conseguenza la somma non è rappresentabile sugli  $n$  bit su cui viene eseguita, il CF varrà 1 in quanto gli verrà assegnato questo riporto (se non c'è riporto infatti gli verrà assegnato 0). Per quanto riguarda gli **interi** il CF è irrilevante:

- se i segni degli operandi sono discordi la somma è sempre rappresentabile.
- se i segni degli operandi sono concordi e la rappresentazione della somma ha lo stesso segno allora è rappresentabile, altrimenti non lo è e OF verrà settato a 1.

## SUBTRACT

Per verificare la rappresentabilità della differenza tra **naturali** devo guardare CF (come per la ADD).

Per quanto riguarda gli **interi**:

- la sottrazione di due numeri concordi è sempre rappresentabile.
- la sottrazione di due numeri discordi è rappresentabile soltanto se il risultato su  $N$  bit ha lo stesso segno del minuendo, altrimenti  $OF = 1$ .

## COMPARE

La cmp fa la seguente operazione:

$$null = dest - source$$

Si comporta quindi come la SUB ma non aggiorna il destinatario. È importante per le istruzioni di salto condizionato in quanto modifica comunque i flag.

**Nota:** *Eventuali istruzioni condizionali successive alla CMP faranno riferimento al destinatario rispetto al sorgente, ad esempio JA salterà se il destinatario è maggiore del sorgente.*

### 1.3.3 Moltiplicazioni e divisioni

Hanno istruzioni diverse per interi e naturali, inoltre richiedono un numero maggiore di bit.

#### Moltiplicazione

Per gestire in modo corretto le dimensioni si hanno 3 tipi di moltiplicazione:

- a 8 bit:  $AX = AL * source$
- a 16 bit:  $DX\_AX = AX * source$  (parte alta su DX e parte bassa su AX)
- a 32 bit:  $EDX\_EAX = EAX * source$  (parte alta su EDX parte bassa su EAX)

Il quale delle 3 venga utilizzata dipende dal numero di bit di *source* (oltre che dal suffisso quando è necessario).

## DIVIDE

Problema aggiuntivo: la divisione ha due risultati (quoziente e resto) non uno: Dato X diviso Y si ha:

- un resto R tale che:  $0 \leq R < Y$
- un quoziente Q tale che:  $0 \leq Q \leq X$

Si hanno quindi come per la MUL 3 diversi tipi a seconda di source:

- a 8 bit: il dividendo sarà a 16 bit (AX) e il quoziente e il resto andranno rispettivamente su AL e AH
- a 16 bit: il dividendo sarà a 32 bit (*DX\_AX*) e il quoziente e il resto andranno rispettivamente su AX e DX
- a 32 bit: il dividendo sarà a 64 bit (*EDX\_EAX*) e il quoziente e il resto andranno rispettivamente su EAX e EDX

Può succedere però che il quoziente può essere grande quanto il dividendo: in questo caso non entra sul numero di bit indicato e parte una eccezione (come nel caso di divisione per 0). Portando a stato di blocco il problema, questa eccezione non deve partire quindi il programmatore deve fare in modo che non avvenga.

## INTEGER DIVIDE

Stesse operazioni della DIV e stessi formati.

Note sui segni: il resto ha sempre il segno del dividendo, questo significa che il quoziente viene sempre approssimato all'intero più vicino allo 0.

### 1.3.4 Istruzioni di traslazione e rotazione

Muovono i bit dell'operando destinatario. La distinzione tra shift logico e aritmetico sta nella rappresentazione del numero su cui si va a operare:

- il logico si usa per i naturali
- l'aritmetico si usa per gli interi

## SAR

In particolare in questo caso si sfrutta l'operazione per gli interi dato che in testa viene inserito ciclicamente quello che era il MSB, questo per mettere di mantenere il segno dell'operando.

Attenzione: se sfruttato per la divisione per  $2^{src}$  non si ottiene lo stesso quoziente della IDIV. Con la SAR infatti si ha sempre approssimazione verso meno infinito.

## ROTATE THROUGH CARRY

A differenza delle normali rotate il CF è un bit compreso nel ciclo di rotazione, non un semplice bit in cui si mostra l'ultimo bit ruotato.

### 1.3.5 Istruzioni di controllo

#### JUMP IF CONDITION MET

Le istruzioni di salto condizionato seguono sempre qualche istruzione che tocca i flag.

### 1.3.6 Meccanismi di protezione e istruzioni privilegiate

Non tutte le istruzioni possono essere usate sempre (per varie ragioni), per questo motivo il processore può funzionare in due modalità:

- utente
- sistema

Nella prima modalità girano i programmi scritti dal programmatore, nella seconda quelli del sistema operativo nei quali possono essere usate alcune istruzioni esclusive:

- quelle di ingresso/uscita: *IN* e *OUT*
- HLT

## 1.4 Programmare in Assembler

Assembler è case insensitive per le parole chiave, per i nomi simbolici invece è case sensitive.

Dichiarazione di variabile:

.BYTE variabile

Dove BYTE specifica la dimensione (infatti si ha anche WORD e LONG).

### 1.4.1 Esempio

(esempi vari)

### 1.4.2 Controllo di flusso

(esempi vari)

#### Istruzioni LOOP/LOOPcond

L'istruzione loop ha come unico operando un indirizzo: decrementa ECX, non tocca i flag, e se dopo il decremento  $ECX \neq 0$  allora salta all'indirizzo, altrimenti continua.

Le loop condizionali appendono i suffissi E, Z, NE, NZ alla loop posta dopo una CMP: in questo caso il salto avviene se la condizione si verifica e se la condizione si verifica.

Attenzione: non si possono fare cicli di loop troppo lunghi in quanto il displacement dell'indirizzo può stare solo su 8 bit.

### 1.4.3 Sottoprogrammi e passaggio dei parametri

Le istruzioni per la gestione di sottoprogrammi sono CALL e RET: visto che non prevedono passaggio di parametri mentre un sottoprogramma generalmente opera su dei parametri, è necessario che il programmatore gestisca questo aspetto tramite uno (o entrambi) dei due metodi possibili:

- usare variabili condivise
- usare i registri

Attenzione: un sottoprogramma, dovendo fare dei conti, userà dei registri; a meno che essi non siano identificati come contenitori di parametri di ritorno non devono essere modificati, al momento della RET, dal punto di vista del programma principale. Per ovviare a questo problema si usa la pila: si fa la *PUSH* di tutti i registri che il sottoprogramma usa all'inizio di esso e alla fine si fa la *POP*, in ordine inverso di essi.

### Dichiarazione e inizializzazione dello stack nel sottoprogramma

Se in un programma usiamo lo stack sarebbe opportuno dichiararlo ed indirizzarlo (nel **nostro** ambiente non importa).

## 1.5 Istruzioni che manipolano le stringhe

In Assembler non esistono tipi o strutture dati, tuttavia è supportato il concetto di vettore:

- è possibile dichiarare vettori di variabili di una certa dimensione
- è possibile indirizzare la memoria con un indirizzamento indiretto che coinvolge fino a due registri + un displacement.

Il vettore è inoltre supportato da istruzioni stringa (sinonimo di vettore) che permettono di copiare operandi dalla memoria alla memoria o interi buffer in modo sequenziale sfruttando i registri indice *%ESI* e *%EDI* come puntatori in memoria.

L'istruzione *MOVSSuf* (in cui il suffisso va sempre specificato) è una istruzione senza operandi che fa due cose:

1. copia il sorgente (indirizzato da esi) nel destinatario (indirizzato da edi)
2. modifica entrambi i registri sommando il numero di byte specificato nel suffisso

Il prefisso *REP* aggiunge un byte all'istruzione e decrementa *ECX*, ripete quindi se *ECX* ≠ 0.

Il vantaggio di usare queste istruzioni è sia a livello di fetch (se ne fa uno solo nel caso dell'esempio) e si ha la possibilità di copiare da memoria a memoria.

Il funzionamento delle istruzioni stringa è guidato dal **direction flag** (DF) che se 0 indica come direzione in cui vengono copiate le stringhe "in avanti", ovvero verso l'indirizzo maggiore, se 1 "indietro" verso l'indirizzo minore. Per impostarlo si usano:

- STD
- CLD

L'istruzione *LODSsuf* carica nel registro *AL*, *AX*, *EAX* il contenuto della locazione (singola doppia o quadrupla) di memoria indirizzata da *ESI*, questo viene incrementato (o decrementato poi di 1,2 o 4).

L'istruzione *STOSSuf* copia il contenuto del registro *AL*, *AX*, *EAX* nella locazione di memoria indirizzata da *ESI*, poi incrementa (o dec).

Ci sono poi le versioni stringa delle istruzioni di in/out (entrambe sono comunque protette):

- *INSSuf* ingresso di suf byte dalla porta I/O il cui offset è contenuto in *DX*, l'operando viene inserito in memoria a partire dall'indirizzo puntato da *EDI*, in o dec poi.
- *OUTSuf* copia suf byte contenuti in memoria a partire da (*ESI*) alla porta il cui offset è contenuto in *DX*, poi inc o dec.

L'istruzione *CMPSSuf* confronta le locazioni puntate, in o dec poi.

L'istruzione *SCASsuf* confronta il contenuto del registro *AL*, *AX*, *EAX* con la/le locazione/i puntate da *EDI*, in o dec poi.

Attenzione: l'ordine in cui queste ultime due istruzioni comparano gli operandi è opposto alla *CMP*

### Prefissi di ripetizione

*REP* già visto, può essere usato con MOVS, LODS, STOS, INS, OUTS. (Nota: con LODS è privo di senso utilizzarlo in quanto sovrascrivi tot volte). (Nota: con le istruzioni di scandiscono (non spostano) stringhe alla ricerca di un qualche matching con *EAX* ha senso solo usare ripetizioni condizionate.

I prefissi *REPE* e *REPNE* si applicano soltanto a *CMP\$* e *SCAS* (continuano fino a un massimo di ECX ripetizioni).

## 1.6 Appendice: programmare in Assembler in modo efficiente (consigli vari)

# Capitolo 2

## Reti Combinatorie

### 2.1 Le reti logiche come modello astratto di sistemi fisici

Dato un sistema fisico semplice composto di due scatolette connesse da un filo, in cui nella scatoletta sx (osservatore) vi è una lampadina che si illumina in base alla tensione sul filo impostata dalla scatoletta dx (produttore), l'unica cosa che ci interessa sapere è che:

- se la tensione va da 0 a 0.8 volt la lampadina è spenta
- da 2 a 5 volt la lampadina è accesa
- nel mezzo l'osservatore è lui a decidere: a volte decide che è spenta altre che è accesa.

Questo sistema fisico è rappresentabile tramite un modello astratto detto rete logica nella quale ci sono:

- due sottoreti Tx e Rx
- una variabile logica  $w$  che le connette (in quanto variabile logica può solo assumere due valori)

In particolare una rete logica è caratterizzata da:

- N variabili logiche di ingresso. L'insieme degli N valori assunti in un certo istante si chiama **stato di ingresso**.
- M variabili logiche di uscita. L'insieme degli M valori assunti in un certo istante si chiama **stato di uscita**.
- una legge di evoluzione nel tempo che dice come le uscite evolvono in funzione degli ingressi.

Si classificano secondo i seguenti criteri:

- presenza o assenza di memoria: si dicono **combinatorie** le reti in cui lo stato di uscite dipende solo dallo stato di ingresso (ad uno stato di ingresso corrisponde uno e un solo stato di uscita), mentre sono **sequenziali** quelle reti il cui stato di uscita dipende dalla storia degli stati di ingresso (sono quindi reti con memoria).

- temporizzazione della legge di evoluzione: la legge che fa corrispondere le uscite agli ingressi può essere resa operativa in ogni istante (reti asincrone) oppure ad istanti discreti nel tempo (reti sincronizzate).

	<b>asincrono</b>	<b>sincronizzato</b>
<b>combinatorio</b>	1) Reti combinatorie	(non hanno un nome, sottocaso del caso 3)
<b>sequenziale</b>	2) Reti sequenziali asincrone	3) Reti sequenziali sincronizzate

### 2.1.1 Limiti del modello

Modello = modo (semplificato) di descrivere una realtà complessa. (commenti brevi)

#### Transizione dei segnali

La tensione (che determina una variabile logica) non cambia, per motivi fisici, istantaneamente; non si conosce quindi il momento esatto in cui la variabile logica cambia valore. Per evitare problemi nei nostri sistemi supporremo sempre che le transizioni fisiche avvengano in un tempo

$$\Delta t \ll \Delta T$$

con  $T$  tempo in cui le variabili logiche mantengono il proprio valore di regime.

#### Contemporaneità

Non siamo in grado di garantire che due grandezze fisiche varino contemporaneamente (abbiamo appena visto che le variazioni non sono istantanee), quindi se baso un ragionamento sulla variazione contemporanea di due variabili di ingresso non sarò poi in grado di realizzarla fisicamente. In conclusione: non è possibile che in una realizzazione fisica di un sistema si presentino in sequenza due stati di ingresso che differiscono tra loro per più di un bit = si ha che gli stati di ingresso consecutivi debbano essere adiacenti.

## 2.2 Generalità sulle Reti Combinatorie

### 2.2.1 Intro

(riassunto)

**Esempio 1** Invertitore: Particolare rete logica combinatoria con un ingresso e una uscita:

$$N = 1, \quad M = 1$$

quindi  $X = \{0, 1\}$  e  $Z = \{0, 1\}$ . La legge di corrispondenza a parole è: "se l'ingresso è zero l'uscita è uno e viceversa".  $\square$

#### Tempo di attraversamento

Una variazione negli ingressi viene a manifestarsi in uscita dopo un tempo finito (si dice che la rete va a regime dopo un tempo  $t$ ).

Chi pilota la rete deve evitare di variare lo stato di ingresso più velocemente del tempo di risposta, se questo viene evitato la rete si dice **pilotata in modo fondamentale**.

## 2.2.2 Modalità di descrizione delle reti combinatorie

Una rete combinatoria si descrive dicendo:

- numero degli ingressi
- numero di uscite
- quale è la funzione  $F$ , ovvero la descrizione funzionale

Quest'ultima può essere in vari modi:

- a parole
- tramite un linguaggio di descrizione (es. verilog)
- tabelle di verità

### Esempio

Note sull'esempio:

- se una uscita si setta dato un certo ingresso si dice che "l'uscita riconosce gli stati di ingresso  $x, y, z\dots$ "
- il non specificato non è un valore logico
- distinzione tra descrizione e sintesi: la prima è un modo formale per dire cosa fa una rete, la seconda è il progetto della realizzazione fisica della rete

## 2.2.3 Reti combinatorie elementari

**Teorema 1 Proprietà:** Una rete combinatoria a  $N$  ingressi ed  $M$  uscite può essere realizzata interconnettendo  $M$  reti combinatorie ed una uscita □

Data questa proprietà possiamo limitarci quindi a considerare soltanto reti con una uscita.

### Reti a zero ingressi

Generatori di costante, sono un caso degenere

### Reti ad un ingresso

Invertitore ed Elemento neutro

### Reti a due ingressi

Le diverse reti combinatorie a  $N$  ingressi e a una uscita sono  $2^{(2^N)}$

## 2.2.4 And ed Or a più ingressi

Si può estendere la funzione logica realizzata dalle porte AND e OR a  $N$  ingressi. Per farlo si possono utilizzare porte AND e OR a 2 ingressi (esistono anche i rispettivi a  $n$  ingressi).

## 2.3 Algebra di Boole

È un sistema algebrico basato su:

- variabili logiche (valgono 0 o 1)
- operatori logici che si applicano alle variabili (sono 3: complemento, prodotto logico, somma logica)

### 2.3.1 Proprietà degli operatori booleani

Involutiva del complemento: $\bar{\bar{x}} = x$
Commutativa della somma e del prodotto
Associativa della somma e del prodotto $x + y + z = (x + y) + z = \dots$
Distributiva della somma rispetto al prodotto e viceversa $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Complementazione: $x \cdot \bar{x} = 0, x + \bar{x} = 1$
Unione e intersezione: $x + 0 = x, x + 1 = 1, x \cdot 0 = 0, x \cdot 1 = x$
Idempotenza: $x + x = x, x \cdot x = x$
Teoremi di De Morgan: $\bar{x} \cdot \bar{y} = \bar{x} + \bar{y}$ $\bar{x + y} = \bar{x} \cdot \bar{y}$

### Teoremi di De Morgan per N variabili logiche

Valgono per N variabili logiche.

**Dimostrazione** Si dimostra per induzione sul numero delle variabili logiche: si dimostra che la proprietà vale per un certo numero  $n_0$  (si dimostra con la tabella di verità per 2 variabili logiche), poi si dimostra che se vale per un generico  $n \geq n_0$  allora vale anche per  $n + 1$  (passo induttivo); se ne conclude quindi che vale per ogni  $n \geq n_0$ .

Passo base: tabella di verità.

Passo induttivo: suppongo che sia vero per N variabili e dimostro che questo implica la validità per  $N+1$ . Scrivo quindi l'ipotesi induttiva con  $x$  da  $x_0$  a  $x_{N-1}$ . Passo poi alla tesi estendendo l'equazione fino ai termini  $x_N$ . A questo punto pongo i primi N termini (fino a  $x_{N-1}$ ) =  $\alpha$  ( $\alpha$  è una variabile logica in quanto funzione di variabili logiche). Mi trovo quindi nella situazione di 2 variabili logiche che verificano il teorema come visto nel passo base, da cui la tesi.

### Equivalenza tra espressioni dell'algebra di Boole e reti combinatorie

Data una rete combinatoria è sempre possibile trovare un'espressione booleana che ne descrive la funzione di corrispondenza e viceversa.

Esistono reti combinatorie logicamente equivalenti (sintetizzano la stessa tabella di verità) = a una stessa descrizione possono corrispondere più sintesi: a noi da ingegneri ci interessa trovare quella a costo minimo.

## 2.4 Reti combinatorie significative

### 2.4.1 Decoder

$N$  ingressi e  $2^N = p$  uscite. L'uscita  $j$ -esima riconosce lo stato di ingresso i cui bit sono la codifica di  $j$  in base 2.

**Sintesi** Faccio la sintesi di ogni uscita: in totale quindi unendo tutto avrà 4 AND a 2 ingressi con alcuni ingressi negati, per usare meno invertitori li sposto sulle variabili di ingresso.

### 2.4.2 Decoder con enabler (espandibile)

Il decoder base non è espandibile. Per fare ciò si aggiunge un ingresso di abilitazione  $e$ : se questo è 1 la rete si comporta come un decoder, se è a 0 tutte le uscite sono a 0.

Nella sintesi le porte AND degli ingressi sono in cascata con l'ingresso  $e$ , non ha senso tenerle in cascata quindi si sposta  $e$  come ingresso delle prime AND.

**Nota:** è sempre sciocco mettere due porte identiche in cascata (tolti eventuali vincoli).

### Esempio: costruzione di un decoder 4 to 16 da decoder 2 to 4

Si costruisce tramite 4 decoder 2 to 4 il cui ingresso di enabler è dato da ognuna delle uscite di un altro decoder comandato dall'ingresso  $e$ , le  $x$  invece sono 2 ( $x_0$  e  $x_1$ ) in ingresso ai 4 decoder mentre le altre 2 sono in ingresso a quello con l'enabler originale.

### 2.4.3 Demultiplexer

Rete con  $N + 1$  ingressi e  $p = 2^N$  uscite.

Un ingresso è la variabile da commutare: l'uscita  $j$ -esima insegue la variabile da commutare se e solo se gli altri ingressi (variabili di comando) sono la codifica in base 2 di  $j$ , altrimenti vale 0.

Ogni uscita  $z_n$  può essere sintetizzata come prodotto tra  $x$  e le variabili  $b$  opportunamente dirette o negate. (Stessa descrizione funzionale del decoder con enabler).

### 2.4.4 Multiplexer

Rete con  $N + 2^N$  ingressi ed 1 uscita. Le variabili di comando qui stabiliscono quale ingresso è connesso all'uscita unica, in altre parole:

$$z = x_i \Leftrightarrow (b_{N-1} \dots b_1 b_0)_2 = i$$

La sua sintesi è: le variabili di comando sono in ingresso a un decoder le cui uscite sono in AND con gli ingressi, le uscite di questi AND sono tutte in OR. Si osserva però che data la struttura del decoder si hanno 2 AND in cascata, si compattano quindi utilizzando AND a più ingressi dati da: ingressi e variabili di comando dirette e negate (sempre tutto in OR alla fine). Una specifica funzionale è data quindi dalla somma di operandi formati ognuno dalla somma di un  $x_n$  (con  $n = 0, \dots, p - 1$ ) in AND con i relativi  $b$  diretti e negati.

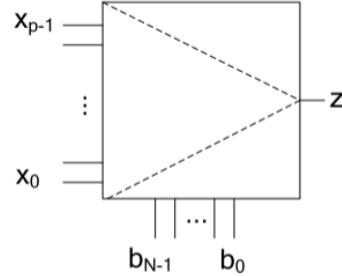
Si nota quindi che questa rete è a due livelli di logica (la strada più lunga tra un ingresso e l'uscita è data dall'attraversamento di due porte). (le not non si contano)

## Il multiplexer come rete combinatoria universale

Ci si rende conto che un multiplexer con  $N$  variabili di comando è in grado di realizzare qualunque legge combinatoria che ha  $N$  ingressi ed una uscita.

Basta infatti scrivere la tabella di verità della rete che si vuole sintetizzare: gli  $N$  ingressi saranno le variabili di comando, in base poi al valore dell'uscita attacco l'ingresso a tensione o a massa.

$j$	$b_2$	$b_1$	$b_0$	$z$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0



Questa osservazione ha delle importanti conseguenze:

- nel multiplexer si hanno 2 livelli di logica e si usano solo porte AND, OR, NOT
- un multiplexer realizza una qualunque rete combinatoria a una uscita
- una rete combinatoria a più uscite può essere scomposta in più reti ad una uscita messe "in parallelo"

**Teorema 2:** Ogni rete combinatoria può essere costruita combinando AND, OR, NOT in al più due livelli di logica  $\square$

Si ha quindi un limite superiore al ritardo di una qualunque rete combinatoria.

## 2.5 Modello strutturale universale per reti combinatorie

Vediamo come sintetizzare una rete logica ad  $N$  ingressi ed  $M$  uscite a partire da una tabella di verità dato che abbiamo appena visto come una rete logica ad  $N$  ingressi ed  $M$  uscite può essere sintetizzata con un decoder a  $N$  ingressi seguito da una barriera di  $M$  OR (quest'ultimo viene detto **modello strutturale universale**).

Si ha quindi:

- $N$  ingressi diretti o negati
- barriera di  $2^N$  AND
- $M$  OR che hanno come ingresso le uscite significative degli AND

Se voglio ottimizzare questo modello inizio con il togliere gli AND che non producono uscite significative. Per proseguire con l'ottimizzazione passo poi all'algebra di Boole: scrivo le uscite come somme (OR finali) di prodotti (barriera di AND del decoder) degli ingressi diretti o negati.

