Circuiti Combinatori: Applicazioni

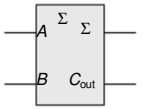
Contesto

- Finora abbiamo visto come **realizzare circuiti combinatori a partire da specifiche/tabelle di verità**
- Inoltre, abbiamo visto come **minimizzare il numero di gate utilizzati per creare un circuito**
- Nelle prossime lezioni, passeremo in rassegna alcuni **esempi di circuiti combinatori tipicamente disponibili e utilizzati per realizzare circuiti digitali più complessi**

Adder

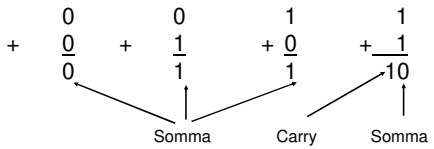
Half Adder

- Somma due bit
- Produce **2 output**:
  - La **somma** dei due bit
  - **L'eventuale riporto (carry)**



Inputs		Outputs	
A	*B*	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

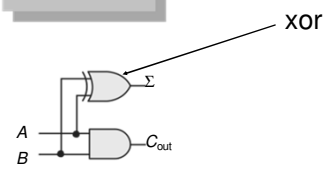
Half Adder



Realizziamolo…

Soluzione (banale!)

Inputs		Outputs	
A	*B*	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

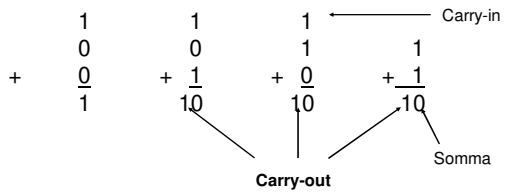


Full Adder

Motivazioni

- E' chiaro che per sommare numeri binari dovremmo **combinare più adder**
- **Problema: l'half-adder ha sempre due input**
  - Ci servirebbe un **ulteriore input per aggiungere il carry della somma delle cifre a destra**

Esempio

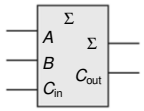


Full Adder

- **3 ingressi**: A, B, e carry-in
- **2 uscite:** Somma, Carry-out

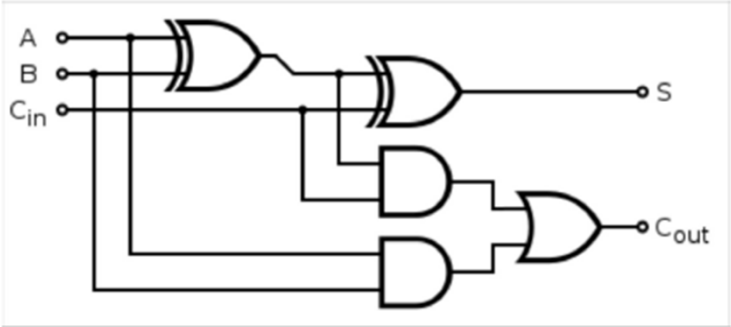
Inputs			Outputs	
A	B	C_{in}	C_{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Rappresentazione:



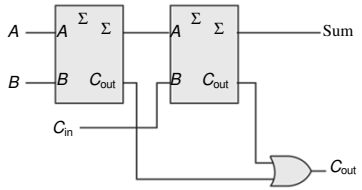
Proviamo a realizzarlo…

Soluzione



Ora, proviamo a realizzarlo **<u>combinando degli half-adder</u>**…

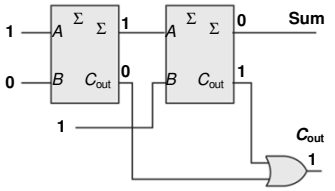
Full Adder



Inputs			Outputs	
A	B	C_{in}	C_{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

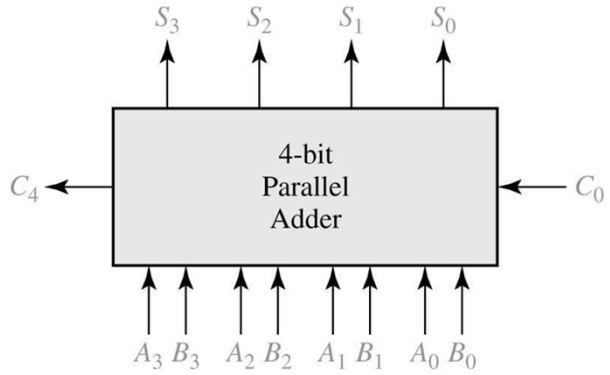
Esempio

Inputs			Outputs	
A	B	C_{in}	C_{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Disegniamo un circuito che implementa un adder a 4 bit