**Definizione 1 Forma canonica SP:** Ogni variabile di uscita è la somma del prodotto di tutte le variabili di ingresso dirette o negate.  $\square$

Sfruttando quindi le proprietà dell'algebra di Boole posso ottenere delle forme ridotte, in particolare: si arriva a una forma detta **forma SP** ovvero ogni uscita è la somma del prodotto di alcune variabili di ingresso dirette o negate. (meno porte di così non si possono mettere a meno di non uscire dal modello SP).

## 2.6 Sintesi di reti in forma SP a costo minimo

Troviamo ora un modo formale di operare (sempre su reti con una uscita, se ho più uscite basta operare separatamente su ogni uscita, non per forza però il risultato sarà ottimo).

Per cominciare bisogna distinguere due criteri di costo:

- a porte: ogni porta costa una unità di costo
- a diodi: ogni ingresso costa una unità di costo

Il metodo che stiamo per proporre non produce la rete di costo minimo in **assoluto** ma produce reti a 2 livelli di logica in forma SP di costo minimo.

Punto di partenza: risultato dovuto a Shannon: "è sempre possibile scrivere qualunque legge  $F$  di una rete combinatoria come somma di prodotti degli ingressi diretti o negati". Data quindi una legge  $z = f(x_{N-1}, \dots, x_0)$  posso scrivere l'espansione di Shannon come:

$$\begin{aligned} z &= f(0, \dots, 0, 0) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot \overline{x_0} \\ &\quad + f(0, \dots, 0, 1) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot x_0 \\ &\quad \dots \\ &\quad + f(1, \dots, 1, 0) \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot \overline{x_0} \\ &\quad + f(1, \dots, 1, 1) \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot x_0 \end{aligned}$$

Figura 2.1: È lo stesso formato che avevo nel multiplexer (le somme erano composte dai prodotti di  $x_i$  e i vari  $b$  opportunamente pilotati

da cui posso ottenere la forma canonica SP osservando che:

- se  $f(\dots) = 0$  dato che  $0 \cdot \alpha = 0$  tutto il termine su quella riga vale 0
- se  $f(\dots) = 1$  dato che  $1 \cdot \alpha = \alpha$  posso togliere  $f(\dots)$  dal prodotto

**Definizione 2 Mintermine:** ciascuno dei termini della somma si chiama mintermine e corrisponde a uno stato di ingresso riconosciuto dalla rete (forma canonica SP = lista dei mintermini).  $\square$

Per andare verso una soluzione di costo minore a partire dalla lista dei mintermini applico, esaustivamente, le regole:

$$\begin{cases} \alpha x + \alpha \bar{x} = \alpha x + \alpha \bar{x} + \alpha \\ \alpha + \alpha = \alpha \end{cases}$$

(la prima regola mi consente di fondere i mintermini: permette di mantenere i termini iniziali aggiungendone uno per magari semplificarlo con altri elementi, la seconda di non inserire duplicati).

Si procede quindi operativamente, a partire dalla lista dai mintermini ( $k_0$ ), controllando ogni coppia che, nel caso verifichi la prima regola, genera un nuovo termine (il termine in comune tra i due analizzati) di una nuova lista ( $k_1$ ); si procede poi a partire da  $k_1$  e si va avanti finché possibile.

Sfruttando quindi il fatto che sto sommando somme di cose vere posso scrivere  $z = k_0 + k_1 + \dots$  ottenendo quindi quella che viene detta lista degli implicanti.

**Definizione 3 Implicante:** un implicante è il prodotto di alcune variabili di ingresso dirette o negate che riconosce alcuni stati di ingresso, quindi un qualsiasi termine di questa lista.  $\square$

**Nota:** un mintermine è quindi un caso particolare di implicante.

Dunque, sfruttando la legge  $\alpha x + \alpha = \alpha$  posso eliminare dalla lista degli implicanti tutti quelli che hanno prodotto qualcosa per fusione: si ottiene quindi la **lista degli implicanti principali**.

**Definizione 4 Implicante principale:** un implicante è detto principale se non è in grado di fondere con nessun altro implicante.  $\square$

**Nota:** Un mintermine vale 1 in corrispondenza di uno ed un solo stato di ingresso riconosciuto dalla rete, un implicante vale 1 in corrispondenza di qualche stato di ingresso riconosciuto dalla rete.

La lista degli implicanti principali costa meno della forma canonica SP: meno termini e probabilmente meno ingressi, potrebbe però essere ancora ridondante.

Cerco quindi la:

**Definizione 5 Lista di copertura:** lista di implicanti la cui somma è una forma SP per la funzione  $f$ . (tutti gli stati di ingresso sono coperti).  $\square$

**Definizione 6 Lista di copertura non ridondante:** tale che se tolgo un elemento dalla lista smette di essere una lista di copertura. (esempio: lista dei mintermini è non ridondante (viene dall'espansione di Shannon), se ne tolgo uno ho uno stato di ingresso non coperto).  $\square$

Vediamo dei metodi per generare la lista degli implicanti principali.

### 2.6.1 Metodo di Quine-McCluskey

Parto dalla lista degli stati di ingresso riconosciuti dalla rete, li raggruppo in base al numero di 1 presenti (gruppi in ordine crescente). A questo punto cerco quelli che fanno fusione solo tra gruppi adiacenti e creo la nuova lista (sempre raggruppata). Ciclo finché posso.

Quelli che non hanno fatto fusione sono principali. (posso scrivere la forma SP).

### 2.6.2 Mappe di Karnaugh

Una mappa di Karnaugh per una rete a N ingressi è una matrice di  $2^N$  celle:

- ogni cella è individuata da uno stato di ingresso che ne costituisce le coordinate
- ogni cella contiene il valore dell'uscita corrispondente a quello stato di ingresso.

Attenzione! le coordinate le devo scrivere in modo che celle contigue sulla mappa abbiano coordinate adiacenti(cambia solo una coordinata).

**Nota:** si usano per reti a massimo 4 ingressi (volendo anche 5), per reti con più ingressi usare il metodo Quine-McCluskey.



Figura 2.2: Standard da usare per le coordinate

**Definizione 7 Sottocubo di ordine 1:** cella che contiene un 1: stato di ingresso riconosciuto dalla rete.  $\square$

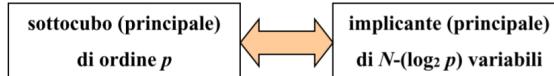
**Definizione 8 Adiacenza tra sottocubi :** differiscono per una sola coordinata avendo le altre  $N - n$  identiche, dove  $n$  indica l'ordine del sottocubo (nella rappresentazione la coordinata che varia va indicata con -)  $\square$

**Definizione 9 Sottocubo principale:** sottocubo tale per cui non esiste nessun sottocubo più grande che lo copre completamente.  $\square$

**Definizione 10 Lista di copertura:** insieme qualunque di sottocubi che coprono tutti i sottocubi di ordine 1  $\square$

**Definizione 11 Lista di copertura non ridondante:** tale che se tolgo un sottocubo non è più una lista di copertura  $\square$

**Teorema 3 Corrispondenza biunivoca tra implicanti principali della legge F e sottocubi principali della mappa di Karnaugh:** .



$\square$

**Nota:** un sottocubo di ordine 1 è un mintermine.

#### Algoritmo di ricerca dei sottocubi principali mediante mappe di Karnaugh

1. Parto dai sottocubi più grandi e li segno **tutti** (sono per definizione principali)
2. se ho coperto l'intera mappa ho finito
3. altrimenti dimezzo la dimensione e segno **tutti** quelli non già coperti dall'ordine maggiore, ritorno al passo 2.

**Nota:** La lista degli implicanti si scrive nel seguente modo: metto in prodotto le coordinate (dirette o negate) che non sono a "non specificato" le quali formano un certo sottocubo, sommo poi tutti questi termini composti da prodotti.

#### Ricerca delle liste di copertura non ridondanti

Eravamo arrivati alla lista degli implicanti principali che però può essere ridondante. Sulla mappa di Karnaugh quindi:

1. Se dei sottocubi sono gli unici a coprire un certo sottocubo di ordine uno, questi sono **essenziali**.
2. Se nell'individuare quelli essenziali si copre qualche altro sottocubo quest'ultimo va tolto (**assolutamente eliminabili**).
3. Qualche sottocubo potrebbe non rientrare nelle due precedenti categorie: è **semplicemente eliminabile**
4. A questo punto per ottenere la lista di costo minimo devo generare tutte le liste di copertura, non ridondanti, che includono un sottoinsieme di sottocubi principali semplicemente eliminabili
5. Tra tutte le liste ottenute scelgo quella a costo minimo (secondo il criterio di costo prestabilito)

Per svolgere gli ultimi 2 passi si deve:

1. considerare un semplicemente eliminabile come essenziale, aggiungerlo quindi alla lista di copertura e di conseguenza rendere eliminabile qualche altro sottocubo
2. considerarlo come assolutamente eliminabile, quindi toglierlo e dunque aggiungerne altri che saranno diventati essenziali

Vado avanti finché non ho esplorato tutto l'albero, scelgo quindi quella di costo minimo (a seconda del criterio).

**Nota:** Bisogna stare attenti che ad ogni passo non si generino liste di copertura ridondanti.  
*Esempio da non fare:*

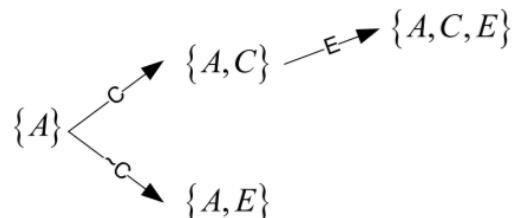
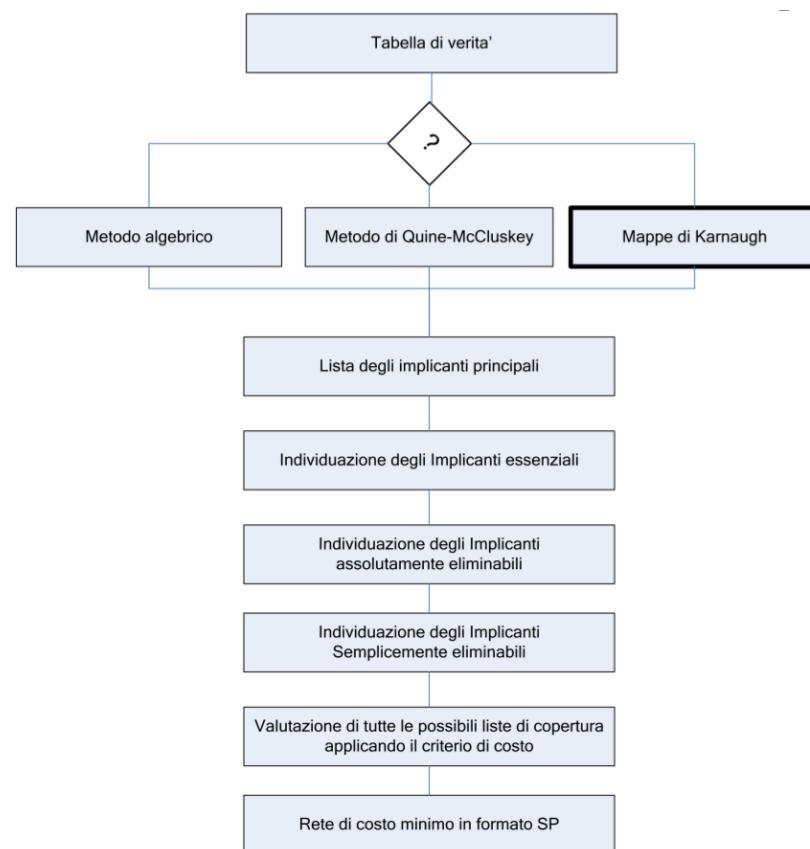


Figura 2.3: Non posso inserire E dopo aver inserito C, ottengo una lista ridondante

## Riepilogo - procedura per la sintesi a costo minimo SP



## Riepilogo - definizione e classificazione di implicanti e sottocubi

Algebra	Definizione
Forma Canonica SP	Data una legge $z = f(x_{N-1}, \dots, x_0)$ , la FC SP è quella che si ottiene dall'espansione di Shannon della legge, cioè: $z = f(0, \dots, 0, 0) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot \overline{x_0}$ $+ f(0, \dots, 0, 1) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot x_0$ $\dots$ $+ f(1, \dots, 1, 0) \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot \overline{x_0}$ $+ f(1, \dots, 1, 1) \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot x_0$ applicando le identità $0 + \alpha = \alpha$ , $1 \cdot \alpha = \alpha$ , $0 \cdot \alpha = 0$
Mintermine	Prodotto di <b>tutte</b> le variabili di ingresso dirette o negate, che compare in una <b>forma canonica SP</b> e riconosce <b>uno stato di ingresso</b> .
Implicante	Prodotto di <b>alcune</b> variabili di ingresso dirette o negate, che compare in una <b>forma SP</b> di una legge di corrispondenza. Si ottiene a partire dalla forma canonica SP applicando esaustivamente le seguenti regole: $\left\{ \begin{array}{l} \alpha x + \alpha \bar{x} = \alpha x + \alpha \bar{x} + \alpha \\ \alpha + \alpha = \alpha \end{array} \right.$ Riconosce <b>alcuni stati di ingresso</b> .
Implicante Principale	Implicante che si ottiene dalla lista degli implicanti applicando esaustivamente la legge $\alpha x + \alpha = \alpha$
I.P. Essenziale	Implicante che è l' <b>unico</b> , tra quelli principali, ad implicare un dato mintermine (a riconoscere uno stato di ingresso).
I.P. Assolut. Eliminabile	Implicante che riconosce solo stati di ingresso già riconosciuti da I.P. essenziali.
I.P. Semplic. Eliminabile	<u>Implicante</u> che riconosce solo stati di ingresso riconosciuti da altri I.P., almeno uno dei quali non riconosciuto da I.P. essenziali.

### Note aggiuntive

1. Nella ricerca degli implicanti principali considero i valori non specificati come degli 1 (mi fa comodo avere implicanti più grandi)
2. Nella ricerca di quelli essenziali considero i non specificati come 0

## 2.7 Sintesi in formato PS

Esiste sempre la possibilità di sintetizzare una rete in formato PS.

Lo schema di realizzazione è identico ma con AND e OR scambiati.

Le varie somme (messe poi a prodotto) prendono il nome di **implicati** (è possibile sviluppare tutta una teoria duale).

Per fare la sintesi in forma PS di una data legge F si affronta così:

1. data una legge F ricavo la legge  $\overline{F}$ : fa corrispondere a ogni stato di ingresso il complemento di quello che farebbe F (nella tabella di verità scambio gli 1 con gli 0)
2. realizzo una sintesi SP di  $\overline{F}$
3. ottengo una sintesi di F inserendo un invertitore in uscita alla rete ottenuta al punto 2
4. applico i teoremi di De Morgan all'indietro a partire dall'ultimo livello di logica e ottengo:
  - al posto della somma finale negata il prodotto dei suoi ingressi negati

- al primo livello di logica al posto di ciascun prodotto negato, somme dei suoi ingressi negati

Dal punto di vista logico:

1. scrivo  $\overline{F}$  in forma SP:  $\overline{z} = P_1 + \dots + P_k$
2. applico De Morgan sull'uscita:  $z = \overline{\overline{z}} = \overline{P_1 + \dots + P_k} = \overline{P_1} \cdot \dots \cdot \overline{P_k}$
3. applico De Morgan sull'ingresso:  $\overline{P_i} = \overline{\Pi x_j} = \Sigma \overline{x_j}$

La rete così ottenuta è in forma PS.

**Teorema 4:** .

- se  $\overline{F}$  è in forma canonica SP, allora F è in forma canonica PS
- se la sintesi SP di  $\overline{F}$  è a costo minimo lo è anche la sintesi PS di F

□

**Nota:** Presa una sintesi di una legge F la sintesi di  $\overline{F}$  si dice *sintesi duale*.

**Nota:** non si può determinare a prescindere se, data una legge F, è meno costosa la sua sintesi a costo minimo SP o PS.

### 2.7.1 Approfondimento: sintesi PS per via algebrica

### 7.1 Approfondimento: Sintesi PS per via algebrica

Esiste una teoria completamente duale a quella SP per arrivare alla forma PS. In particolare, data una legge  $z = f(x_{N-1}, \dots, x_0)$ , posso infatti scriverne l'**espansione di Shannon** come segue:

$$\begin{aligned} z = & \left[ f(0, \dots, 0, 0) + x_{N-1} + x_{N-2} + \dots + x_1 + x_0 \right] \\ & \cdot \left[ f(0, \dots, 0, 1) + x_{N-1} + x_{N-2} + \dots + x_1 + \overline{x}_0 \right] \\ & \cdots \\ & \cdot \left[ f(1, \dots, 1, 0) + \overline{x}_{N-1} + \overline{x}_{N-2} + \dots + \overline{x}_1 + x_0 \right] \\ & \cdot \left[ f(1, \dots, 1, 1) + \overline{x}_{N-1} + \overline{x}_{N-2} + \dots + \overline{x}_1 + \overline{x}_0 \right] \end{aligned}$$

Notare che le variabili a sommare sono complementate rispetto allo stato su cui è calcolata la funzione.

Dall'espansione di Shannon della legge, posso ottenere la **forma canonica PS** osservando che:

- se  $f(x_{N-1}, \dots, x_0) = 1$ , tutto il termine corrispondente sulla riga vale 1 (in quanto  $1 + \alpha = 1$ ). Allora, visto che  $1 \cdot \alpha = \alpha$ , posso togliere l'intera riga
- se  $f(x_{N-1}, \dots, x_0) = 0$ , visto che  $0 + \alpha = \alpha$ , posso togliere uno degli addendi della somma.

Un po' di nomenclatura:

- forma **PS** perché  $z$  è ottenuta come prodotto di somme
- forma **canonica** perché ogni prodotto contiene **tutti gli ingressi** diretti o negati
- ciascuno dei termini del prodotto si chiama **maxtermine**, e corrisponde ad uno stato di ingresso non riconosciuto dalla rete.

Si osserva che ogni maxtermine a prodotto vale **sempre 1**, tranne quando lo stato di ingresso ha la configurazione opposta di variabili. Quindi, il prodotto vale zero se e soltanto se uno dei maxtermeni è nullo, altrimenti vale uno.

Prendiamo come esempio la tabella di verità di una rete a 4 ingressi ed 1 uscita, già usata nella sintesi SP.

$x_3$	$x_2$	$x_1$	$x_0$	$z_0$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Espansione di Shannon

$$z = [0 + x_3 + x_2 + x_1 + x_0] \\ \cdot [1 + x_3 + x_2 + x_1 + \bar{x}_0]$$

...

$$\cdot [0 + \bar{x}_3 + \bar{x}_2 + \bar{x}_1 + x_0]$$

$$\cdot [0 + \bar{x}_3 + \bar{x}_2 + \bar{x}_1 + \bar{x}_0]$$

Forma canonica PS  
(lista dei maxtermini)

$$z = [x_3 + x_2 + x_1 + x_0] \\ \cdot [\bar{x}_3 + \bar{x}_2 + x_1 + x_0] \\ \cdot [x_3 + \bar{x}_2 + x_1 + \bar{x}_0] \\ \cdot [\bar{x}_3 + x_2 + \bar{x}_1 + x_0] \\ \cdot [\bar{x}_3 + \bar{x}_2 + x_1 + x_0] \\ \cdot [\bar{x}_3 + \bar{x}_2 + x_1 + \bar{x}_0] \\ \cdot [\bar{x}_3 + \bar{x}_2 + \bar{x}_1 + x_0] \\ \cdot [\bar{x}_3 + \bar{x}_2 + \bar{x}_1 + \bar{x}_0]$$

Partendo dalla lista dei maxtermini, applico **esaurientemente** le due seguenti regole:

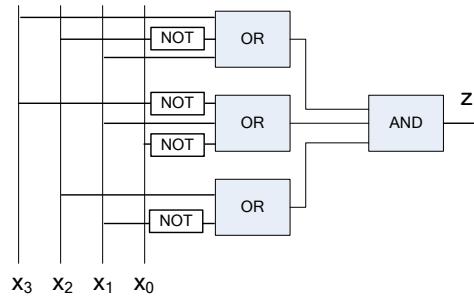
$$\left\{ \begin{array}{l} (\alpha + x) \cdot (\alpha + \bar{x}) = (\alpha + x) \cdot (\alpha + \bar{x}) \cdot \alpha \\ \alpha \cdot \alpha = \alpha \end{array} \right.$$

- La prima legge consente di **fondere i maxtermini**. Dati due termini che differiscono per **una sola variabile**, che è diretta in un caso e negata nell'altro, posso produrre un termine che contiene la sola parte comune
- La seconda legge ci ricorda di **non inserire duplicati**.

In questo modo, dalla lista dei maxtermini costruisco la lista degli **implicati**.

## 7.2 Esercizio (per casa)

Data la rete combinatoria di figura:



- 1) disegnare la mappa di Karnaugh per la legge  $z$ , sapendo che non è possibile che si presentino stati di ingresso in cui tutte le variabili hanno lo stesso valore.

## 2.8 Porte logiche universali

Le porte NAND e NOR sono dette porte logiche universali: ogni legge combinatoria può essere sintetizzata usando esclusivamente uno solo questi due tipi di porta.

	Porta	realizz. a NAND	realizz. a NOR
<b>NOT</b> $x = x \cdot x \Rightarrow \bar{x} = \overline{\bar{x} \cdot x}$			
<b>AND</b> $x \cdot y = \overline{(\bar{x} \cdot \bar{y})}$ $x \cdot y = \bar{\bar{x}} \cdot \bar{y}$			
<b>OR</b> $x + y = \overline{\bar{x} \cdot \bar{y}}$ $x + y = \overline{(x \cdot y)}$			

Dunque dato che si può fare soltanto con AND OR NOT posso farlo solo con una tra NAND e NOR.

### 2.8.1 Sintesi a porte NAND

Si parte dal circuito in forma SP, sostituisco la porta OR con l'equivalente a NAND, faccio lo stesso con le AND, (i not sugli ingressi li lascio stare in quanto non fanno parte della sintesi), elimino ogni coppia di NAND in cascata.

Data una sintesi SP la sintesi a porte NAND ha lo stesso costo (per entrambi i criteri).

Dal punto di vista algebrico un'espressione a NAND si ottiene a partire da quella SP complementando 2 volte e applicando De Morgan una volta:

$$\begin{aligned} z &= P_1 + P_2 + \dots + P_K \\ &= \overline{\overline{P_1 + P_2 + \dots + P_K}} \\ &= \overline{\overline{P_1} \cdot \overline{\overline{P_2}} \cdot \dots \cdot \overline{\overline{P_K}}} \end{aligned}$$

### 2.8.2 Sintesi a porte NOR

Equivalenti a quella NAND partendo però dalla sintesi in forma PS.

**Nota:** Data una sintesi PS di un circuito, la sintesi a porte NOR che gli corrisponde ha lo stesso costo (per entrambi i criteri).

## 2.9 Porte tri-state

Per evitare i problemi che si hanno a livello elettrico connettendo le uscite di più reti su uno stesso bus si ha la necessità di apparati capaci di disabilitare (fisicamente) le uscite: le porte tri-state.

**Definizione 12 Porte tri-state:** quando b vale 1 la porta si comporta come elemento neutro, quando b vale 0 l'uscita è come se fosse disconnessa dal resto della rete (si trova in alta impedenza).  $\square$

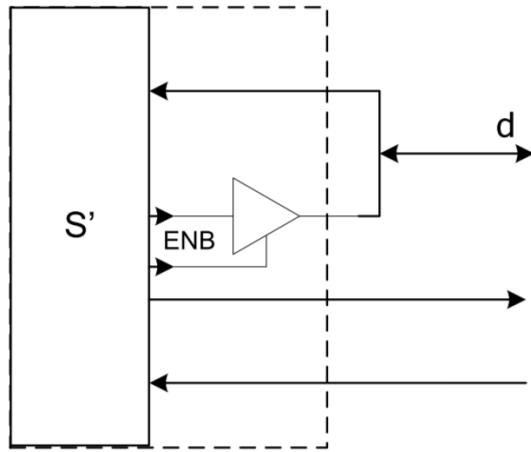


Figura 2.4: Esempio di forchettatura che sfrutta le porte tri-state

**Nota:** L'alta impedenza non è un valore logico e come tale non si propaga.

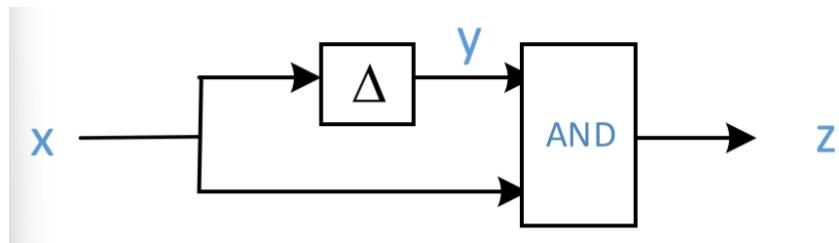
## 2.10 Circuiti di ritardo e formatori di impulsi

I buffer di solito si realizzano con un numero pari di porte NOT e in quanto tali hanno un ritardo simmetrico (identico sulle variazioni 1-0 e 0-1).

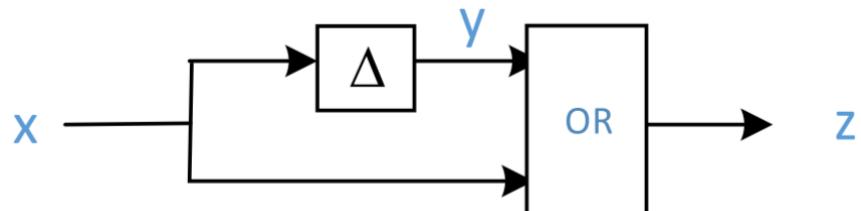
### 2.10.1 Circuiti di ritardo

Circuiti con ritardo asimmetrico si realizzano con AND, OR e un buffer di ritardo noto.

#### Ritardo grande su 0-1



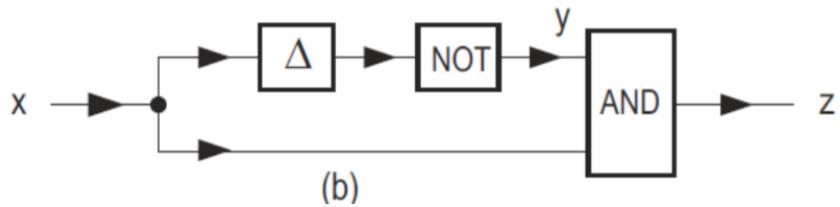
#### Ritardo grande su 1-0



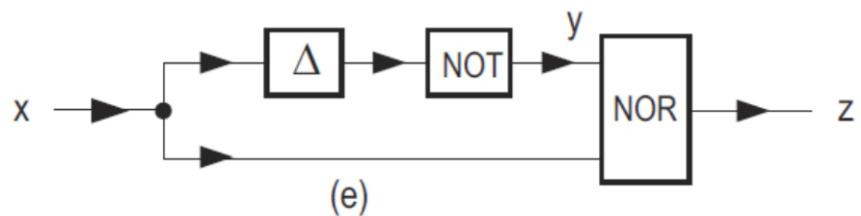
## 2.10.2 Formatori di impulsi

Circuiti combinatori che generano un impulso di durata nota sull'uscita

**Impulso su fronte di salita**



**Impulso su fronte di discesa**



# Capitolo 3

## Aritmetica dei Calcolatori

### 3.1 Rappresentazione dei numeri naturali

Un sistema di rappresentazione è formato da:

- una base di rappresentazione  $\beta \geq 2$
- $\beta$  cifre, ovvero simboli a cui è associato un numero naturale da 0 a  $\beta - 1$
- una legge di corrispondenza che fa corrispondere a ogni sequenza di cifre un numero naturale

In un sistema posizionale la legge che fa corrispondere un numero naturale con la sua rappresentazione è

$$A = \sum_{i=0}^{n-1} a_i \cdot \beta^i$$

**Nota:** Notazione posizionale significa che nella rappresentazione di un numero naturale una cifra contribuisce a determinare il numero in modo differente a seconda della propria posizione. (non sempre così: vedi numeri romani)

#### 3.1.1 Teorema della divisione con resto

**Teorema 5 Teorema della divisione con resto:** Dato  $x \in \mathbb{Z}, \beta \in \mathbb{N}, \beta > 0$  esiste ed è unica la coppia di numeri  $q, r$  con  $q \in \mathbb{Z}$  e  $r \in \mathbb{N}, 0 \leq r < \beta$  tale che  $x = q \cdot \beta + r$ .  $\square$

**Nota:** Vale anche per i naturali in quanto sottoinsieme degli interi.

**Teorema 6 Dimostrazione:**  $\square$

Chiamiamo quindi  $q$  quoziente e  $r$  resto della divisione di  $x$  per  $\beta$  e li indichiamo come:

$$q = \left\lfloor \frac{x}{\beta} \right\rfloor, \quad r = |x|_\beta$$

**Nota:** i simboli nella definizione di  $q$  indicano "parte intera inferiore"

## Proprietà dell'operatore modulo

Dato  $\alpha \in \mathbb{N}, a > 0$ :

1.  $|x + k \cdot \alpha|_\alpha = |x|_\alpha$  con  $k$  intero.
2.  $|x \pm y|_\alpha = ||x|_\alpha \pm |y|_\alpha|_\alpha$
3.  $|x \cdot y|_\alpha = ||x|_\alpha \cdot |y|_\alpha|_\alpha$

### 3.1.2 Correttezza ed unicità della rappresentazione dei numeri in una data base

Per trovare la rappresentazione in qualunque base di un numero naturale applico iterativamente il Th. del resto (algoritmo delle divisioni successive). Il teorema mi garantisce che la n-upla trovata sia unica.

#### Rappresentazione su un numero finito di cifre

Date  $n$  cifre a disposizione e una base  $\beta$  potrò rappresentare  $\beta^n$  numeri naturali.

Dato che si vogliono rappresentare intervalli che includono lo 0 il naturale più grande rappresentabile sarà  $\beta^n - 1$  la cui rappresentazione è

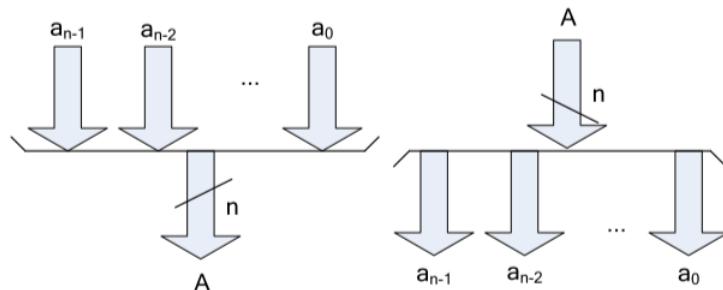
$$A = \sum_{i=0}^{n-1} (\beta - 1) \cdot \beta^i$$

ovvero il numero composto solo da cifre di valore massimo.

## 3.2 Elaborazione di numeri naturali tramite reti combinatorie

Abbiamo come obiettivo costruire reti logiche che elaborino numeri naturali rappresentati in una base (genericamente base 2). Si tratterà di reti combinatorie.

Per poter arrivare a realizzarle ci dotiamo di una notazione indipendente dalla base: A è individuato da  $n$  cifre in base  $\beta$ :



Note varie sulla codifica ecc.

Noi lavoreremo per lo più sulle cifre e non sulla codifica per quanto riguarda le operazioni aritmetiche.

### 3.2.1 Complemento

**Definizione 13 Complemento:** Definiamo complemento di  $A$  in base  $\beta$  su  $n$  cifre il numero:

$$\bar{A} = \beta^n - 1 - A$$

ovvero il numero che sommato ad  $A$  da il massimo numero rappresentabile in quella base.  $\square$

**Nota:** il complemento richiede che si specifichi il numero di cifre.

### Circuito logico per l'operazione di complemento

Basta saper sintetizzare una rete che fa il complemento di una singola cifra, in base 2 è facile da realizzare basta utilizzare una barriera di  $n$  NOT.

### 3.2.2 Moltiplicazione e divisione per una potenza della base

Notiamo come proprietà della notazione posizionale che:

- moltiplicare un numero su  $n$  cifre per  $\beta^k$  significa costruire un nuovo numero di  $n + k$  cifre le cui  $k$  cifre meno significative sono 0.
- il quoziente della divisione di un numero su  $n$  cifre per  $\beta^k$  è un numero costituito dalle  $n - k$  cifre più significative del numero di partenza
- il resto della divisione di un numero su  $n$  cifre per  $\beta^k$  è un numero costituito dalle  $k$  cifre meno significative del numero di partenza

Le reti che implementano queste operazioni sono gli **shift** e hanno complessità nulla.

**Nota:** Anche le operazioni di concatenamento e di scomposizione di e in due numeri hanno complessità nulla.

### 3.2.3 Estensione di campo

Operazione con cui si intende rappresentare un numero naturale con un numero maggiore di cifre. Si fa aggiungendo zeri in testa.

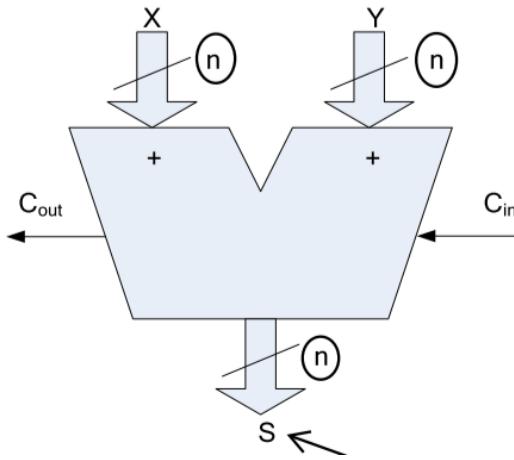
### 3.2.4 Addizione

L'algoritmo che abitualmente usiamo per la base 10 vale per tutte le basi.

Dati quindi  $X, Y$  in base  $\beta$  con  $0 \leq X, Y < \beta^n - 1$  e dato il riporto entrante  $0 \leq C_{in} \leq 1$  vogliamo calcolare la somma  $Z$ .

**Nota:** il ruolo del riporto entrante è fondamentale per la modularità dell'operazione

$Z$  è sempre rappresentabile su  $n + 1$  cifre ma non sempre su  $n$ , la cifra più significativa può essere solo 0 o 1. Ci adoperiamo quindi ad usufruire di  $n$  cifre per la somma considerando l'ultima come riporto uscente alla luce di quanto di quanto visto.



**Nota:** Il valore del riporto uscente ci dà la rappresentabilità o meno su  $n$  cifre.

**Nota:** attenzione al dimensionamento del sommatore: il numero delle cifre degli ingressi e dell'uscita sono per tutti lo stesso.

### Full adder in base 2

Possiamo sintetizzare il circuito sommatore tramite la scomposizione in somme sulle singole cifre con propagazione del riporto.

Il montaggio che si ottiene si chiama **ripple carry** e i suoi componenti sono sommatori ad una cifra (detti ognuno **full adder**).

**In base 2** Si tratta di una rete a 3 ingressi e 2 uscite: sintetizzandola:

X <sub>i</sub>	y <sub>i</sub>	C <sub>in</sub>	S <sub>i</sub>	C <sub>out</sub>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

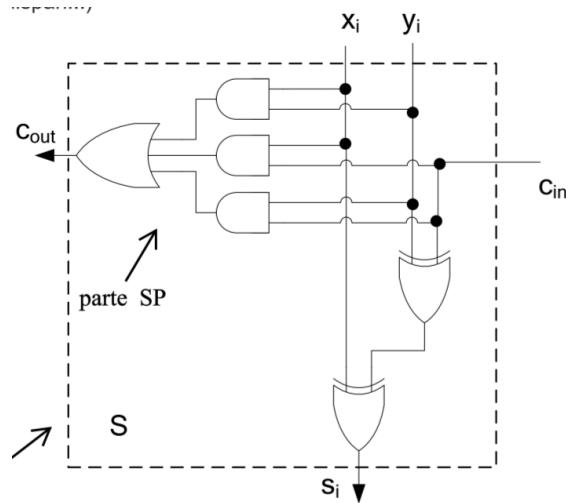
  

C <sub>in</sub>		x <sub>i</sub> y <sub>i</sub>	00	01	11	10
0	0	0	1	0	1	0
1	0	1	0	1	0	0

C <sub>in</sub>		x <sub>i</sub> y <sub>i</sub>	00	01	11	10
0	0	0	0	A	1	0
1	0	B	1	0	1	1

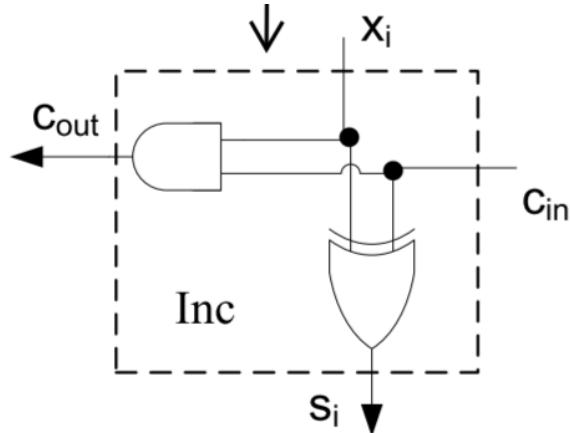
- il riporto uscente si ottiene in forma SP con 3 porte AND a 2 ingressi e una OR a 3 ingressi.
- la somma si osserva che è ad 1 solo se il numero degli ingressi a 1 è dispari: uso quindi XOR in cascata.



### Incrementatore

Circuito sommatorio particolare: lo possiamo vedere come una somma tra un numero dato e 0 con il riporto entrante.

Si ottiene quindi, semplificando, il circuito:



### 3.2.5 Sottrazione

Come per la somma l'algoritmo che conosciamo per la sottrazione non cambia per le altre basi di rappresentazioni (rimanendo in notazione posizionale).

Dati  $X, Y$  naturali in base  $\beta$  su  $n$  cifre con  $0 \leq X, Y < \beta^n - 1$  e dato  $0 \leq b_{in} \leq 1$  prestito entrante, vogliamo calcolare  $Z = X - Y - b_{in}$ .

Per la rappresentabilità

$$-\beta^n \leq X - Y - b_{in} \leq \beta^n - 1$$

quindi il risultato può non essere naturale.

Si dimostra che  $b_{out} \in \{0, 1\}$  indipendentemente dalla base e quindi possiamo scrivere:

$$Z = -b_{out} \cdot \beta^n + D = X - Y - b_{in}$$

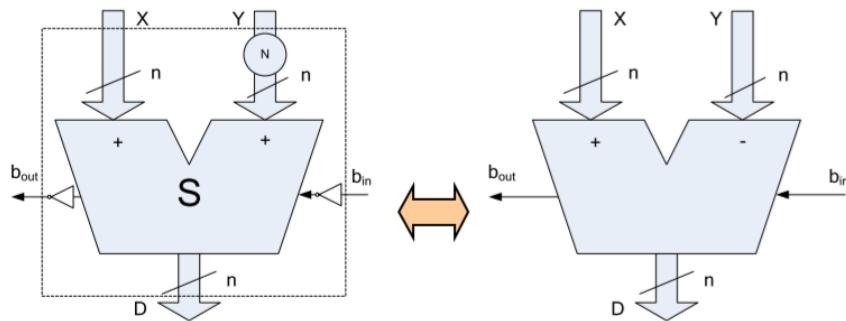
La differenza è rappresentabile su  $n$  cifre se produce un numero naturale, altrimenti si ha il prestito uscente.

Si può ricavare dalle relazioni della differenza la stessa di un sommatore con ingressi

- $X$
- $\bar{Y}$
- $\bar{b}_{in}$

Se il riporto uscente da questa somma è 1, la differenza è un numero naturale pari a  $D$  ed il prestito uscente è ottenuto complementando il riporto uscente: è quindi 0.

Se invece il riporto uscente della somma è 0 la differenza non è un numero naturale ed il prestito uscente, ottenuto complementando, è 1.



## Base 2

**Nota:** anche il sottrattore in base 2 è quindi una rete a 2 livelli di logica, da cui si può ricavare un decrementatore

### Comparazione di numeri naturali

Si realizza con un sottrattore guardando il prestito uscente  $b_{out}$ .

**Nota:** per l'uguaglianza si guarda anche il risultato se uguale a 0

Il comparatore per naturali ( $A < B$ ) che consideriamo da ora in poi sarà dotato di due uscite:

- $flag\_eq$  che vale 1 se sono uguali
- $flag\_min$  vale 1 se  $A < B$

### 3.2.6 Moltiplicazione

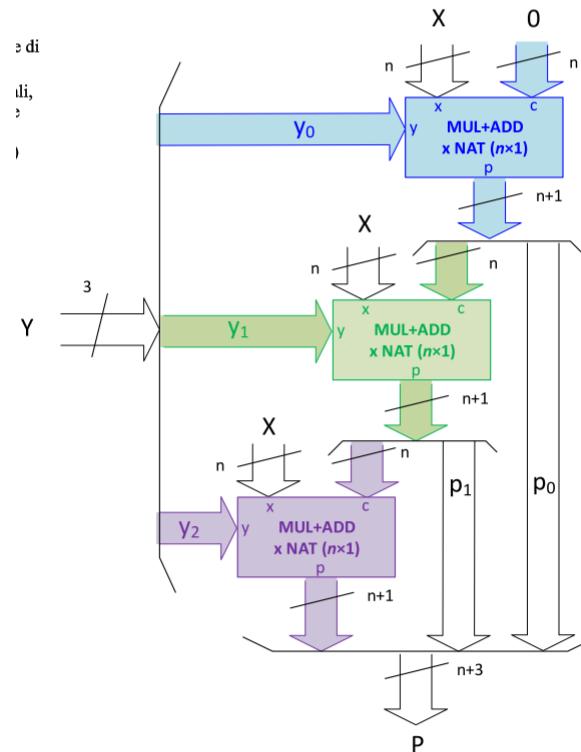
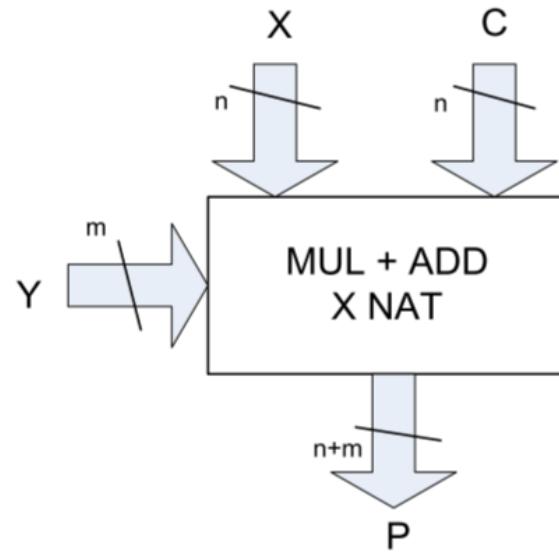
Dati:

- $X, C$  naturali base  $\beta$  su  $n$  cifre
- $Y$  naturali base  $\beta$  su  $m$  cifre

vogliamo calcolare  $P = X \cdot Y + C$

**Nota:** Il termine sommato serve, di nuovo, per rendere possibile la scomponibilità

Il prodotto è rappresentabile su  $n + m$  cifre.



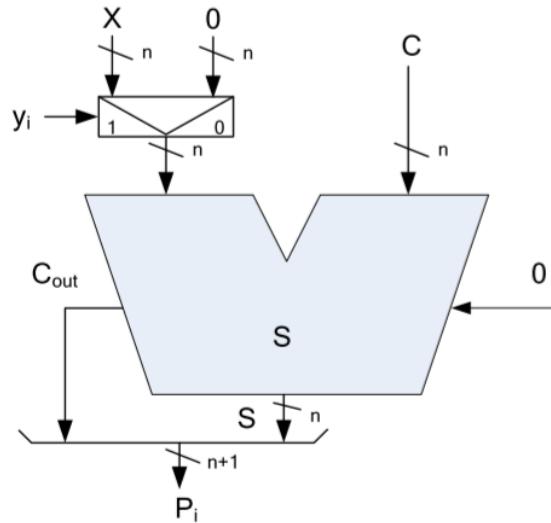
Dall'ultima figura si nota come si possa realizzare la moltiplicazione utilizzando soltanto moltiplicatori con addizionatori ad  $n \times 1$  cifra.