4-bit Parallel Adder



4-bit Parallel Adder

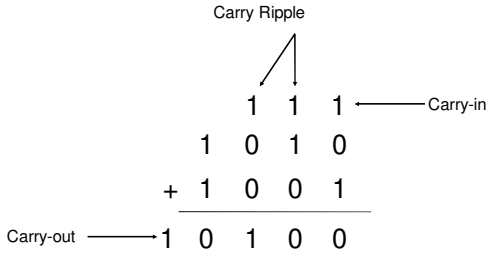
- **Primo approccio:** costruire una tabella di verità per un circuito con 9 ingressi (2 bit per ogni addendo, + carry in) e 5 uscite (4 somme parziali e carry out)
- **Approccio migliore:** cercare di riutilizzare l'adder a 2 bit e costruire adder a n bit connettendo tra loro più adder

Interconnessione adder

Due soluzioni possibili:

- **Ripple carry adder**
- **Carry lookahead adder**

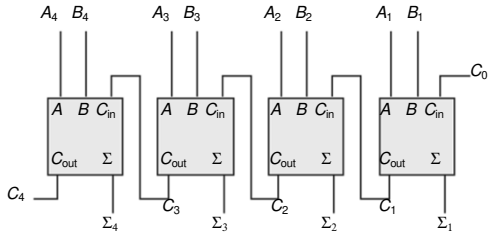
Ripple Carry Adder



Il carry-in sulla somma in posizione **i** è disponibile una volta che la somma in posizione **i-1** è stata effettuata e quindi l'eventuale carry-out prodotto

Ripple Carry Adder

Combiniamo due (o più) full adder in maniera da gestire somme più bit.



Il **carry out (C_4) non è pronto fintanto che il carry non viene propagato lungo tutti gli adder**. Discorso analogo per C_3, C_2, C_1

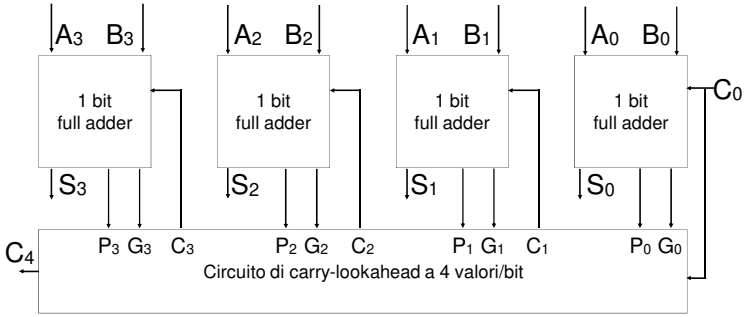
Ripple Carry Adder

- Consiste in **n** full adder
- Il carry-out alla posizione **i** è connesso al carry-in alla posizione **(i+1)**
- **Vantaggio:** design semplice
- **Svantaggio:** lentezza
  - Ciascun bit di somma può essere calcolato solo una volta che il bit di carry-out precedente è stato calcolato
  - **Ritardo ~ n * Ritardo(FA)**

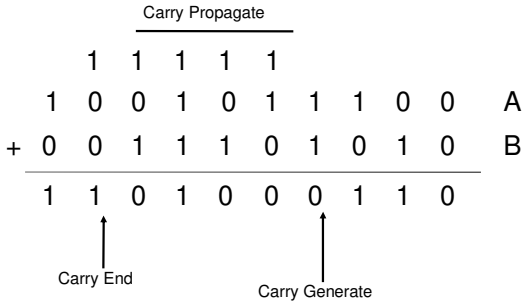
Quindi…

- Il Ripple Carry Adder può diventare estremamente **lento per un numero elevato di bit**
- Il **Carry Lookahead Adder** garantisce **minori tempi di risposta al costo di hardware addizionale necessario per gestire il calcolo del carry propagation**