### Moltiplicatore con addizionatore nx1 in base 2

Per sintetizzarlo in base 2 notiamo come il risultato sia:

$$P_i = y_i \cdot X + C = \begin{cases} C & y_i = 0 \\ X + C & y_i = 1 \end{cases}$$

**Nota:** Attenzione al dimensionamento:  $m = 1$



**Nota:** Il multiplexer può essere sostituito da una porta AND con ingressi  $y_i$  (1 cifra) e  $X$  ( $n$  cifre)

### 3.2.7 Divisione

Dati:

- $X$  numero naturale in base  $\beta$  su  $n+m$  cifre (dividendo) tale che  $0 \leq X \leq \beta^{m+n} - 1$
- $Y$  numero naturale in base  $\beta$  su  $m$  cifre (divisore) tale che  $0 \leq Y \leq \beta^m - 1$

vogliamo calcolare  $Q$  e  $R$  tali che:

$$X = Q \cdot Y + R$$

**Nota:** è necessaria una uscita no\_div, per la divisione per 0 e non solo.

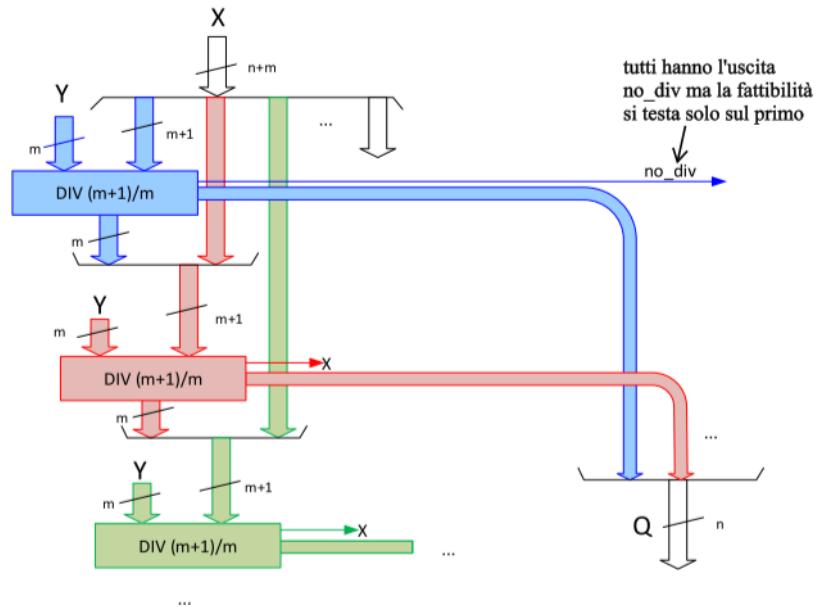
Il resto sta su  $m$  cifre, il quoziente (che sta sicuramente su  $n+m$  cifre) lo vogliamo su  $n$  cifre. Questo implica che:

$$X = Q \cdot Y + R \leq (\beta^{n-1} - 1) \cdot Y + (Y - 1) = \beta^n \cdot Y - 1$$

Abbiamo quindi come ipotesi aggiuntiva, per stare su  $n$  cifre che:

$$X < \beta^n \cdot Y$$

Andando a osservare il processo iterativo per la divisione si nota come il suo nucleo sia una divisione di  $m + 1$  cifre per  $m$  cifre il quale produce un quoziente su 1 cifra e un resto su  $m$  cifre. In termini circuituali:



**Nota:** L'uscita `no_div` è quindi testata con "le  $m$  cifre più significative del dividendo rappresentano un numero più piccolo del divisore"

### Divisore elementare in base 2

Vediamo come sintetizzare un divisore in base 2 che divide un dividendo a  $m + 1$  cifre per un divisore ad  $m$  cifre sotto l'ipotesi che  $X < 2Y$ , che produrrà un quoziente su 1 cifra e un resto su  $m$  cifre.

In particolare il quoziente vale:

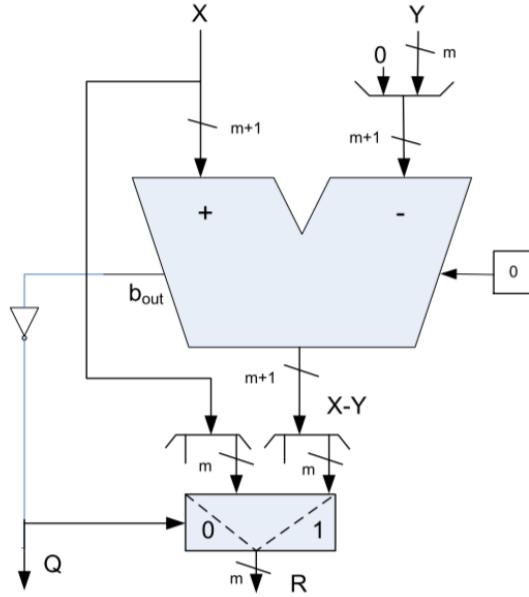
- 0 se il divisore è maggiore del dividendo
- 1 altrimenti

Il resto :

- è uguale al dividendo se questo è minore del divisore
- altrimenti è dato dalla loro differenza

Il che si traduce in

$$Q = \begin{cases} 0 & X < Y \\ 1 & X \geq Y \end{cases}, \quad R = \begin{cases} X & X < Y \\ X - Y & X \geq Y \end{cases}$$



### 3.3 Rappresentazione dei numeri interi

Utilizziamo la rappresentazione in **complemento alla radice**

Facciamo corrispondere alla rappresentazione di un numero naturale (che sappiamo già fare) quella di un numero intero tramite una legge  $L() : \mathbb{Z} \rightarrow \mathbb{N}$  tale per cui dato  $A$  naturale e  $a$  intero:

$$A = L(a), \quad a = L^{-1}(A)$$

**Nota:** la funzione  $L$  che usiamo è quella, come già specificato, del complemento alla radice

L'intervallo di rappresentabilità, dovendo rispettare le proprietà di:

- essere contiguo (privo di "buchi")
- essere più simmetrico possibile rispetto allo 0

sarà dunque:

$$\left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right]$$

#### 3.3.1 Possibili leggi di rappresentazione dei numeri interi

Abbiamo definito gli intervalli di  $L$  come:

$$L_i : \left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right] \rightarrow [0, \beta^n - 1]$$

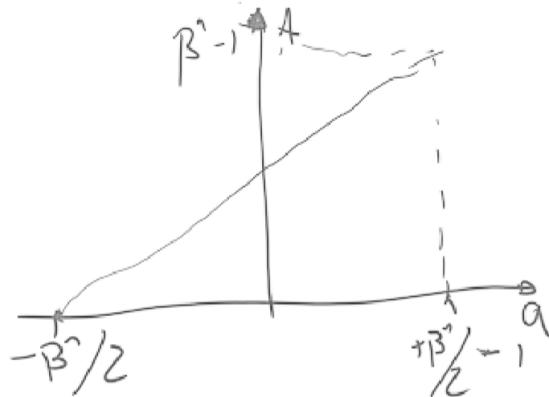
Vediamo quindi le possibili leggi utilizzabili.

## Traslazione

$$L : A = a + \frac{\beta^n}{2}$$

**Nota:** note varie:

- $\frac{\beta^n}{2}$  è detto *fattore di polarizzazione*.
- è *monotona*:  $a < b \Leftrightarrow A < B$



## Complemento alla radice

$$L : A = \begin{cases} a & 0 \leq a < \frac{\beta^n}{2} \\ \beta^n + a & -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

**Nota:** note:

- è quella usata nei calcolatori
- è monotona all'interno di numeri con lo stesso segno

**Modulo e segno** Si fa corrispondere ad un intero una coppia numero naturale e variabile logica (segno).

$$s = \begin{cases} 0 & a \geq 0 \\ 1 & a < 0 \end{cases}, \quad M = \text{abs}(a)$$

### 3.3.2 Proprietà del complemento alla radice

**Determinazione del segno** Possiamo determinare il segno guardandone la rappresentazione:

$$\begin{cases} a \geq 0 \Leftrightarrow 0 \leq A < \frac{\beta^n}{2} \\ a < 0 \Leftrightarrow \frac{\beta^n}{2} \leq A < \beta^n \end{cases}$$

**Nota:** Per guardare se la rappresentazione  $A$  è un numero naturale maggiore o minore di  $\frac{\beta^n}{2}$  basta guardare la cifra più significativa se  $< \frac{\beta}{2}$  oppure se  $\geq \frac{\beta}{2}$  che indica, rispettivamente, numero positivo o numero negativo. Questo si traduce in base 2 con il guardare se questa è 0 o 1.

### Legge inversa

$$L : A = \begin{cases} a & 0 \leq a < \frac{\beta^n}{2} \\ \beta^n + a & -\frac{\beta^n}{2} \leq a < 0 \end{cases} \Leftrightarrow L^{-1} : a = \begin{cases} A & 0 \leq a < \frac{\beta^n}{2} \\ A - \beta^n & \frac{\beta^n}{2} \leq A < \beta^n \end{cases}$$

che semplificata diventa:

$$L^{-1} : a = \begin{cases} A & a_{n-1} \leq \frac{\beta}{2} \\ -(\bar{A} + 1) & a_{n-1} > \frac{\beta}{2} \end{cases}$$

### Forma alternativa per L

$$A = |a|_{\beta^n}$$

se  $-\frac{\beta^n}{2} \leq a < \frac{\beta^n}{2}$

## 3.4 Operazioni su interi in complemento alla radice

È importante sottolineare che i circuiti vedono solo cifre: il legame tra esse e i numeri interi lo conosce solo il programmatore.

### 3.4.1 Valore assoluto

Dato  $a$  (intervallo classico interi) vogliamo  $B = ABS(a) \in [0, \beta^n/2]$  sempre su  $n$  cifre.

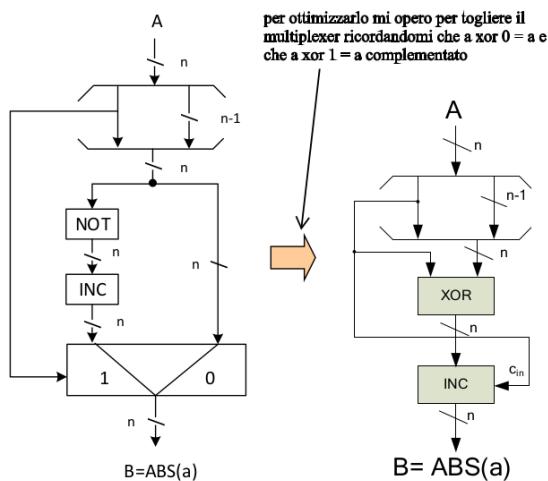
Il circuito che vogliamo realizzare deve seguire l'operazione ABS:

$$ABS(a) = \begin{cases} a & a \geq 0 \\ -a & a < 0 \end{cases}$$

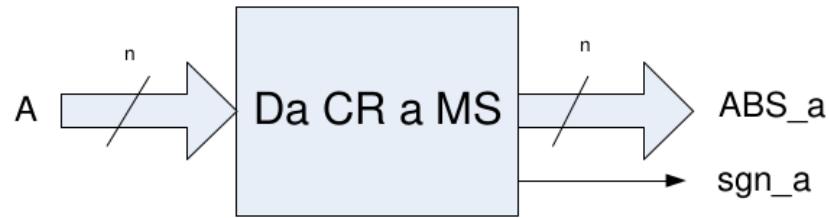
Alla luce di quello che sappiamo fare (vedi prima) dobbiamo realizzare un circuito che produce  $B$  in questo modo:

$$B = ABS(a) = \begin{cases} A & a_{n-1} < \beta/2 \\ \bar{A} + 1 & a_{n-1} \geq \beta/2 \end{cases}$$

In base 2 .

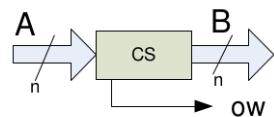


## Circuito di conversione da CR a MS



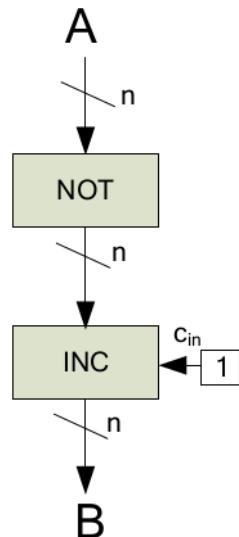
### 3.4.2 Calcolo dell'opposto

Operazione non sempre possibile: i negativi sono 1 in più dei numeri positivi.



dove con *ow* si intende il segnale di overflow che va ad 1 se l'operazione non è possibile.

In base 2 .



Per controllare se c'è *ow* bisogna guardare se A e B hanno entrambi segno negativo: si fa con una AND sul bit più significativo.

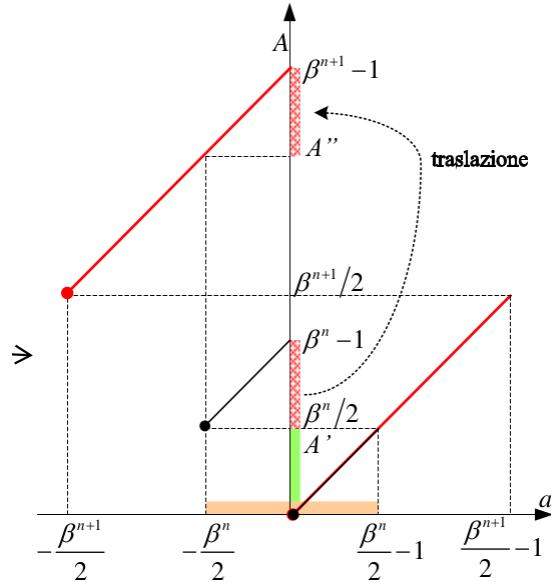
### 3.4.3 Estensione di campo

Sempre fattibile perché l'intervallo su  $n + 1$  cifre contiene quello su  $n$ .

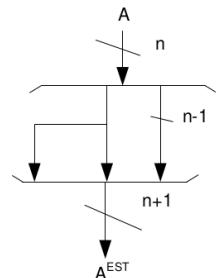
Si realizza seguendo la legge:

$$L'_{n+1} : A_{\text{EST}} = \begin{cases} a & 0 \leq a < \frac{\beta^n}{2} \\ \beta^{n+1} + a & -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

Graficamente:



In base 2 .



ovvero ripetere la cifra più significativa.

#### 3.4.4 Riduzione di campo

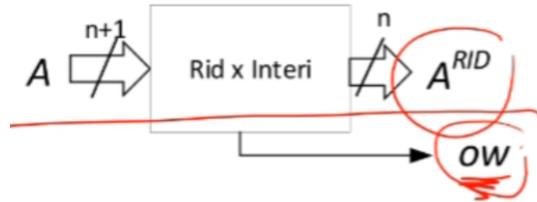
Dato che vogliamo portare un numero su  $n+1$  cifre a uno su  $n$  cifre il problema ha soluzione solo se  $a \in [-\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1] \subseteq [-\frac{\beta^{n+1}}{2}, \frac{\beta^{n+1}}{2} - 1]$  Si ha dunque una condizione di riducibilità:

$$\text{ow} = 0 \leftrightarrow \left( a_n = 0 \wedge a_{n-1} < \frac{\beta}{2} \right) \vee \left( a_n = \beta - 1 \wedge a_{n-1} \geq \frac{\beta}{2} \right)$$

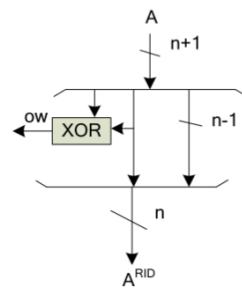
Nel caso in cui un numero sia riducibile la sua rappresentazione su  $n$  cifre è data dalle  $n$  cifre meno significative di  $A$ .

**Nota:** Possiamo anche dire  $A^{RID} = |A|_{\beta^n}$

Si ha quindi il circuito:



**In base 2** si ha il circuito:



**Nota :** Riduzione di  $k$  cifre: *Se devo ridurre di  $k$  cifre:*

- utilizzare  $k$  XOR a 2 ingressi emettere in OR tutte le uscite (non si fa)
- utilizzare una AND e una OR a  $k + 1$  ingressi e controllare con una XOR che le loro due uscite siano identiche (si fa così)

### 3.4.5 Moltiplicazione/Divisione per potenza della base

**Moltiplicazione** Dato  $A$  su  $n$  cifre vogliamo trovare  $B$  su  $n+1$  dato da  $B = \beta \cdot A$  (passando per le rappresentazioni)

$$L : \quad B = \begin{cases} b = \beta \cdot a & 0 \leq a < \frac{\beta^n}{2} \\ \beta^{n+1} + b = \beta^{n+1} + \beta \cdot a = \beta \cdot (\beta^n + a) & -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

ovvero basta mettere uno 0 in coda(?).

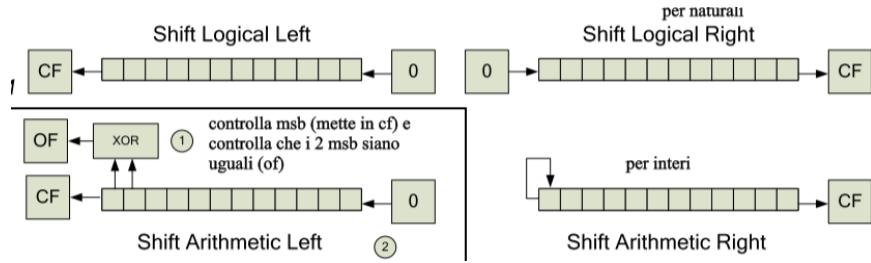
**Divisione** La soluzione è

$$B = \left\lfloor \frac{A}{\beta} \right\rfloor$$

**Nota:** queste due operazioni sono di costo nullo e si fa allo stesso modo che per i naturali

#### Shift logico ed aritmetico

Abbiamo visto che sono identiche a quelle per i naturali quando possiamo estendere e ridurre le cifre di rappresentazione, nel calcolatore però i registri hanno dimensione fissa: dobbiamo avere qualche accortezza:



Notiamo che per la SAL:

1. avendo  $n$  bit bisogna stabilire se il numero di partenza sta su  $n - 1$ , lo si fa tramite il detettore di riducibilità la cui uscita va in OF
2. si fa poi la moltiplicazione come per i naturali

Si nota quindi che basta una sola istruzione di shift sinistro, sta al programmatore discriminare quale flag guardare.

Per la divisione:

1. estendere la rappresentazione ad  $n + 1$  cifre ripetendo la cifra più significativa.
2. buttare via LSB

Si nota che al contrario gli shift destri sono significativamente diversi.

### 3.4.6 Somma

Dati A e B su  $n$  cifre la somma  $s$  sarà compresa tra  $-\beta^n \leq s \leq \beta^n - 2$  quindi potrebbe non essere rappresentabile su  $n$  cifre ma lo è sicuramente su  $n + 1$ . Si ha quindi la necessità di una uscita di ow.

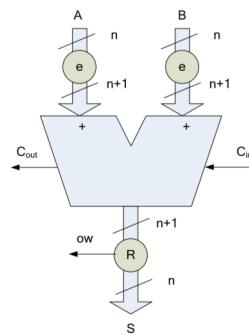
Se  $s$  è rappresentabile su  $n$  cifre si ha:

$$S = |s|_{\beta^n} = |a + b|_{\beta^n} = ||a|_{\beta^n} + |b|_{\beta^n}|_{\beta^n} = |A + B|_{\beta^n}$$

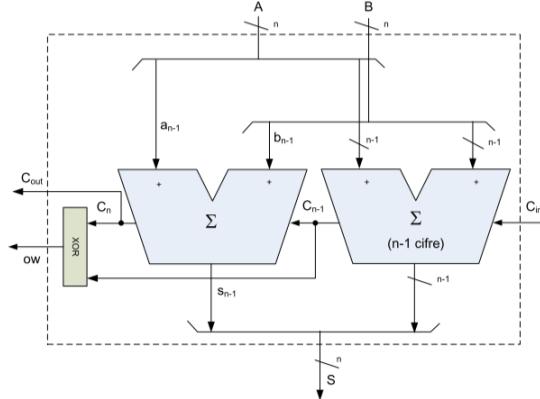
ovvero la rappresentazione della somma è uguale alla somma delle rappresentazioni modulo  $\beta^n$ , il che giustifica usare il complemento alla radice (tanto tutti gli altri moduli si basano sul sommatore).

Dato che l'uscita  $C_{out}$  è significativa solo per somma tra naturali, per gli interi si opera in questo modo:

1. si fa la somma su  $n + 1$  estendendo gli addendi
2. si controlla la riducibilità della somma con un detettore



In base generica si usa quindi uno stadio di sommatore in più, in base 2 il detettore si realizza con uno XOR delle due cifre più significative. Andando a ottimizzare algebricamente si ottiene che basta uno XOR tra i riporti uscenti degli ultimi due full adder.



Risulta quindi essere un sommatore per naturali con una piccola aggiunta (di costo praticamente nullo): non esiste quindi un sommatore per interi e uno per naturali (distinti).

### 3.4.7 Sottrazione

Sempre rappresentabile su  $n + 1$ , non sempre su  $n$ . Noi la vogliamo su  $n$ , quindi dati  $A$  e  $B$  vogliamo  $D = A - B$  su  $n$  cifre. Quando  $d$  è rappresentabile su  $n$  cifre abbiamo:

$$D = |d|_{\beta^n} = |a - b|_{\beta^n} = ||a|_{\beta^n} - |b|_{\beta^n}|_{\beta^n} = |A - B|_{\beta^n} = |A + \bar{B} + 1|_{\beta^n}$$

Notiamo come l'ultima espressione non sia altro che l'uscita di un sommatore per naturali, possiamo utilizzare quindi un solo sottrattore per naturali ed interi.

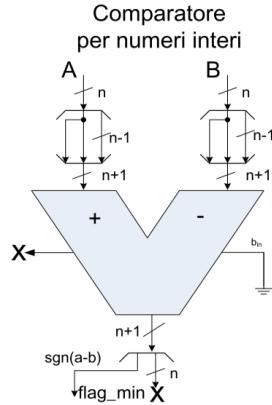
Per testare l'ow:

- in una base generica si fa controllando la riducibilità del risultato della differenza su  $n + 1$  cifre
- per la **base 2** si verifica che *ow* è lo XOR degli ultimi 2 prestiti (**non c'è quindi bisogno di estendere**)

### Comparazione di numeri interi

Si fa sempre con un sottrattore ma non si guarda il prestito uscente bensì il segno della differenza: bisogna quindi estendere gli operandi.

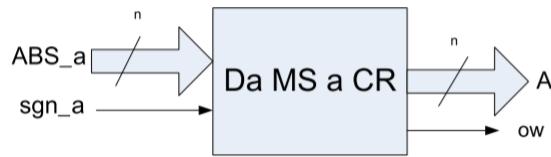
**Circuito per la base 2** Per una base generica si ricava...



### 3.4.8 Moltiplicazione e divisione

Per riutilizzare i circuiti dei naturali si opera sui valori assoluti per poi aggiustare il segno.

#### Circuito di conversione de MS a CR



È presente l'uscita  $\text{ow}$  in quanto su  $n$  cifre l'intervallo dell'ingresso è  $-(\beta^n - 1) \leq a \leq +(\beta^n - 1)$  mentre per l'uscita abbiamo  $-\frac{\beta^n}{2} \leq a \leq \frac{\beta^n}{2} - 1$

Se l'operazione è fattibile si fa così:

$$A = |A|_{\beta^n} = \begin{cases} |\text{ABS}_a|_{\beta^n} & \text{sgn}_a = 0 \\ |-\text{ABS}_a|_{\beta^n} & \text{sgn}_a = 1 \end{cases} = \begin{cases} |\text{ABS}_a|_{\beta^n} & \text{sgn}_a = 0 \\ |\overline{\text{ABS}_a} + 1|_{\beta^n} & \text{sgn}_a = 1 \end{cases}$$

ovvero un multiplexer e un circuito per il calcolo dell'opposto.