Carry Lookahead Adder



Carry Lookahead Adder



2 azioni: generazione e propagazione

Carry Lookahead Adder

- Usa della logica addizionale rispetto all'adder per capire se una somma di bit **genera** o **propaga** un carry
- Un carry è **generato** se sia A_i che B_i sono = 1
  - Funzione "generate": $\mathbf{G(A_i, B_i) = A_i \cdot B_i}$
- Un carry è **propagato** se almeno uno tra A_i e B_i è 1
  - **Funzione "propagate":** $\mathbf{P(A_i, B_i) = A_i + B_i}$

Carry Lookahead Adder

Per ciascun valore (o stage) dell'adder il carry-out del bit **i** (ovvero il carry in del bit **i+1**, indicato con $\mathbf{C_{i+1}}$) può essere definito in termini delle **funzioni di generazione e propagazione del bit precedente**

$$\mathbf{C_{i+1} = G_i + (P_i \cdot C_i)}$$

carry-out carry-in

C_{i+1} è alto se lo stage i genera un carry ($\mathbf{G_i}$)
OPPURE propaga C_i: $\mathbf{(P_i \cdot C_i)}$

Carry Lookahead Adder

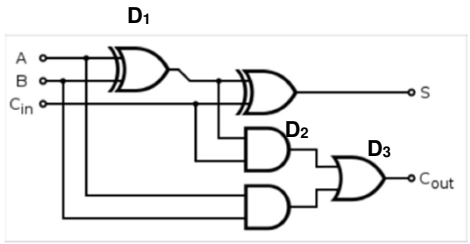
Per lo stage **0**:

- $C_1 = G_0 + (P_0 \cdot C_0)$
- $C_1 = (A_0 \cdot B_0) + ((A_0 + B_0) \cdot C_0)$

Notare che:

- C_1 è funzione degli ingressi del sistema
- Il circuito è a **3 livelli, quindi ho un ritardo causato da 3 gate**
  - Ok, **fin qui il ritardo è lo stesso generato dal full'adder sul carry**…

Ritardo sul carry in un Full Adder



Ritardo sul carry: $D_1+D_2+D_3$

Carry Lookahead Adder

Stage 1:

- $C_2 = G_1 + (P_1 \cdot C_1)$
- $C_2 = (A_1 \cdot B_1) + ((A_1 + B_1) \cdot C_1)$
- $C_2 = (A_1 \cdot B_1) + ((A1 + B1) \cdot ((A_0 \cdot B_0) + ((A_0 + B_0) \cdot C_0))$
- $C_2 = A_1B_1 + (A_1+B_1) \cdot A_0B_0 + (A_1+B_1)(A_0+B_0) \cdot C_0$
- C_2 è **funzione degli input primari del circuito e del carry-in d'ingresso**
  - C_2 non dipende dal carry degli stadi precedenti, solo da C_0
  - Circuito a **3 livelli, quindi ritardo solo su 3 gate**

Carry Lookahead Adder

Stage 2:

- $C_3 = G_2 + (P_2 \cdot C_2)$
- $C_3 = G_2 + (P_2 \cdot (G_1 + (P_1 \cdot C_1))$
- $C_3 = G_2 + (P_2 \cdot (G_1 + (P_1 \cdot (G_0 + (P_0 \cdot C_0)))$
- $C_3 = G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0)$
- $C_3 = G_2 + P_2 \cdot G_1 + P_2P_1G_0 + P_2P_1P_0C_0 =$
  $A_2B_2 + (A_2+B_2) \cdot A_1B_1 + (A_2+B_2) \cdot (A_1+B_1) \cdot A_0B_0 + (A_2+B_2)(A_1+B_1)(A_0+B_0) \cdot C_0$
- C_3 è ancora funzione soltanto degli input del circuito, e del carry d'ingresso C_0
  - Il circuito risultante **è sempre a 3 livelli, quindi ho un ritardo causato da 3 gate**

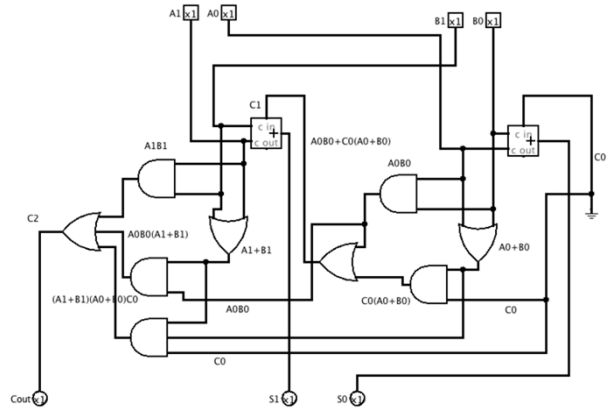
Carry Lookahead Adder

Stage i:

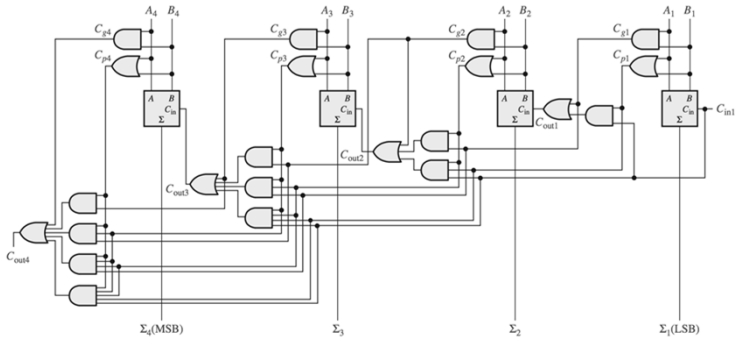
$C_{i+1} = F(G_0..G_i, P_0..P_i, C_0)$

- Per i>4, il circuito di lookahead **diventa troppo complesso e richiede troppi gate**…
- **Tradeoff: velocità vs. area occupata sul silicio/numero di gate**

Implementazione (2 bit)

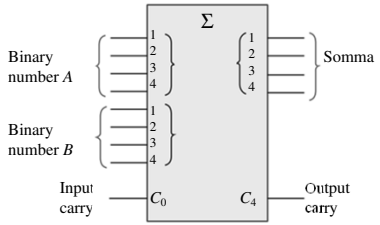


Implementazione completa



Circuito Integrato Parallel Adder a 4 bit

2 ingressi a 4 bit, un bit di carry in e uno di carry out



- Esempio di componente: 74LS283
- Look-ahead carry, con delay massimo sul carry out = 17 ns.

4-bit CLA

National Semiconductor

June 1989

54LS283/DM54LS283/DM74LS283
4-Bit Binary Adders with Fast Carry

General Description

These full adders perform the addition of two 4-bit binary numbers. The sum (Σ) outputs are provided for each bit and the resultant carry (C4) is obtained from the fourth bit. These adders feature full internal look ahead across all four bits. This provides the system designer with partial look-ahead performance at the economy and reduced package count of a ripple-carry implementation.

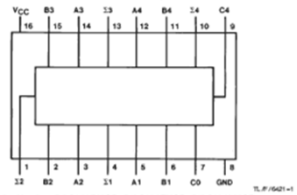
The adder logic, including the carry, is implemented in its true form meaning that the end-around carry can be accomplished without the need for logic or level inversion.

Features

- Full-carry look-ahead across the four bits
- Systems achieve partial look-ahead performance with the economy of ripple carry
- Typical add times
  - Two 8-bit words 25 ns
  - Two 16-bit words 45 ns
- Typical power dissipation per 4-bit adder 95 mW
- Alternate Military/Aerospace device (54LS283) is available. Contact a National Semiconductor Sales Office/Distributor for specifications.

Connection Diagram

Dual-In-Line Package



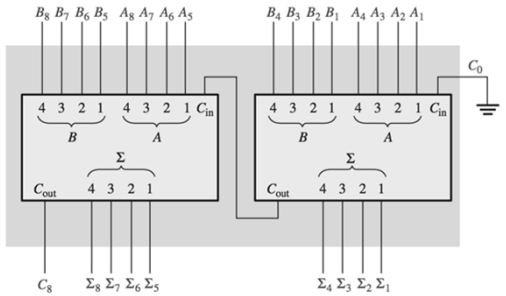
TL/F/6421–1

Order Number 54LS283DMQB, 54LS283FMQB, 54LS283LMQB, DM54LS283J, DM54LS283W, DM74LS283M or DM74LS283N
See NS Package Number E20A, J16A, M16A, N16E or W16A

Come realizzo adder a 8, 16 bit?