Per l' $\text{ow}$ :

$$\text{ow} = 1 \Leftrightarrow (|\text{ABS}_a| > \beta^n/2) \vee (|\text{ABS}_a| = \beta^n/2 \text{ and } \text{sgn}_a = 0)$$

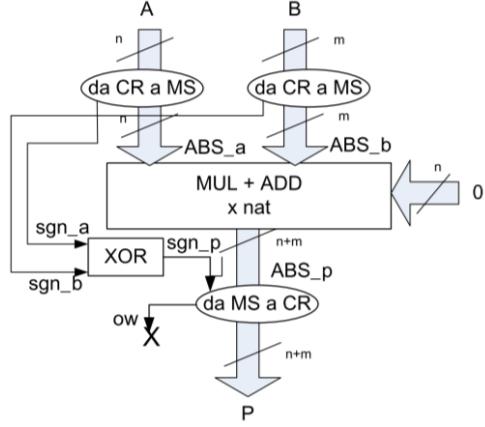
#### Moltiplicazione

Dati A su  $n$  cifre e B su  $m$  vogliamo P su  $n+m$ .

Dato che  $-\beta^{n+m}/4 \leq p \leq \beta^{n+m}/4$  non ci sono problemi di rappresentazione. Si ha quindi che :

$$\text{ABS}(p) = \text{ABS}(A) \cdot \text{ABS}(B)$$

$$\text{sgn}(p) = \text{sgn}(a) \cdot \text{sgn}(b)$$



**Nota:** nel moltiplicatore per interi non esiste un ingresso di somma

### Divisione

Dato A su  $n+m$  e b su  $m$  vogliamo Q su  $n$  e R su  $m$  cifre. Bisogna tener conto che q può non esistere o non essere rappresentabile su  $n$  cifre.

Per l'unicità del risultato non ci bastano le condizioni del Th. del resto e quindi ne aggiungiamo altre:

$$\begin{cases} ABS(r) < ABS(b) \\ sgn(r) = sgn(a) \end{cases}$$

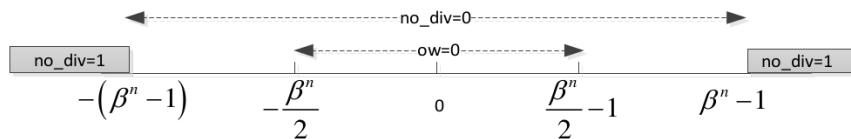
quindi si approssima per troncamento ( $q \cdot b$  è sempre più vicino all'origine di  $a$ ).

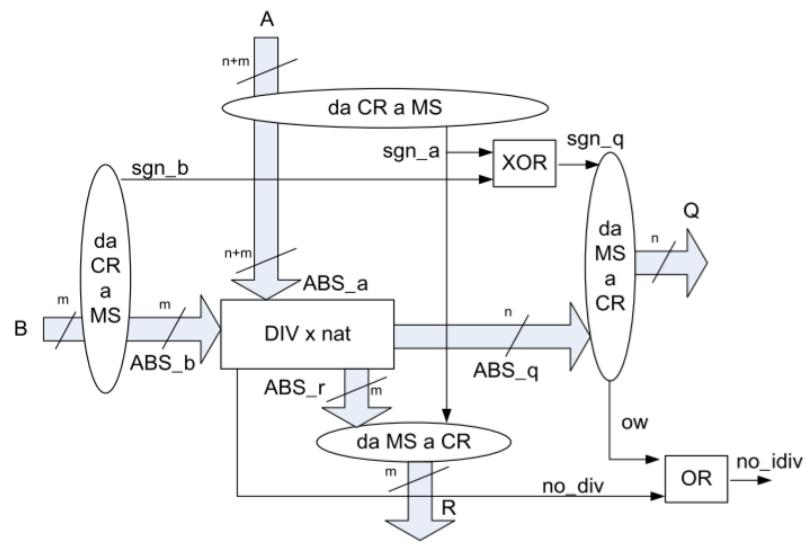
Per la fattibilità, derivando che trattiamo una divisione tra naturali (sui moduli), si ha intanto che:

$$ABS(a) < \beta^n \cdot ABS(b)$$

che copre l'uscita *no\_div* della divisione naturale; inoltre abbiamo il problema dell'ow quando si va a fare la conversione da MS a CR, si ha infatti anche la condizione:

$$-\frac{\beta^n}{2} \leq q \leq \frac{\beta^n}{2} - 1$$



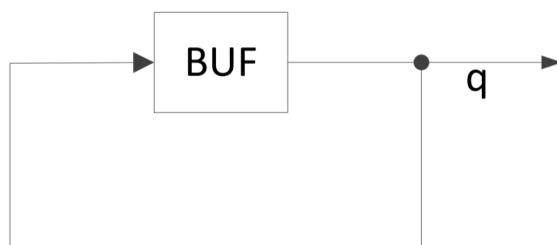


# Capitolo 4

## Reti Sequenziali

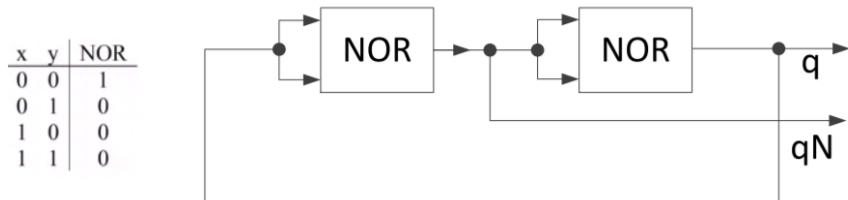
### 4.1 La funzione di memoria e le reti sequenziali asincrone

La memoria si implementa tramite anelli di retroazione, gli stati in cui essi si trovano prendono il nome di  $S_0$  e  $S_1$  che corrispondono allo stato in cui l'uscita si trova a 0 ed 1 rispettivamente.



Il buffer è fondamentale in quanto assegna a  $q$  un valore logico, altrimenti sarebbe un filo staccato.

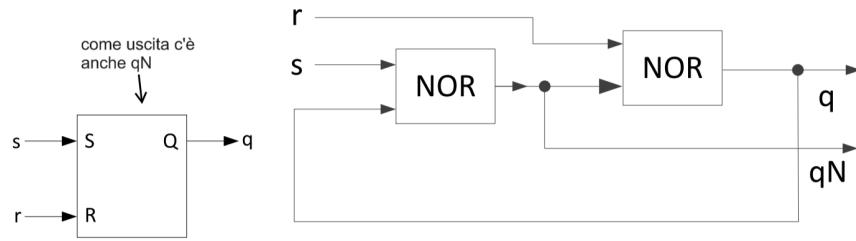
Si può implementare il buffer tramite due NOT implementate a porte NOR.



Per convenzione si dice che il circuito memorizza il bit il cui valore è quello di  $q$ . Se accendendo il circuito  $q$  e  $qN$  sono discordi la rete rimane in quello stato, se invece sono concordi in teoria ciascuna delle due uscite oscilla all'infinito, tuttavia nella realtà la rete si stabilizza velocemente in quanto il tempo di risposta delle porte è diverso (fisicamente impossibile averle identiche) e in quanto tale si verificherà velocemente una situazione in cui  $q$  e  $qN$  sono discordi.

#### 4.1.1 Il latch SR

Per poter impostare il valore di uscita bisogna avere ingressi da pilotare:



Le variabili  $s$  e  $r$  sono attive alte:

- $s=1, r=0$ : La rete si porta nello stato  $S_1$ , ovvero l'uscita si setta
- $s=0, r=1$ : La rete si porta nello stato  $S_0$ , l'uscita si resetta
- $s=0, r=0$ : L'uscita **conserva** il valore che aveva precedentemente
- $s=1, r=1$ : entrambe le uscite valgono 0 e quindi la regola che vuole l'uscita negata come l'opposto di  $q$  è violata: non si ha quindi uno stato di ingresso permesso in corretto pilotaggio.

È una rete asincrona in quanto si aggiorna continuamente.

Per descriverne il comportamento si può usare la tabella di applicazione ( $q$  e  $q'$ ).

**Regole di pilotaggio:** (per reti combinatorie)

1. Pilotaggio in modo fondamentale: cambiare gli ingressi solo quando la rete è a regime
2. Stati di ingresso consecutivi devono essere adiacenti

La regola 1 vale sempre, la regola 2 vale in generale per le RSA, eccetto per il latch-SR che è robusto a pilotaggi scorretti (infatti anche se per un attimo si passa dall'ingresso 11 entrambe le uscite saranno a 0, ma non potendo cambiare anch'esse contemporaneamente non ci saranno problemi).

**Nota:** *non si deve mai passare dagli ingressi 11 a 00: il bit che si va a memorizzare è casuale (il primo che transisce a 0 in base al tempo di risposta delle porte).*

#### 4.1.2 Il problema dello stato iniziale

Come abbiamo visto all'accensione il bit contenuto nel latch SR è casuale, questo però non va bene per tutti gli elementi di memoria (es ok per RAM, non ok per registri).

Si definisce quindi una fase di **reset** in cui inserire in tutti gli elementi di memoria il contenuto iniziale desiderato.

Questo si traduce quindi in due modi:

- un comando che mette a zero l'uscita del latch SR
- la fase di ritorno ad una condizione iniziale di un sistema di elaborazione

In un sistema di elaborazione infatti il reset viene comandato da un pulsante che attiva la carica/scarica di un condensatore la cui tensione viene tradotta tramite un trigger di Shmitt in una uscita 0 o 1.

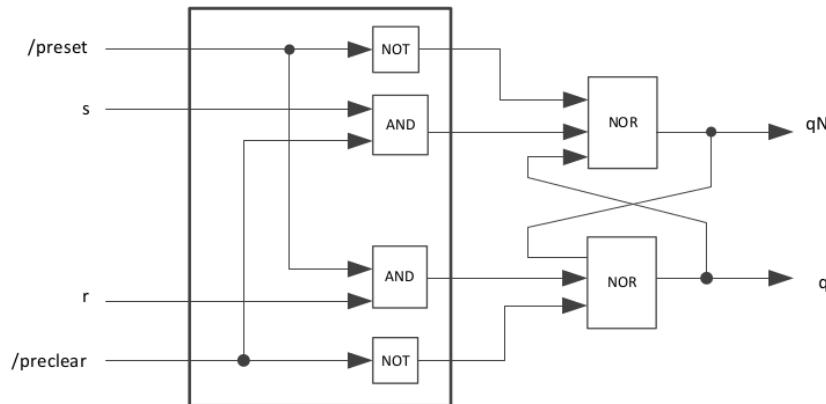
La variabile generata è chiamata */reset* ed è attiva bassa. Si utilizza per comandare l'inizializzazione degli elementi di memoria: quando */reset* è a 0 l'elemento si porta, indipendentemente dal valore degli ingressi, nel suo stato iniziale desiderato; quando è a 1 l'elemento si comporta normalmente.

Per poter applicare questi principi è necessario aggiungere 2 ingressi detti **/preset** e **/preclear**:

- se entrambi a 1 la rete si comporta come un latch SR
- se */preset* = 0 la rete si porta nello stato S1 (indipendentemente dai valori di *s* e *r*)
- se */preclear* = 0 la rete si porta nello stato S0 ("")
- non sono mai contemporaneamente a 0

Se si vuole quindi inizializzare a 1 l'elemento di memoria si connette */preset* a */reset* e */preclear* a Vcc. Al contrario se si vuole inizializzare a 0 l'elemento di memoria si connette */preclear* a */reset* e */preset* a Vcc.

Per implementare il latch SR in questo modo gli si antepone una rete combinatoria che una volta ottimizzata diventa:



#### 4.1.3 Tabelle di flusso e grafi di flusso

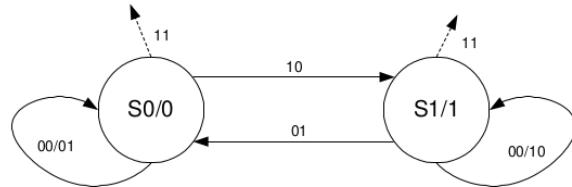
Le RSA si descrivono usando tabelle di flusso o grafi di flusso.

##### Tabelle di flusso Esempio

		sr	00	01	11	10	
		s0	(S0)	(S0)	-	S1	q
		s1	(S1)	S0	-	(S1)	0
							1

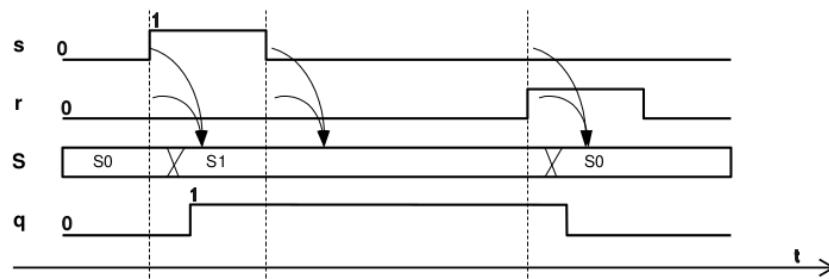
**Nota:** i cerchi indicano che la rete ha raggiunto la stabilità (è a regime) o che quello stato interno è stabile con quello stato di ingresso.

## Grafi di flusso Esempio



**Nota:** La stabilità di un certo stato interno in questo caso è indicata dagli archi che ricadono sullo stesso nodo

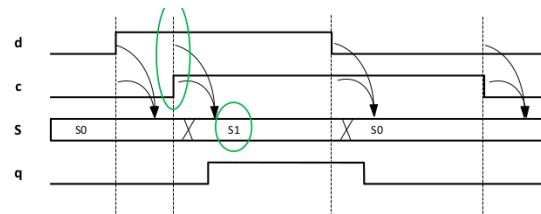
**Diagramma di temporizzazione** Fa vedere i cambiamenti dello stato interno al variare degli stati di ingresso.



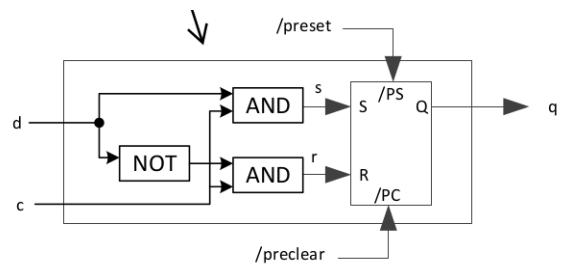
### 4.1.4 Il D-latch trasparente

RSA con due ingressi ( $d$  data e  $c$  control) descritta come: "il D-latch memorizza l'ingresso  $d$  (1 bit) quando  $c$  vale 1 (trasparenza). Quando  $c$  vale 0 invece è in conservazione, cioè mantiene in uscita (memorizza) l'ultimo valore che  $d$  ha assunto quando  $c$  valeva 1."

Esempio di temporizzazione:



Una sintesi del Dlatch può essere ottenuta facilmente a partire da quella di un latch SR anteponendogli una rete combinatoria:



Le **regole di pilotaggio** di questa rete stabiliscono che si debba tenere  $d$  costante a cavallo della transizione di  $c$  da 1 a 0 per i tempi  $T_{setup}$  e  $T_{hold}$ . Questo garantisce che la rete non veda transizioni multiple di ingresso e che di conseguenza si stabilizzi in modo prevedibile.

**Trasparenza** Con questo termine si indica il fatto che, come vediamo nell'esempio del D-latch, l'ingresso è direttamente connesso (dal punto di vista logico) all'uscita.

**Definizione 14:** Il D-Latch è una rete trasparente, cioè la sua uscita cambia mentre la rete è sensibile alle variazioni di ingresso.  $\square$

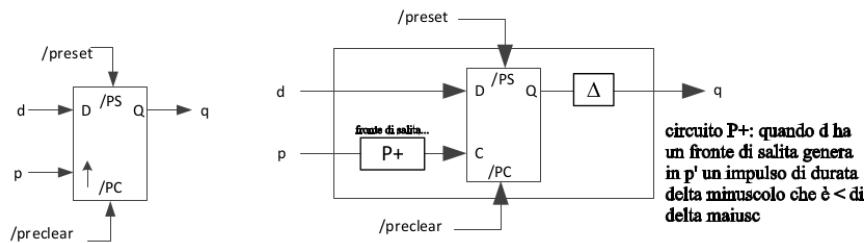
Non si può quindi memorizzare, in un d-latch, niente che sia funzione dell'uscita  $q$  in quanto potrebbero verificarsi problemi di pilotaggio (balla indefinite).

**Nota:** Sono trasparenti tutte le reti viste finora.

#### 4.1.5 Il D flip-flop

Rete non trasparente.

Noto anche come "positive edge-triggered D flip-flop" è una rete con 2 variabili di ingresso che si comporta così: "quando  $p$  ha un fronte di salita memorizza  $d$ , aspetta un po' e adegua l'uscita".



**Nota:** Per avere la non trasparenza è fondamentale che  $\Delta$  sia maggiore di  $P+$ : così facendo l'uscita viene adeguata al valore campionato **dopo** che la rete ha smesso di essere sensibile al valore di  $d$ .

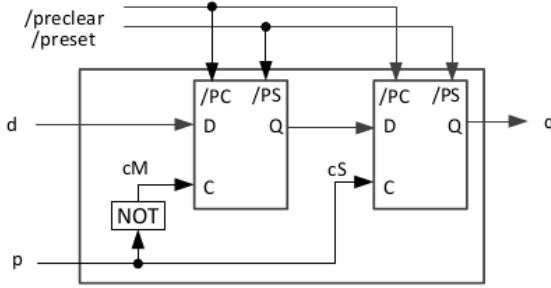
#### Regole di pilotaggio

- a cavallo del fronte di salita di  $p$  la variabile  $d$  deve essere costante per  $T_{setup}$  e  $T_{hold}$ .
- tra due transizioni in salita di  $p$  deve passare abbastanza tempo perché l'uscita si possa adeguare:  $T_{prop}$

**Nota:** quest'ultimo deve essere maggiore di  $T_{hold}$  per garantire la non trasparenza.

**Nota:** L'uscita del D-FF non oscilla mai, posso montarlo in qualsiasi modo: posso mettere in ingresso a  $d$  qualunque funzione dell'uscita  $q$ .

**Montaggio master-slave** Sintesi del D-FF tramite due D-latch in cascata



Quando:

- $p = 0$ , il master campiona e lo slave conserva
- $p = 1$ , il master conserva e lo slave campiona

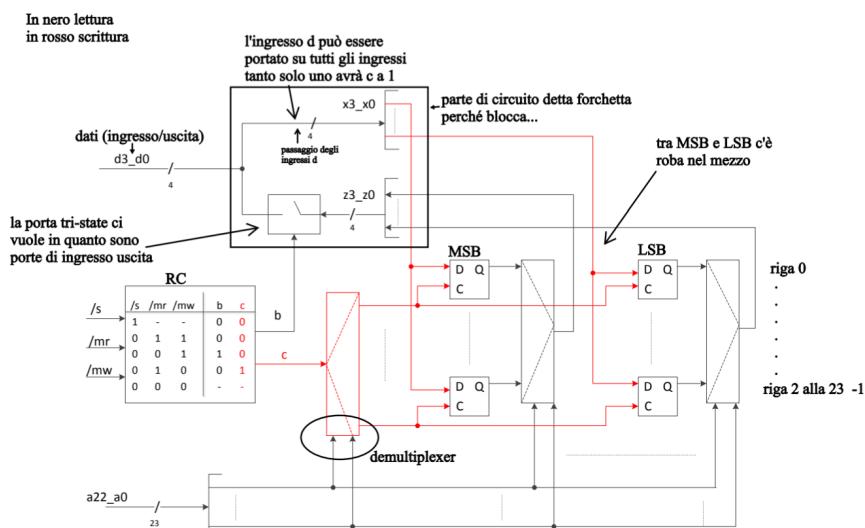
#### 4.1.6 Le memorie RAM statiche (cache)

Sono batterie di D-latch montati a matrice: una riga costituisce una locazione di memoria che può essere letta o scritta (mai entrambe simultaneamente).

Lato utente è composta da:

- fili di indirizzo
- fili di dati
- $/mr$  e  $/mw$
- $/s$

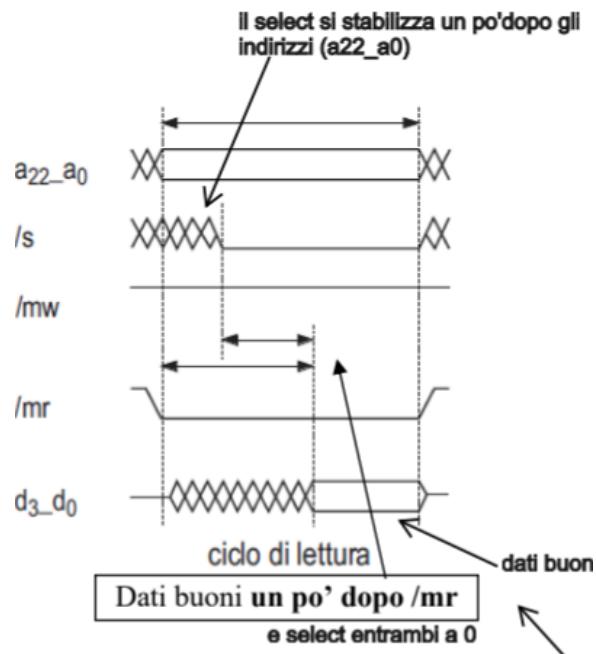
**Nota:** dati questi 3 ultimi ingressi non è permesso il loro pilotaggio "000"



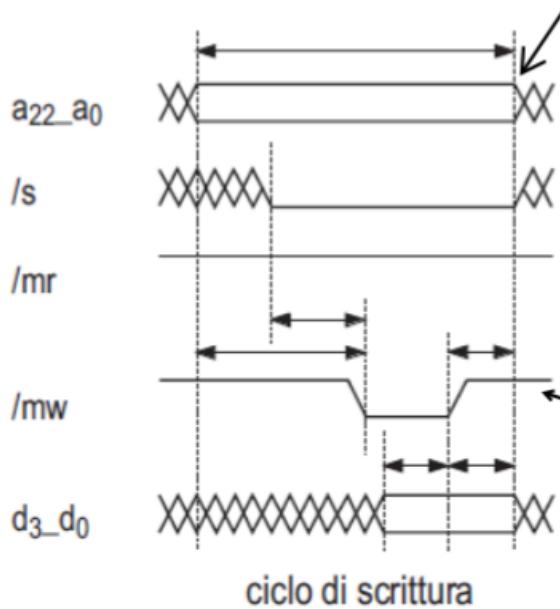
**Nota:** avendo pochi livelli di logica il loro tempo di attraversamento è molto veloce

## Ciclo di lettura

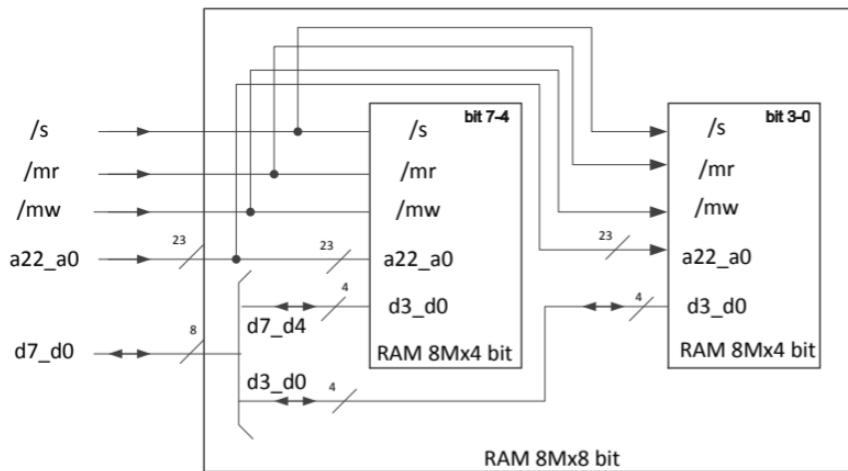
- gli indirizzi si stabilizzano
- allo stesso tempo arriva il comando di  $/mr$
- dopo poco arriva il *select*
- dopo poco vanno in conduzione le tri-state e i multiplexer sulle uscite vanno a regime, da questo momento i **dati** sono **buoni** e possono essere prelevati
- quando  $/mr$  viene tirato su i dati tornano in alta impedenza e il resto può ballare a piacimento.