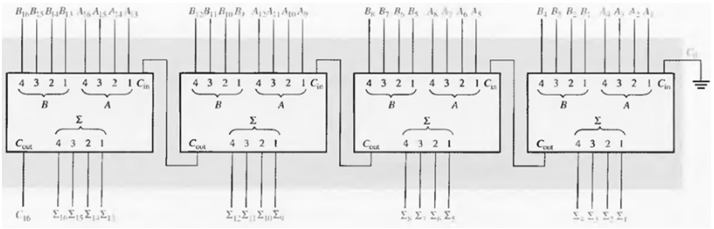
Oltre 4 bit, **la logica lookahead diventa troppo complessa**…

2 adder in cascata



Compromesso: **2 4-bit lookahead collegati in ripple carry-in**

4 adder in cascata



Circuito Subtractor

Sottrazione tra 2 numeri binari

1 1 0 1 -
0 1 1 0 =

Come procediamo?
Potremmo **eseguire la sottrazione così come sappiamo fare per i numeri decimali**…

Realizzazione subtractor

Potremmo costruire un circuito **sottrattore in base alla tabelle di verità della sottrazione**

- **Poco comune e poco efficiente**

Soluzione: usare la **rappresentazione dei numeri negativi in complemento a 2**

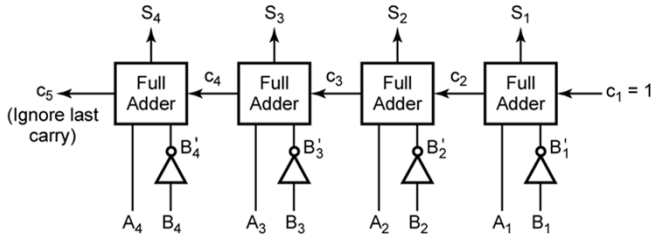
- Utilizzo di un adder
- Dovendo calcolare A-B, si effettua la **somma tra A e il complemento a 2 di B**

Problemi

- Non posso convertire in numeri negativi numeri positivi con la cifra più significativa alta
- Di fatto **riduco il range di positivi che posso rappresentare** con **n** bit (numeri da $-2^{(n-1)}$ a $2^{(n-1)}-1$
- Necessità di **gestire gli overflow**

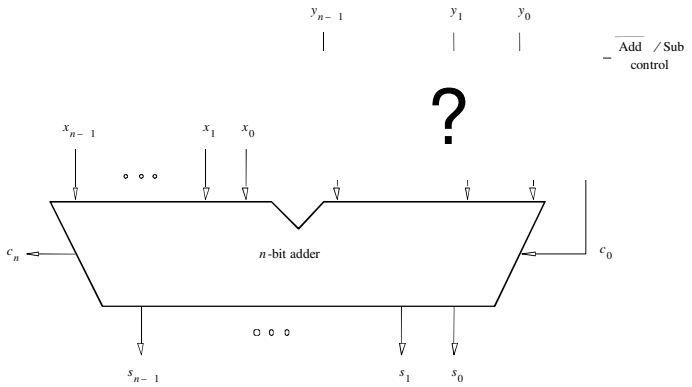
Sottrattore a 4 bit

Il complemento a **2 si ottiene negando i bit e aggiungendo 1** (ovvero ponendo a 1 il carry-in del circuito)



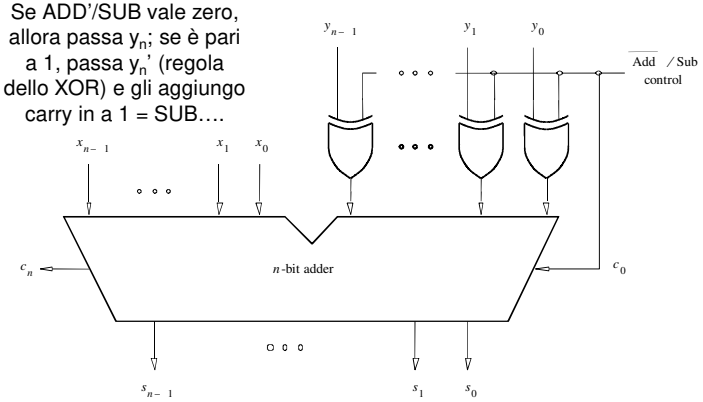
Ora realizziamo un circuito in grado di effettuare **sia addizioni che sottrazioni**…

Adder/Subtractor a n bit



Adder/Subtractor a n bit

Se ADD'/SUB vale zero, allora passa y_n; se è pari a 1, passa y_n' (regola dello XOR) e gli aggiungo carry in a 1 = SUB….