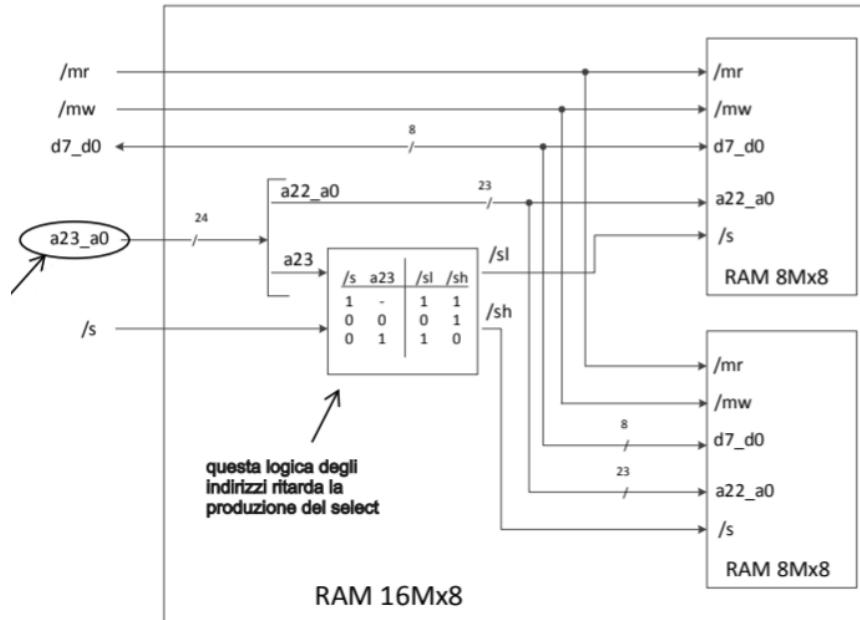
**Ciclo di scrittura** Differenze dovute al fatto che è distruttiva: devo attendere che select e indirizzi siano stabili prima di abbassare  $/mw$ , i dati possono ballare quanto vogliono tranne a cavallo del fronte di salita di  $/mw$



Montaggio in parallelo: raddoppio della capacità di ogni cella



## Montaggio in serie: raddoppio del numero di locazioni



### Collegamento a bus e maschere

Spiega brevemente come funzionano le maschere.

#### 4.1.7 Le memorie Read-only

Le memorie ROM sono circuiti combinatori: ogni locazione contiene infatti valori costanti inseriti in modo indelebile e dipendente dalla tecnologia.

Sono montate insieme alle RAM nello spazio di memoria costituendone la parte non volatile.

Possono essere descritte per semplificazione delle RAM togliendone la parte di scrittura:

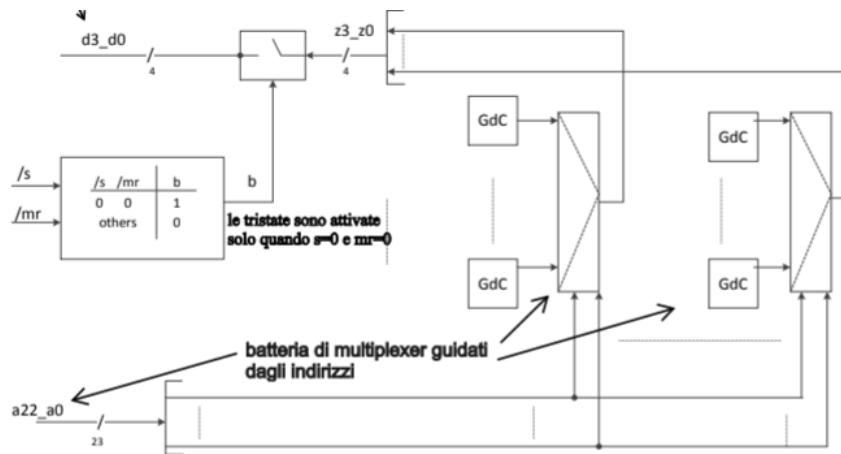


Figura 4.1: GdC = generatori di costante

A seconda della programmabilità dei generatori di costante si distinguono PROM, EPROM, EEPROM.

**Nota:** la programmabilità non può avvenire durante il funzionamento altrimenti si parlerebbe di RAM.

**Nota:** essendo reti combinatorie posso realizzarle facendo riferimento al modello strutturale universale.

### ROM programmabili

- PROM = one time programmable rom (la matrice di connessione è fatta da fusibili)
- EPROM = erasable programmable rom: programmabilità per via elettrica e cancellazione via ultravioletta. (programmabilità ristretta a un certo numero di volte)
- EEPROM = Electrically Erasable Programmable ROM: possono essere riprogrammate *on-chip* un numero di volte comunque "limitato".

## 4.2 Il linguaggio verilog

È un HDL ovvero un hardware description language.

## 4.3 Reti Sequenziali Sincronizzate

Si evolvono soltanto in corrispondenza di istanti temporali ben precisi detti "di sincronizzazione".

**Nota:** non sono più i cambiamenti di ingresso che fanno evolvere la rete.

Normalmente l'evento di sincronizzazione è il fronte di salita del clock.

### 4.3.1 I registri

**Definizione 15:** Un registro a  $W$  bit è una collezione di  $W$  D-FF che hanno:

- ingressi  $d_i$  e uscite  $q_i$  separati
- ingresso  $p$  comune

□

**Requisito di pilotaggio** Gli ingressi  $d$  devono essere stabili intorno al fronte di salita del clock per  $T_{setup}$  e  $T_{hold}$

**Nota:** come sappiamo l'uscita cambia dopo  $T_{prop}$

**Nota:** nei registri è irrilevante l'adiacenza o meno degli ingressi

### 4.3.2 Prima definizione e temporizzazione di una RSS

Una RSS è in approssimazione una collezione di registri e RC montati in qualunque modo purché non ci siano anelli combinatori e purché i registri abbiano lo stesso clock.

**Teorema 7 Unica regola di pilotaggio:** Detto  $t_i$  l'i-esimo fronte di salita del clock, lo statodi ingresso ai registri deve essere stabile in:

$$[t_i - T_{\text{setup}}, t_i + T_{\text{hold}}]$$

per ogni  $i$ . □

Questo ci porta a non poter fare il clock veloce quanto ci pare; definiamo quindi i ritardi:

- $T_{\text{in\_to\_reg}}$ : il tempo di attraversamento della più lunga catena fatta di sole reti combinatorie che si trovi tra un piedino di ingresso e all'ingresso di un registro
- $T_{\text{reg\_to\_reg}}$ : " tra l'uscita di un registro e l'ingresso di un registro
- $T_{\text{in\_to\_out}}$ : " tra un piedino di ingresso e un piedino di uscita
- $T_{\text{reg\_to\_out}}$ : " tra l'uscita di un registro e un piedino di uscita

Si hanno poi 3 vincoli temporali:

1. ingressi costanti in  $[t_i - T_{\text{setup}}, t_i + T_{\text{hold}}]$
2. vicolo di pilotaggio in ingresso: tempo  $T_{\text{a\_monte}}$  per poter cambiare gli ingressi per chi pilota
3. vicolo di pilotaggio in uscita: tempo  $T_{\text{a\_valle}}$  per poter usare le uscite per chi le usa

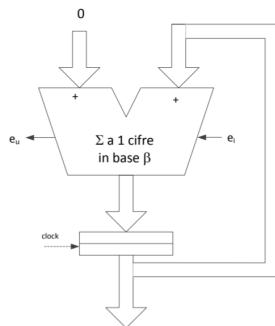
Possiamo quindi dimensionare il clock tenendo conto di questi vincoli:

$$\begin{aligned} T &\geq T_{\text{hold}} + T_{\text{a\_monte}} + T_{\text{in\_to\_reg}} + T_{\text{setup}} \\ T &\geq T_{\text{prop}} + T_{\text{reg\_to\_reg}} + T_{\text{setup}} \\ T &\geq T_{\text{hold}} + T_{\text{a\_monte}} + T_{\text{in\_to\_out}} + T_{\text{a\_valle}} \\ T &\geq T_{\text{prop}} + T_{\text{reg\_to\_out}} + T_{\text{a\_valle}} \end{aligned}$$

**Nota:** ci sarebbero inoltre  $T_{\text{sfas}}$ , massimo sfasamento tra due clock, e  $T_{\text{reg}}$ , sfasamento tra i flip flop del registro, ma li trascuriamo in quanto: il primo si applica a tutte le disequazioni mentre il secondo è piccolo e va implicito nei  $T_{\text{prop}}$ .

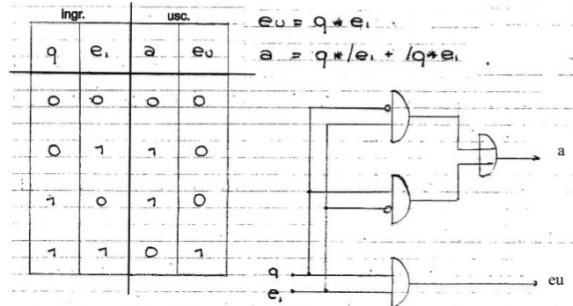
#### 4.3.3 Contatori

RSS il cui stato di uscita è un numero naturale ad  $n$  cifre in base  $\beta$ .



**Nota:** contatori a  $n$  cifre possono essere scomposti in  $n$  contatori a 1 cifra connessi in ripple carry

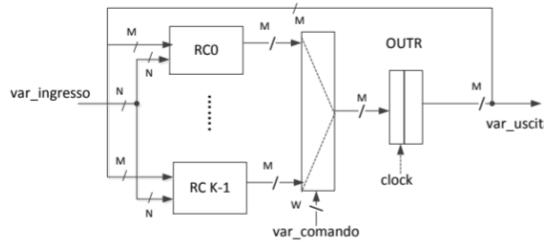
In base 2 la sintesi che si ottiene è:



**I contatori dividono in frequenza** Si può usare il bit più significativo dell'uscita del contatore per costruire un clock più lento.

#### 4.3.4 Registri multifunzionali

Rete che all'arrivo del clock memorizza nel registro una tra  $K$  funzioni combinatorie possibili



**Nota:** esempio: registro caricamento/traslazione: a seconda della variabile di comando  $b$ :

- carica un nuovo valore
- trasla a sx il proprio contenuto

#### 4.3.5 Modello di moore

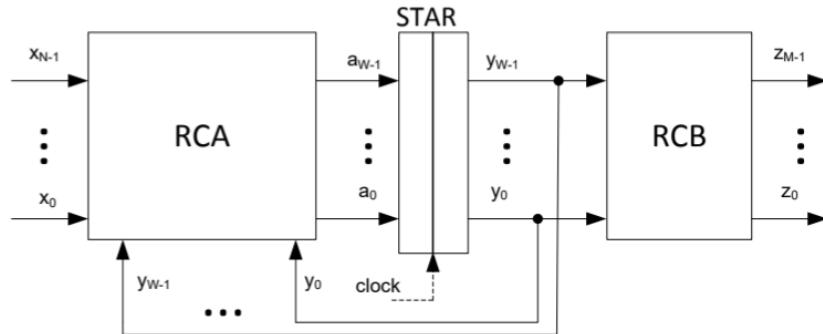
dati:

- $N$  variabili di ingresso
- $M$  variabili di uscita
- meccanismo di marcatura dello stato interno  $S$
- legge di evoluzione nel tempo  $A : X \times S \rightarrow S$
- legge di evoluzione nel tempo  $B : S \rightarrow Z$

Legge di temporizzazione: all'arrivo del clock:

$$S = S' = A(S, X)$$

mentre continuamente  $Z = B(S)$ .



Per il dimensionamento del clock devo tener presente:

$$\begin{aligned} T &\geq T_{hold} + T_{a\_monte} + T_A + T_{setup} \\ T &\geq T_{prop} + T_A + T_{setup} \\ T &\geq T_{prop} + T_Z + T_{a\_valle} \end{aligned}$$

La seconda la possiamo considerare implicata dalla prima.

### Equazioni di uscita e stato

$$S[t_{i+1}] = A(X[t_i], S[t_i])$$

$$Z[t_i] = B(S[t_i])$$

### Flip flop JK

RSS a due ingressi e 1 uscita, all'arrivo del clock si comporta come segue:

$jk$	Azione in uscita
00	Conserva
10	Setta
01	Resetta
11	Commuta

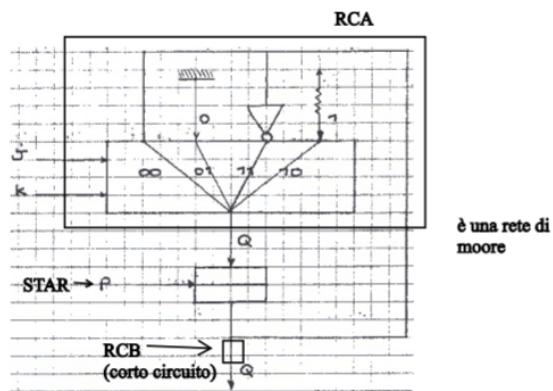


Figura 4.2: È una rete di moore

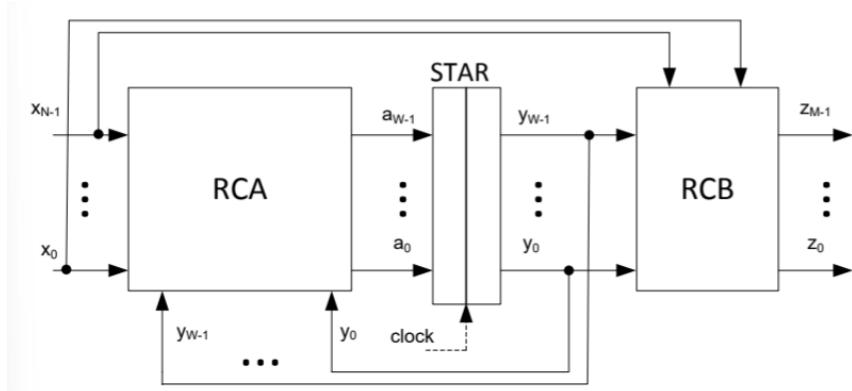
**Nota:** può essere visto come un registro multifunzionale a 1 bit

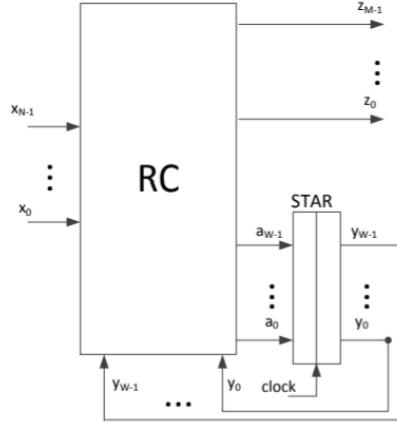
**Nota:** nelle RSS quando si realizza la tabella di flusso non ha senso chiedere gli stati e parlare di stabilità in quanto quest'ultimo concetto è legato alla presenza di anelli combinatori.

#### 4.3.6 Modello di Mealy

A differenza del modello di moore la legge B è:

$$B : X \times S \rightarrow Z$$





Dimensionamento del clock:

$$T \geq T_{hold} + T_{a\_monte} + T_{RC} + T_{setup}$$

$$T \geq T_{prop} + T_{RC} + T_{setup}$$

$$T \geq T_{hold} + T_{a\_monte} + T_{RC} + T_{a\_valle}$$

$$T \geq T_{prop} + T_{RC} + T_{a\_valle}$$

La seconda è implicata dalla prima, la quarta dalla terza, alla fine quindi la più vincolante è la terza. Sommando sia il tempo a monte che a valle, mai insieme nel modello di moore, concludiamo che le reti di mealy hanno un clock più lento.

### Equazioni di uscita e stato

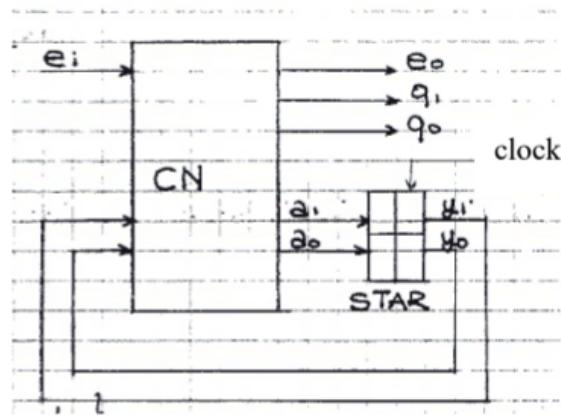
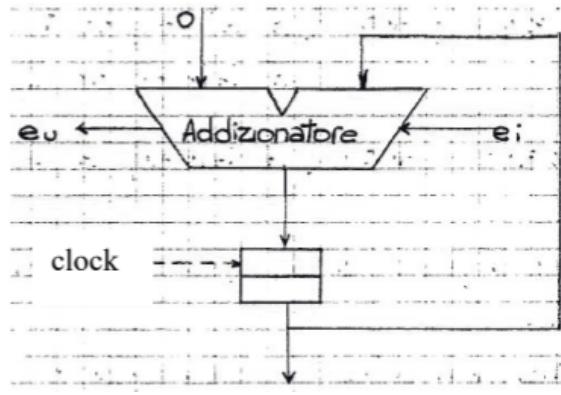
$$S[t_{i+1}] = A(X[t_i], S[t_i])$$

$$Z[t_i] = B(X[t_i], S[t_i])$$

**Nota:** essendo la legge  $B$  più flessibile di solito si riescono a risolvere i soliti problemi in un numero minore di stati, ad esempio un riconoscitore di sequenza si può fare in  $n$  stati (quando con moore ce ne volevano  $n + 1$ )

### Esempio: sintesi del contatore espandibile in base 3

È una rete di mealy: il riporto uscente è funzione combinatoria dello stato interno e del riporto entrante (che è anche ingresso della rete)



### Note finali

**Nota:** Qualunque problema sia risolvibile con un modello (mealy o moore) è risolvibile anche con l'altro.

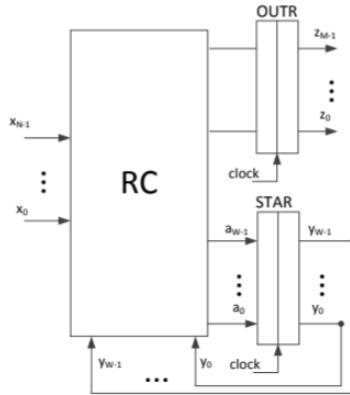
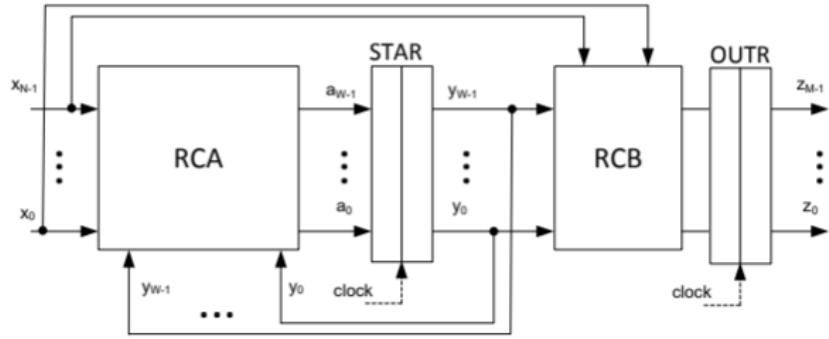
**Nota:** Mealy ha una velocità di risposta maggiore ma clock più lento.

**Nota:** Mealy è **trasparente**.

#### 4.3.7 Modello di Mealy ritardato

Si mette in uscita alla rete di mealy il registro OUTR, in questo modo le uscite:

- variano sempre all'arrivo del clock
- variano in maniera netta (non oscillano)
- rimangono stabili per un clock
- sono **non trasparenti**



Temporizzazione del clock:

$$T \geq T_{hold} + T_{a\_monte} + T_{RC} + T_{setup}$$

$$T \geq T_{prop} + T_{RC} + T_{setup}$$

$$T \geq T_{prop} + T_{a\_valle}$$

La più vincolante è la prima (che implica anche la seconda)

### Equazioni di uscita e stato

$$S[t_{i+1}] = A(X[t_i], S[t_i])$$

$$Z[t_{i+1}] = B(X[t_i], S[t_i])$$

### Note

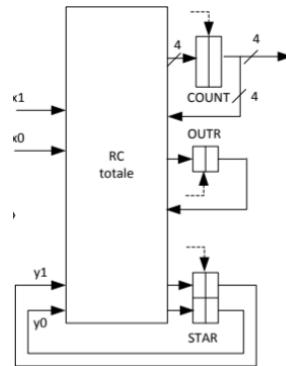
**Nota:** reti di Mealy ritardato sono **non** trasparenti.

## 4.4 Descrizione e sintesi di reti sequenziali sincronizzate complesse

Prendiamo come punto di partenza il modello di Mealy ritardato.

#### 4.4.1 Linguaggio di trasferimento tra registri

Partendo come esempio da una rete che conta modulo 16 sequenze di 3 coppie di bit ricevuti in ingresso si arriva a un modello del tipo:



che vediamo non essere un modello di mealy ritardato perché ha più di due registri.

Si conclude quindi un modello che ha:

- un registro STAR
- quanti registri operativi vogliamo
- posso usare il contenuto dei registri operativi per fornire ingresso a RC (cosa in più rispetto al mealy ritardato)
- le uscite sono tutte sostenute da registri

Note riguardo ai verilog di queste:

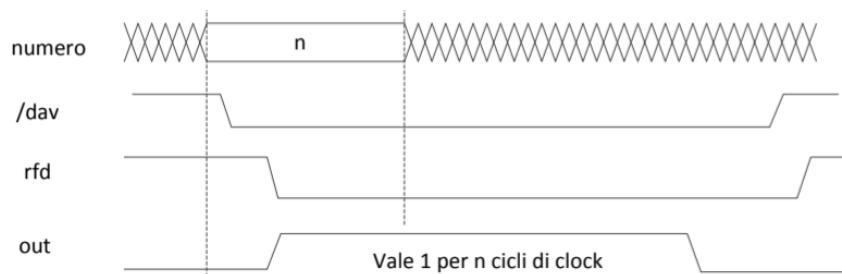
- gli assegnamenti ai registri operativi si chiamano  $\mu$  – *istruzioni*
- gli assegnamenti al registro STAR si chiamano  $\mu$  – *salti*

**Nota:** i vincoli di temporizzazione sono gli stessi del mealy ritardato

#### Esempio: formattore di impulsi con handshake /dav-rfd

Abbiamo due RSS mutuamente asincrone: vogliamo sincronizzarle tra di loro. Aggiungiamo due fili:

- /dav: attivo basso, sta per *data valid*
- rfd: attivo alto, sta per *ready for data*



- al reset **entrambe** le linee sono a **1**
- la prima mossa la fa il produttore che dopo aver preparato il dato e averlo messo sull'uscita **abbassa /dav**
- il consumatore quindi preleva il dato e dunque **abbassa rfd**

**Nota:** una volta abbassato rfd il produttore non è più tenuto a tenere il dato sull'uscita

- il produttore riporta quindi **/dav a 1**
- **dopo** aver visto che dav è andato a 1 il consumatore riporta **rfd a 1**

**Teorema 8:** Quando si hanno cicli di decremento e test, se in uno stato  $S_{init}$  inizializzo un registro a  $k$  ed in  $S_{test}$  lo decremento e per uscire testo se il valore è pari a  $j$  (con  $j \leq k$ ) il numero di cicli nello stato  $S_{test}$  è  $k - j + 1$   $\square$

### Esempio: formatore di impulsi con handshake soc/eoc

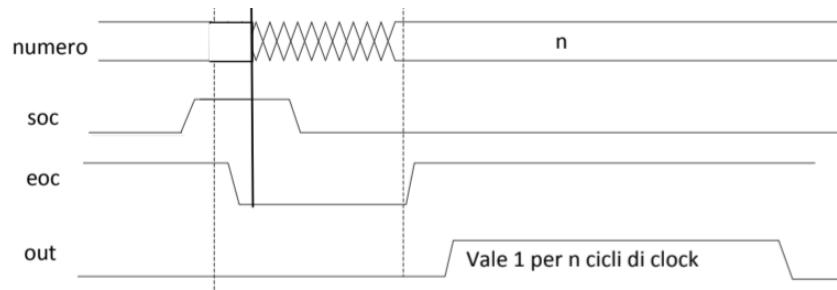
Un altro modo per sincronizzare due reti è utilizzare i fili:

- soc: attivo alto, sta per *start of computation*
- eoc: attivo alto, sta per *end of computation*

**Nota:** questo handshake è tipico dei convertitori analogico/digitali

Si ha:

- a riposo **soc=0** e **eoc=1**
- il consumatore porta **soc a 1** quando vuole un nuovo dato
- il produttore risponde portando **eoc a 0**
- il consumatore riporta **soc a 0** per ackn
- il produttore prepara il nuovo dato, lo mette sull'uscita e dopo riporta **eoc a 1**



#### 4.4.2 Sintesi di RSS complesse - scomposizione in "parte operativa" e "parte di controllo"

# Capitolo 5

## Struttura del calcolatore

Scopo del capitolo: descrizione in verilog di un sistema-calcolatore completo di:

- processore
- memoria
- interfacce
- dispositivi di ingresso uscita

Tutti questi moduli sono messi in comunicazione tramite una rete di interconnessione (bus).

Introduciamo un processore d'esempio: il **sEP8** (8-bit simple Educational Processor) che è in grado di elaborare dati a 8 bit in aritmetica in base 2 rappresentando gli interi in C2, è inoltre in grado di indirizzare una memoria di 16 Mbyte.

Il calcolatore è una serie di RSS (processore e quasi tutte le interfacce) dotati di un piedino */reset*, la RAM invece è una RSA.

### 5.0.1 Visione del calcolatore da parte del programmatore

La memoria appare come uno spazio di  $2^{24}$  locazioni da un byte, per indirizzarle sono necessari indirizzi a 24 bit.

Lo spazio di I/O (=l'insieme dei registri di interfaccia indirizzabili) appare come uno spazio lineare da  $2^{16} = 64K$  locazioni o porte. Per indirizzare una di queste porte il processore dovrà specificarne l'offset dell'interno dello spazio di I/O, tuttavia non necessariamente ad ogni locazione dello spazio di I/O corrisponderà un registro di interfaccia (la maggior parte di questo spazio di indirizzamento non ha controparte fisica).

Il processore sEP8 ha 3 tipi di registri:

- Registri accumulatore: AH e AL, contengono operandi di operazioni.
- Registro dei flag: 8 bit di cui 4 significativi (CF, ZF, SF, OF in ordine dal bit 0 al 3)
- Registri puntatore: 3 a 24 bit perché devono contenere indirizzi di memoria (IP, StackPointer, DataPointer (indirizzo degli operandi))

Affiché il processore parta in stato consistente al reset vanno inizializzati IP ('HFF0000) e F (0).

### 5.0.2 Descrizione del linguaggio macchina del processore sEP8

Per un programmatore assembler di questo processore le istruzioni di esso seguiranno il formato:

OPCODE source, destination

Dove OPCODE è il codice operativo dell'istruzione, source e destination i due operandi (a volte può mancare source a volte entrambi come per NOP e HLT).

Le modalità di indirizzamento sono:

- per le istruzioni operative:
  - di registro: uno o entrambi sono registri
  - immediato: source è una costante
  - di memoria: valido esclusivamente per source o destination e si distingue in:
    - \* diretto: indirizzo specificato direttamente
    - \* indiretto: indirizzo contenuto nel registro DP (utilizzo stile puntatore (DP))
- Per le istruzioni di controllo: nei JMP e in CALL l'indirizzo specificato va a sostituire quello in IP, in RET non va specificato (preleva dalla pila e sostituisce a IP)

Le modalità di indirizzamento sono importanti più del tipo di operazione, infatti il dove procurarsi gli operandi è un discriminante importante per quello che riguarda la fase di fetch, infatti più fetch diversi (memoria, immediato, registri) possono corrispondere poi ad una unica operazione (una MOV è una copia a prescindere da dove sono stati presi gli operandi, il dove vengono presi è diverso).

Ciascuna istruzione macchina è lunga almeno un byte, il primo di essi:

- discrimina il tipo di operazione (opcode)
- e il modo in cui si recuperano gli operandi, detto formato dell'istruzione

Essendo i possibili formati 8 si ha che nel primo byte i primi 3 bit codificano il formato e i restanti 5 l'opcode. Descrizione dei formati

- Formato F) (000): in questa categoria rientrano tutte le istruzioni per le quali il processore non deve compiere nessuna azione per procurarsi gli operandi (sono registri o non ci sono), le istruzioni di questo formato sono quindi di **1 byte**.
- Formato F2 (010): istruzioni in cui source sta in memoria indirizzato in modo indiretto tramite il registro puntatore DP. Anche in questo caso l'istruzione sta su **1 byte**.
- Formato F3 (011): istruzioni in cui destination è indirizzato in modo indiretto utilizzando DP. L'istruzione sta su **1 byte**
- Formato F4 (100): istruzioni in cui source è indirizzato in modo immediato e sta su 8 bit. Istruzione lunga **2 byte** (nel secondo ci sta source). (nota: essendo lunga 2 byte IP va incrementato durante il fetch).
- Formato F5 (101): istruzioni in cui source è indirizzato in modo diretto. Istruzioni lunghe **4 byte** (1 opcode 3 indirizzo).

- Formato F6 (110): destination è in memoria indirizzato in modo diretto. Istruzione di **4 byte**.
- Formato F7 (111): istruzioni in cui ho un indirizzo di salto. Istruzione di **4 byte**.
- Formato F1(001): tutte le istruzioni mancanti: quelle relative allo spazio I/O, per le quali è necessario prelevare in memoria l'indirizzo a 16 bit della porta di I/O sorgente/destinatario, e le istruzioni MOV che hanno come operando uno dei registri a 24 bit DP o SP. Per queste, è necessario quantomeno leggere altri 3 byte dopo il codice operativo, puntati dal registro IP. Siccome i passi da eseguire sono diversi a seconda dell'istruzione, verranno gestiti successivamente nelle fasi di esecuzione individuali (ancorché questa modalità sia poco pulita dal punto di vista concettuale). Per le istruzioni di questo formato, la fase di fetch si limita quindi al prelievo del codice operativo (lettura di **1 byte** in memoria)

### 5.0.3 Architettura del calcolatore

Vediamo la struttura interna. Sulla rete di interconnessione si ha:

- fili di indirizzo: 24, sono uscite per il processore (imposta gli indirizzi dove vuole leggere e scrivere) e ingressi per gli altri. (nello spazio di I/O che è solo di 64k porte alcuni fili vengono buttati)
- fili di dati: 8 perché il processore legge/scrive byte. Devono essere forchettati da tutte le parti (sono pilotati sia dal processore che da altri dispositivi)
- fili di controllo: tutti attivi bassi /mr, /mw, /ior, /iow. Sono uscite per il processore e ingressi per gli altri
- segnale di clock
- fili di interconnessione tra interfacce e dispositivi
- fili di comunicazione tra la memoria video e l'adattatore grafico (non lo analizziamo)

**Nota:** *il reset non è sul bus*

#### Spazio di memoria

16Mbyte, in parte RAM in parte EPROM (64k) in parte memoria video (64k). La EPROM deve essere montata in modo che copra le locazioni tra 'HFF0000 e 'HFFFFFF. La memoria video copre gli indirizzi fisici 'H0A0000–'H0AFFFFF. Il resto è tutto RAM.

Per realizzare questa memoria devo realizzare un montaggio in serie; mi serve quindi un select per i 3 moduli ci vuole quindi della logica combinatoria, pilotata dagli 8 fili più alti, che prende il nome di maschera.

#### Spazio di I/O

È realizzato tramite interfacce per la sua maggior parte. Queste fungono da raccordo tra bus e dispositivi di I/O e quindi ha collegamenti su ambo i lati: quelli lato bus saranno identici a una piccola RAM dove le locazioni prendono il nome di **porte** di ingresso e uscita. Saranno simili le temporizzazioni per i cicli di lettura e scrittura, differiscono infatti per:

- alcune porte supportano **solo** scrittura o lettura.
- se un'interfaccia implementa una sola porta, non sono necessari i fili di indirizzo (basta /s).

Lato dispositivo i collegamenti variano a seconda del caso.

Se non si attaccano direttamente i dispositivi al bus è perché:

- i dispositivi hanno velocità più lente del processore (diversi ordini di grandezza) e diverse tra i vari dispositivi
- i dispositivi hanno modalità di trasferimento molto diverse tra loro (bit o byte)

Le interfacce rendono quindi i dispositivi visibili in modo standard.

## Processore

Contiene i registri:

- STAR
- MJR
- instruction registers (OPCODE, SOURCE, DEST\_ADDR)
- Registri che sostengono le uscite
- Registro DIR: per abilitare le porte tri-state quando il processore deve scrivere sul bus (nello spazio di I/O)
- Registri di appoggio APPx e NUMLOC per i cicli di lettura/scrittura

Al reset alcuni registri vengono inizializzati, seguono poi ciclicamente le fasi di fetch e di esecuzione da cui si esce solo per HLT o per prelievo di istruzione non valida.

Al reset quindi si inizializzano:

- IP e F: 'HFF0000 e 0
- /MR, /MV, /IOR, /IOW: tutti a 1
- DIR a 0 (fili in alta impedenza, a 1 solo quando si scrive)
- STAR con l'etichetta del primo statement della fase di fetch

Fase di fetch, il processore:

- preleva un byte all'indirizzo indicato in IP
- incrementa IP (modulo  $2^{24}$ )
- controlla che il byte abbia un opcode conosciuto (altrimenti si blocca)
- inserisce il byte nel registro OPCODE e valuta il formato, a seconda di esso:
  - si procura un operando source a 8 bit e lo mette in SOURCE (formati F2, F4, F5), a seconda dei formati

- \* accesso in memoria puntato da DP
- \* accesso in memoria puntato da IP (incremento di 3 byte di IP)
- \* due accessi in memoria all'indirizzo puntato da IP ho l'indirizzo del dato (incremento di 3 byte di IP)
- si procura l'indirizzo dell'operando destinatario e lo mette in DEST\_ADDR (formati F3, F6, F7):
  - \* F3: indirizzo già in DP
  - \* F6 e F7: in memoria 3 byte puntati da IP (va incrementato):
    - F0 non deve fare nulla
    - F1 fa cose particolari

Alla fine del fetch si guarda OPCODE.

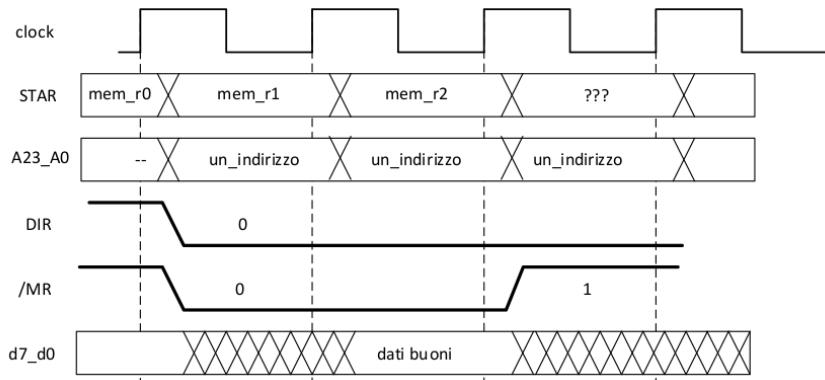
Fase di esecuzione: il processore esegue l'istruzione e torna al fetch.

### Lettura e scrittura in memoria e nello spazio I/O

**Lettura** Nel fetch si leggeva in memoria. In esecuzione il processore deve leggere e scrivere in memoria o nello spazio di I/O.

#### Ciclo di lettura in memoria .

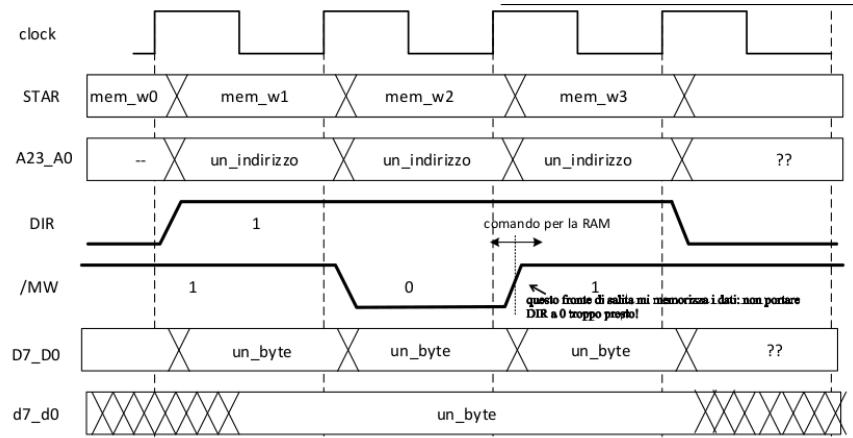
```
mem_r0: begin A23_A0<=un_indirizzo; DIR<=0; MR_<=0; STAR<=mem_r1; end
mem_r1: begin STAR<=mem_r2; end //stato di wait
mem_r2: begin QUALCHE_REGISTRO<=d7_d0; MR_<=1; .....
wait state nel ciclo i dati non sono buoni
```



**Nota:** *DIR non lo posso mettere subito a 1, aspettare un po' dopo /mr=1 (conviene tenere DIR sempre a 0)*

#### Ciclo di scrittura in memoria .

```
mem_w0: begin A23_A0<=un_indirizzo; D7_D0<=un_byte; DIR<=1;
           STAR<=mem_w1; end
mem_w1: begin MW_<=0; STAR<=mem_w2; end
mem_w2: begin MW_<=1; STAR<=mem_w3; end
mem_w3: begin DIR<=0; ....; end
Non posso riportare D
menti i dati non sono st
lita di /mw, come invec
```



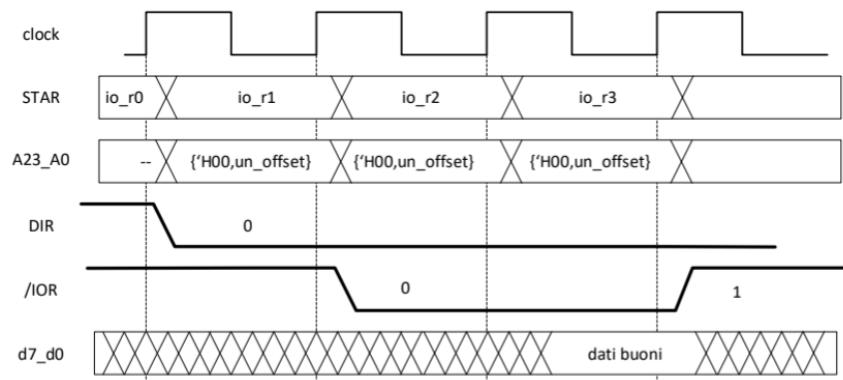
**Nota:** la RAM memorizza il dato sul fronte di salita di /mw e quindi gli indirizzi devono essere stabili intorno ad esso: non posso portare subito DIR a 0 perché i dati andrebbero in alta impedenza al momento della memorizzazione. (stesso per il registro A23\_0 in mem\_w2)

### Ciclo di lettura nello spazio di I/O

**Nota:** Per i cicli di lettura e scrittura nello spazio di I/O ricordarsi che gli indirizzi sono a 16bit e si usano /ior e /iow

Differenza sostanziale: gli indirizzi devono essere pronti un clock prima del comando di lettura (/ior=0) per via del funzionamento delle interfacce (nello spazio di I/O anche le letture possono essere distruttive in quanto la lettura in alcuni casi porta una riscrittura da parte del dispositivo, inoltre leggere un dato può portare il cambiamento del contenuto di un'altra porta: caso delle porte che contengono informazioni del tipo "hai già letto il dato di quest'altra porta" che possono riscontrare inconsistenze se gli indirizzi o i select ballano quando si ha ancora /ior a 0).

```
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 /state di wait
io_r3: begin QUALCHE_REGISTRO=d7_d0; IOR<=1; .... ; end
```



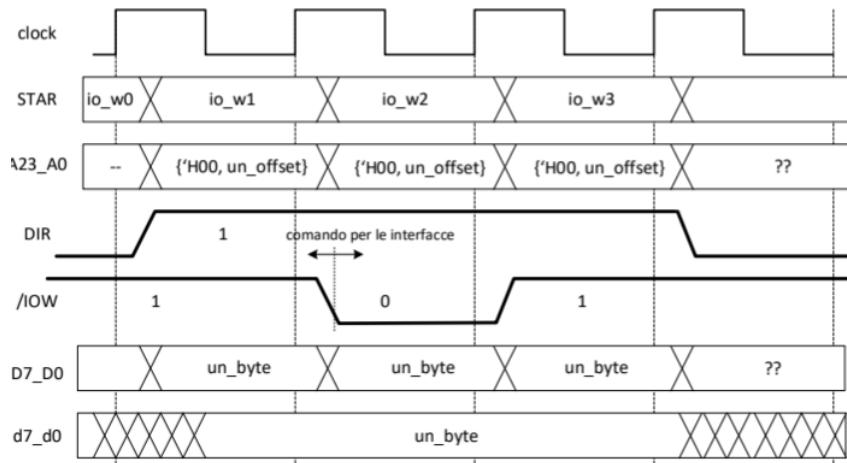
### Ciclo di scrittura nello spazio di I/O .

```

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

```

**Nota:** Differenza dalla memoria, i dati devono essere già pronti sul fronte di discesa di /iow perché molte interfacce memorizzano sul fronte di discesa (in memoria si memorizzava sul fronte si salita)



**Accessi per più di un byte alla memoria** Per fare la accessi a più byte (per prelevare indirizzi nello spazio di I/O e di memoria (2 e 3 byte rispettivamente)) fa comodo dotarsi di micro-sottoprogrammi. Lo facciamo utilizzando il registro MJR e il registro interno NUMLOC come contatore di byte da leggere/scrivere e i registri APP0, ..., APP3 per contenere questi byte.

La descrizione per leggere in memoria è quindi:

```

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

Per leggere un byte contenuto in memoria alla locazione un_indirizzo si scrive:
//MICROPROGRAMMA PRINCIPALE
Sx: begin ... A23_A0<=un_indirizzo; MJR<=Sx+1; STAR<=readB; end
Sx+1: begin ... <utilizzo di APP0> end

```

La descrizione per scrivere in memoria è quindi:

```

// MICROSTOTTOPROGRAMMA PER SCRITTURE IN MEMORIA
writeb: begin D7_D0<=APPO; DIR<=1; NUMLOC<=1; STAR<=write0; end
writeln: begin D7_D0<=APPO; DIR<=1; NUMLOC<=2; STAR<=writeln; end
writem: begin D7_D0<=APPO; DIR<=1; NUMLOC<=3; STAR<=writeln; end
write0: begin D7_D0<=APPO; DIR<=1; NUMLOC<=4; STAR<=writeln; end
write1: begin MW <=0; STAR<=writeln; end
write2: begin D7_D0<=APP1; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
           STAR<=writeln3; end
write3: begin MW <=0; STAR<=writeln4; end
write4: begin MW <=1; STAR<=(NUMLOC==1)?writeln1:writeln5; end
write5: begin D7_D0<=APP2; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
           STAR<=writeln6; end
write6: begin MW <=0; STAR<=writeln7; end
write7: begin MW <=1; STAR<=(NUMLOC==1)?writeln1:writeln8; end
write8: begin D7_D0<=APP3; A23_A0<=A23_A0+1; STAR<= writeln9; end
write9: begin MW <=0; STAR<= writeln10; end
writeln10: begin MW <=1; STAR<= writeln11; end
writeln11: begin DIR<=0; STAR<=MJR; end

E quindi, per scrivere dato_16_bit a partire da un_indirizzo, dovremo scrivere:
Sx: begin ... APP1<=dato_16_bit[15:8]; APP0<=dato_16_bit[7:0];
           A23_A0<=un_indirizzo; MJR<=Sx+1; STAR<=writelnW; end
Sx+1: begin ... end

```

## Descrizione del processore in Verilog

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

```

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

// TERMINAZIONE DELLA FASE DI CHIAMATA
// TERMINAZIONE CON BLOCCO PER ISTRUZIONE NON VALIDA
nvi: begin STAR <= nvi; end
// TERMINAZIONE REGOLARE CON PASSAGGIO ALLA FASE DI ESECUZIONE
fetchEnd: begin MJR <= first_execution_state(OPCODE);
           STAR <= fetchEnd1; end
fetchEnd1: begin STAR <= MJR; end

```

Ricapitolando quindi cosa succede in ciascuno dei formati (in particolare quelli che prevedono operandi in memoria):

		parte comune		parte specifica del formato	
F	Byte	OPCODE	SOURCE	DEST_ADDR	
F0	1	readB @ IP	--	--	
F1	1 ?	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	

**Nota:** All'uscita della fase di fetch:

- OPCODE contiene il codice operativo dell'istruzione
- se l'istruzione ha un operando sorgente in memoria, questo sta in SOURCE
- se l'istruzione ha un operando destinatario in memoria, il suo indirizzo sta in DEST\_ADDR
- IP è stato incrementato il numero di volte necessarie e punta alla prossima istruzione

**Fase di esecuzione** La complessità del fetch è già stata gestita e le istruzioni complesse sono implementate tramite reti combinatorie, si tratta quindi di semplici assegnamenti procedurali.