Individuazione di Problemi di Overflow

Overflow nelle addizioni

- Si verifica se il **risultato è fuori range**
- Non può verificarsi quando sommiamo un numero positivo con un numero negativo
- Quindi, si verifica se **sommiamo due numeri con lo stesso segno**
- Effetto collaterale che potrebbe verificarsi:
  - **Somma due numeri positivi → Numero negativo**
  - **Somma due numeri negativi → Numero positivo**

Overflow nelle sottrazioni

- Si verifica se il risultato è fuori range
- Non **può verificarsi quando sottraiamo due numeri con lo stesso segno**
- Può verificarsi, invece, **quando sottraiamo un numero positivo da uno negativo o viceversa**.
  - # positivo - # negativo → # negativo
  - # negativo - # positivo → # positivo

Calcolo overflow

$$V = C_{i-1} \text{ XOR } C_{i-2}$$

In pratica, metto in **XOR i carry-out degli ultimi 2 bit, ovvero il carry in e il carry out dell'ultimo bit**

Perché funziona?

Calcolo overflow

$V = C_{i-1}$ XOR C_{i-2}

Caso 1: carry-in=0, carry-out=1 (C_{i-1}=1, C_{i-2}=0)

- Se il carry-in è 0, l'unica ragione per cui il carry out possa essere 1 è quando i bit più significativi dei due addendi sono entrambi 1 ($x_{i-1} = 1$, $y_{i-1} = 1$), e la somma produce 0 nel bit più significativo
- Ovvero, **sommo due negativi** e ottengo un numero positivo!

Calcolo overflow

$V = C_{i-1}$ XOR C_{i-2}

Caso 2: 1 carry-in=1, carry-out=0 (C_{i-1}=0, C_{i-2}=1)

- In questo caso, se il carry-in è 1, l'unico caso in cui il carry out possa essere 0 è quando entrambe le cifre più significative sono 0 ($x_{i-1} = 0$, $y_{i-1} = 0$), e la somma produce 1 nel bit più significativo
- Ovvero, sommo **due numeri positivi e ottengo un numero negativo**

Esempi

Entrambi positivi

0100+ 0010= ——— C_3=0, C_4=0 NO OVERFLOW	0111+ 0110= ——— C_3=1, C_4=0 OVERFLOW
(4+2)	(7+6)

Entrambi negativi

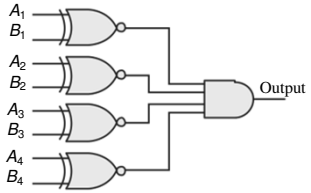
1001+ 1100= ——— C_3=0, C_4=1 OVERFLOW	1101+ 1100= ——— C_3=1, C_4=1 NO OVERFLOW
(-7-4)	(-3-4)

Comparatori

Realizzazione comparatore

- Il comparatore confronta due numeri binari e ne determina la relazione (A>B, A=B, A<B)
- **Nella forma più semplice, il comparatore verifica soltanto la condizione di uguaglianza**
- Come realizziamo tale comparatore nel caso di numeri a n bit (es. 4 bit)?

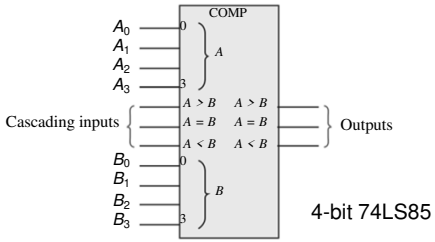
Risposta: mettiamo in AND 4 porte XNOR



Circuito Integrato Comparatore

Oltre che gli output A>B, A=B, A<B, il circuito integrato è dotato di "cascading input" corrispondenti.

Tali input sono utilizzati per consentire l'interconnessione di più comparatori.



4-bit 74LS85

Interconnessione modulare comparatori

E' possibile integrare più comparatori in cascata, interconnettendo gli output A>B, A=B, A<B del primo col i corrispondenti ingressi del secondo.

Gli ingressi A>B, A<B del primo comparatore sono collegati con un valore logico basso. L'ingresso A=B del primo comparatore è collegato con un valore logico alto.