**Nota:** Per le operazioni logico aritmetiche se ne occupa la ALU (anche per quanto riguarda i flag)

**Nota:** Per tutte le istruzioni di salto la decisione se saltare o meno è presa dalla rete combinatoria `jmp_condition()` che sulla base dell'opcode stabilisce la veridicità della condizione. In caso di salto il nuovo valore di IP è contenuto in DEST\_ADDR (formato F7)

In tutto si hanno 86 statment, quindi STAR e MJR devono essere di 7 bit. Si noti che:

- in ogni stato ci sono micro-salti al massimo a due vie (la maggior parte sono a una sola via)
- altro

## 5.1 Interfacce

Ci soffermiamo su quelle parallele (in grado di trasmettere un byte alla volta), quelle seriali (un bit alla volta) e quelle per la conversione A/D e D/A che trasformano gruppi di bit in tensioni e viceversa.

Essendo per natura tutte identiche dal lato del processore, per distinguerne il tipo è necessario osservarle dal lato del dispositivo.

Per farlo ne diamo una descrizione dal punto di vista funzionale (cioè di chi ci deve interagire): ad esempio una interfaccia di ingresso/uscita ha due registri detti Receive Buffer Register (RBR) e Transmit Buffer Register (TBR). Questi due sono distinti dall'indirizzo

interno a 1 bit che viene portato dal filo *a0* del bus. In particolare RBR contiene i dati scritti dal dispositivo mentre TBR contiene i dati da mandare al dispositivo. Per quanto riguarda i collegamenti sono standard lato bus mentre quelli lato dispositivi dipendono dalla natura di quest'ultimi.

Se il programmatore vuole leggere un dato dovrà eseguire un'istruzione del tipo:

```
IN offset_RBR, %AL
```

Se in vece vuole mandare qualcosa al dispositivo:

```
OUT %AL, offset_TBR
```

Interfacce di questo tipo non permettono però una sincronizzazione tra processore e dispositivo (garantire che tra due IN o due OUT ci sia stato il tempo di elaborare i dati); si introducono quindi dei registri di stato, all'interno delle interfacce, che permettono l'implementazione di un handshake. Questi registri prendono il nome di Receive Status Register (RSR) e Transmit Status Register (TSR) dei quali è significativo solo un bit (vengono infatti spesso collassati in un unico registro RTSR). I bit significativi hanno il nome di **flag di buffer ingresso pieno (FI)** e **flag di buffer uscita vuoto (FO)** e sono gestiti dall'interfaccia che li setta e resetta in base ai rispettivi eventi.

**Nota:** per indirizzare 4 registri ci vogliono 2 fili di indirizzo.

FI è inizialmente a 0 e viene settato a 1 quando il dispositivo scrive in RBR, quando poi il processore legge con una IN tale registro l'interfaccia riporta a 0 il flag.



FO è inizialmente a 1 e viene resettato a 0 quando il processore scrive in TBR (con una OUT), quando il dispositivo legge il registro l'interfaccia riporta FO a 1.



Questa tecnica è detta **accesso a controllo di programma** ed è inefficiente, a calcolatori si vede l'accesso ad interruzione di programma.

### 5.1.1 Interfacce parallele

Le più basiche (1 sola porta) interfacce parallele di ingresso (o di uscita) saranno collegate con:

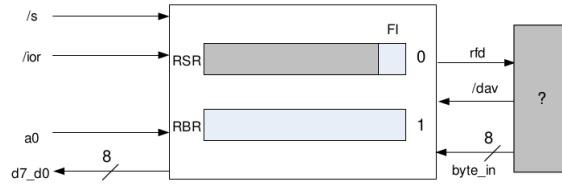
- /s segnale di select
- filo di /ior (o /iow)

- 8 fili di dati
- nessun filo di indirizzo (dato che ha una sola porta)
- 8 fili di ingresso (o uscita) dal lato del dispositivo

Si possono anche avere interfacce di ingresso/uscita dove la porta a indirizzo pari è per la lettura mentre quella dispari per la scrittura tramite della poca logica combinatoria per i select.

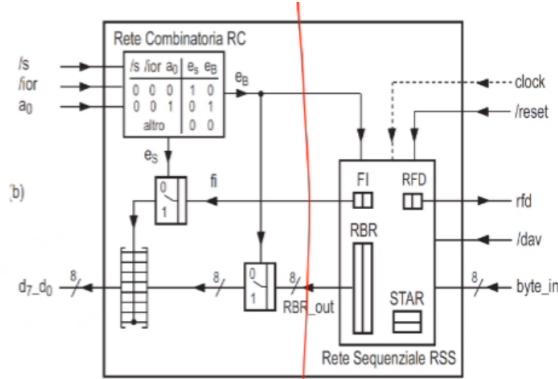
### Interfacce parallele con handshake (ingresso)

Per permettere la sincronizzazione sono dotate, lato processore, del flag FI mentre, lato dispositivo, dei fili di handshake (/dav e rfd).



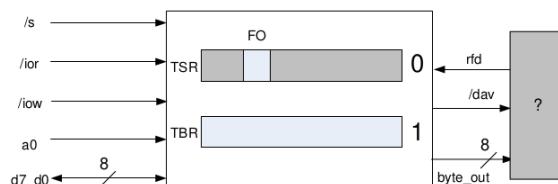
**Visione funzionale** La RC interna deve generare i segnali di abilitazione per le tri-state quando il processore accede in lettura a RBR o RSR (mai in conduzione contemporaneamente).

Si ha poi una RSS che gestisce l'handshake.



### Interfacce parallele con handshake (uscita)

Visione funzionale:



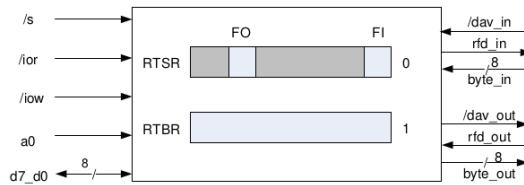
C'è una RC con ruolo analogo a quella di ingresso. Unica differenza è che  $e_B$  stavolta non serve per abilitare le tri-state ma per far progredire l'handshake.

**Nota:** il flag *FO* è il bit 5 del registro (sesto bit).

La RSS gestisce gli handshake in maniera duale a prima.

### Interfaccia parallela di ingresso-uscita

I due tipi appena visti possono essere connessi in un'unica interfaccia parallela:



**Nota:** non ci sono problemi se  $/s=0$ ,  $/ior=0$ ,  $a0=0$  perché vanno in conduzione le tristate di *INT\_IN* e *INT\_OUT* ma i registri *FO* e *FI* sono su due fili diversi del bus dati.

#### 5.1.2 Interfaccia seriale start/stop

Interfaccia nella quale la trasmissione dei singoli bit avviene in modo seriale. In realtà la trasmissione di byte avviene sempre in maniera seriale, quello che le differenzia dalle parallele è che al suo interno avviene la serializzazione di unità trasmissive più grandi.

L'interfaccia:

- riceve dal bus byte (perché il processore scrive byte nei registri I/O) e trasmette all'esterno bit
- riceve dall'esterno bit e presenta al processore byte

Un pc ha di solito più di una interfaccia seriale (modem, mouse<sup>1</sup> ecc.). Molti di questi dispositivi hanno del firmware configurabile, questo è possibile tramite una interfaccia seriale.

Essendo molto complesse noi ne vediamo una versione semplificata, prima però vediamo come avviene la comunicazione seriale tra 2 entità: si hanno 2 linee di cui una a massa che serve come riferimento e una che porta una tensione la quale rappresenterà solo 2 valori:

- **markin** ovvero 1 logico
- **spacing** ovvero 0 logico

La trasmissione di un bit consiste nel tenere la linea in uno dei due stati per un determinato tempo T detto **tempo di bit**. Un insieme di bit scambiato si chiama **trama** o **frame** che per adesso assumiamo essere costituita da un byte trasmesso dal LSB al MSB.

Per avere trasmissione in entrambe le direzioni servono 3 fili di cui 2 di tensione e uno a massa.

Per affrontare il problema della sincronizzazione si ha che:

- entrambi i lati concordano sul tempo di bit T
- entrambi i lati concordano sul numero di bit che compone la trama (di solito 5-8)

---

<sup>1</sup>un tempo

- una trama deve essere riconoscibile, devono concordare sul modo che indica l'inizio di una trama:
  - normalmente la linea stra in marking, quando voglio iniziare la trama la porto nello stato di spacing: si ha quindi che ogni trama inizia con il bit 0 (non fa parte della trama in sè) detto bit di start.
  - una volta che è stato trasmesso l'ultimo bit di una trama bisogna riportare la linea in uno stato di marking **almeno** per un tempo di bit (bit di stop).

**Nota :** Osservazione: *Se voglio trasmettere trame di un byte sarà necessario utilizzare almeno due bit in più che non hanno alcun significato informativo, questo comporta che se posso trasmettere un certo numero di bit al secondo  $x = 1/T$  non posso trasmettere delle trame lunghe un byte ad una velocità netta maggiore di  $x \cdot 8/10$  bit al secondo. Non potendo sfruttare la piena velocità della linea di trasmissione e massimizzarne quindi l'efficienza converrebbe aumentare il numero di bit per trama (dato che per mandarne  $n$  ne devo mandare  $n = 2$ ).*

*Avendo però i clock del trasmettitore e del ricevitore una discrepanza, per evitare di campionare il bit "fuori" dal bit che si vuole campionare bisogna cercare il più possibile di campionare i bit vicino alla loro metà (non possiamo conoscere infatti il segno di  $\Delta T$ )*

*Per sapere quali bit vengono trasmessi bisogna quindi:*

- aspettare  $3/2T$  da quando la linea transisce da marking a spacing
- campionare il valore
- aspettare  $T$  e ripetere il campionamento

*In questo modo (campionando nel mezzo) si evitano anche imprecisioni sui fronti date dai componenti elettrici.*

*Per poter fare ciò senza che si sfori mai dal bit bisogna avere che*

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

*da cui  $\frac{\Delta T}{T} \leq \frac{1}{2n}$  ovvero l'errore relativo che si può tollerare sul clock è inversamente proporzionale al numero di bit che devono essere trasmessi, si ha quindi un limite per la precisione.*

*L'inverso del tempo di bit si chiama bitrate e si misura in bit secondo.*

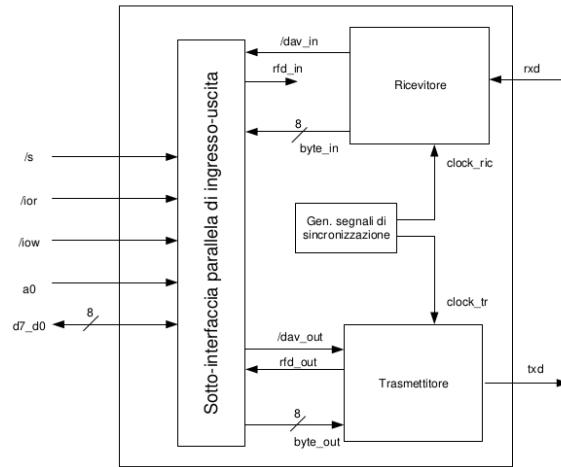
**Nota:** le tensioni non sono le solite caratteristiche di quelle di una rete logica: 1 viene fatto corrispondere a una tensione negativa (tra -3V e -25V) mentre 0 viene fatto corrispondere a una tensione positiva (tra +3V e +25V)

## Visione funzionale e struttura interna dell'interfaccia

Dal punto di vista del programmatore è identica, lato dispositivo è composta da due fili (nel caso di ingresso/uscita):

- txd
- rxd

Dal punto di vista della struttura interna c'è una sottointerfaccia parallela di I/O con handshake che colloquia con due RSS dette trasmettitore e ricevitore.



### Descrizione del trasmettitore

### Descrizione del ricevitore

Non potendo controllare il flusso di dati in ingresso dalla linea seriale non ha senso che gestisca un handshake completo con la sottointerfaccia parallela (se i dati non vengono letti in tempo vengono sovrascritti).

**Nota :** Velocità del clock: *per poter campionare i bit a metà del tempo di bit al massimo il clock del ricevitore dovrà avere un periodo pari a  $\frac{T}{2}$  ovvero frequenza doppia rispetto al bitrate del canale. Conviene avere un clock ancora più veloce per avere una migliore stima dell'istante in cui inizia il bit di start: garanzia per maggiore per i campionamenti successivi. (noi ipotizziamo un periodo pari a un sedicesimo del tempo di bit)*

### 5.1.3 Conversione analogico/digitale e digitale/analogica

La grandezza analogica che consideriamo è la tensione che convertiremo in un numero (naturale o intero) in base 2 e viceversa.

Questa ( $v$ ) sarà su una scala di FSR volts (Full-Scale Range) e verrà convertita nel numero  $x$  su  $N$  bit. A seconda dell'interpretazione si può distinguere:

- conversione unipolare
- conversione bipolare

Definiamo  $K = \frac{FSR}{2^N}$  **costante di proporzionalità** tra i due intervalli, che implica una conversione ideale

$$v = K \cdot x$$

Tuttavia ci dobbiamo accontentare di:

$$|v - K \cdot x| \leq err$$

dove  $err$  è l'errore di conversione dato da:

- imprecisioni circuituali (si parla quindi di **errore di non linearità**)
- **errore di quantizzazione:** solamente nella conversione A/D bisogna convertire una grandezza continua in una discreta e per farlo si perde parte dell'informazione per via dell'arrotondamento.

L'errore di non linearità deve essere più piccolo di  $\frac{k}{2}$  altrimenti si avrebbero intervalli parzialmente sovrapposti:

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

L'errore massimo di quantizzazione è, a prescindere, pari a  $\frac{k}{2}$ : se divido il FSR in  $2^N$  intervalli larghi  $K$  la conversione sarà:

- esatta al centro dell'intervallo
- errata di  $\pm \frac{k}{2}$  agli estremi

Alla fine abbiamo quindi:

- nella D/A:  $err < \frac{k}{2}$  (= solo non linearità)
- nella A/D:  $err < \frac{k}{2} + \frac{k}{2} = K$  (= non linearità e quantizzazione)

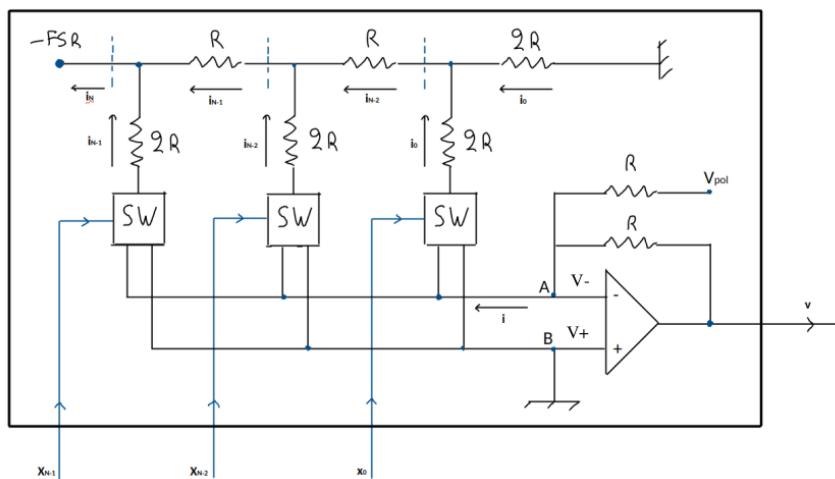
**Tempi di risposta** I convertitori hanno le seguenti prestazioni:

- D/A essendo circuiti "combinatori" semplici sono velocissimi (pochi  $ns$ )
- A/D hanno tempi di risposta variabili in quanto circuiti sequenziali. Quelli che vediamo noi (ad approssimazioni successive (SAR)) sono nell'ordine di qualche centinaio di  $ns$

**Nota:** I convertitori bipolarari rappresentano gli interi con rappresentazione in traslazione.

**Nota:** Per convertire un numero da traslazione a complemento a 2 basta complementare il bit più significativo.

### Convertitore Digitale/Analogico e relativa interfaccia di conversione



Dato che gli switch sono messi a massa alla destra di ogni tratteggio ho resistenza pari a  $R$  (infatti ho ricorsivamente, a partire da destra, due resistenze  $2R$  in parallelo che risultano quindi essere  $R$  la quale va in serie con la  $R$  a sinistra che risulta quindi  $2R$  ecc.)

Si ha quindi che in ogni ramo verticale scorre la stessa corrente che scorre nel ramo orizzontale sulla destra e quindi sul ramo verticale a sx avrà il doppio della corrente. All'estremo sinistro si avrà quindi corrente pari a  $2^N$  volte la corrente  $i_0$  che esce da massa a destra.

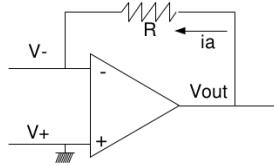
Se dunque la tensione massima di uscita ha valore FSR la corrente a sinistra varrà  $i_N = \frac{FSR}{R}$  da cui posso ricavare  $i_0$  come:

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

Gli switch visti nell'immagine sono interruttori comandati da una variabile di comando quale l'i-esimo bit della rappresentazione del numero da convertire in analogico. Essi commutano ciò che sta sopra con massa (B) quando la variabile vale 0 o, quando vale 1, con la linea (A).

Si ha poi l'amplificatore operazionale, esso:

- non fa passare corrente
- in uscita da una tensione  $V^{out} = \alpha \cdot (V^+ - V^-)$  con  $\alpha >> 1$  (siamo nell'ordine delle centinaia)



Dato che  $V^+$  è connesso a massa si ottiene che:  $V^{out} = -\alpha \cdot V^-$  da cui si ha

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

$$\begin{aligned} V^{out} &= -\alpha \cdot V^{out} + \alpha R \cdot i_a \\ V^{out} &= \frac{\alpha}{1 + \alpha} \cdot R \cdot i_a \approx R \cdot i_a \end{aligned}$$

da cui si conclude che  $V^- \approx 0$ .

L'amplificatore operazionale serve quindi ad ancorare a zero la tensione del punto A senza alterare il bilanciamento della corrente, la quale uscente da A verso sinistra varrà:

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

$$= i_0 \cdot \sum_{i=0}^{N-1} 2^i \cdot x_i$$

che notiamo essere la rappresentazione in base due del naturale  $X$ , sostituendo quindi il valore prima trovato di  $i_0$  si ha dunque

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

quindi variando gli switch si varia anche la corrente che scorre da A verso sinistra.

Per far uscire la tensione giusta dal convertitore vediamo come le equazioni di bilancio al nodo A siano:

$$\begin{aligned}\frac{K}{R} \cdot X &= \frac{V_{pol} + V}{R} \\ V &= K \cdot X - V_{pol} \\ V &= K \cdot \left( X - V_{pol} \cdot \frac{2^N}{FSR} \right)\end{aligned}$$

quindi:

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

Capiamo quindi come un convertitore D/A è un circuito "combinatorio", che garantisce velocità ma che è esposto a problemi di transazioni multiple. Per evitare problemi di questo tipo si aggiunge un filtro **passa-basso** a valle di questi che taglia variazioni a frequenza troppo elevata.

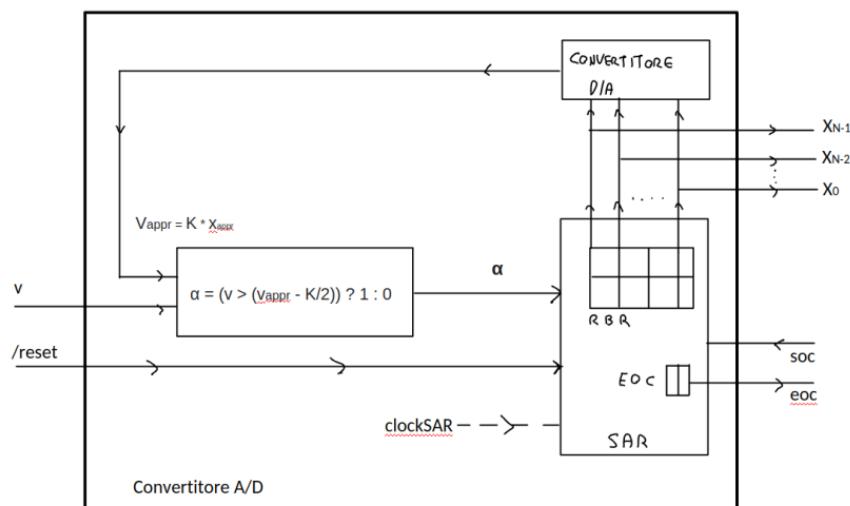
Analizzando gli errori sulle resistenze sulla destra si nota che:

- quello sulla R in serie a  $V_{pol}$  si traduce con un errore di offset: trasla la retta in verticale lasciandone invariata la pendenza
- quello sulla R in serie alla tensione di uscita si traduce con un errore di guadagno: modifica la pendenza della retta

Queste due R sono dunque dei potenziometri che permettono la taratura del convertitore (per ottenere errore di non linearità minore di  $\frac{k}{2}$ ).

**Nota:** Da punto di vista funzionale un'interfaccia per la conversione D/A appare come una interfaccia di uscita senza handshake dotata solo di TBR.

### Convertitore Analogico/Digitale e relativa interfaccia di conversione



Quello che vediamo è un convertitore A/D ad approssimazioni successive (a 8 bit).

È composto da una RSS centrale detta SAR (Successive Approximation Register) e un convertitore D/A (della stessa polarità del A/D).

Si comporta come segue: data una tensione in ingresso opera una ricerca logaritmica di essa: prende un byte intermedio, lo converte in analogico, lo confronta e decide se produrre un byte più grande o più piccolo, ripete.

In base 2 questo si traduce con mettere a posto un bit alla volta a partire dal MSB: per poter fare questo procedimento (lento: qualche decina di  $ns$ ) il convertitore A/D porta avanti con l'interfaccia di conversione a cui è connesso un handshake soc/eoc; inoltre la tensione analogica in ingresso deve rimanere costante per tutto il tempo di conversione (si fa tramite un latch analogico).

Descrizione di SAR semplice: guarda disegno.

Descrizione interfaccia: aggiungere.