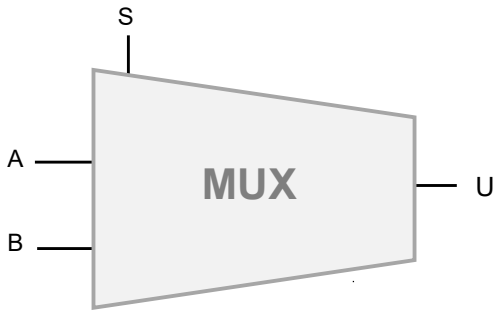
Multiplexer a 2 vie (1 ingresso) di selezione a 1 bit

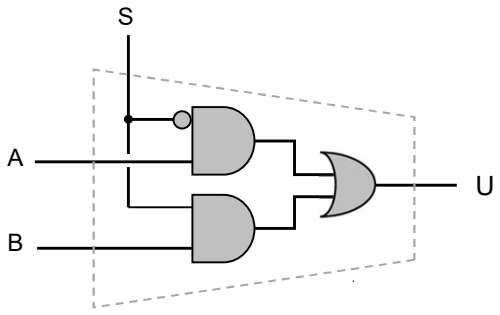
INPUT			OUTPUT
S	A	B	U
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



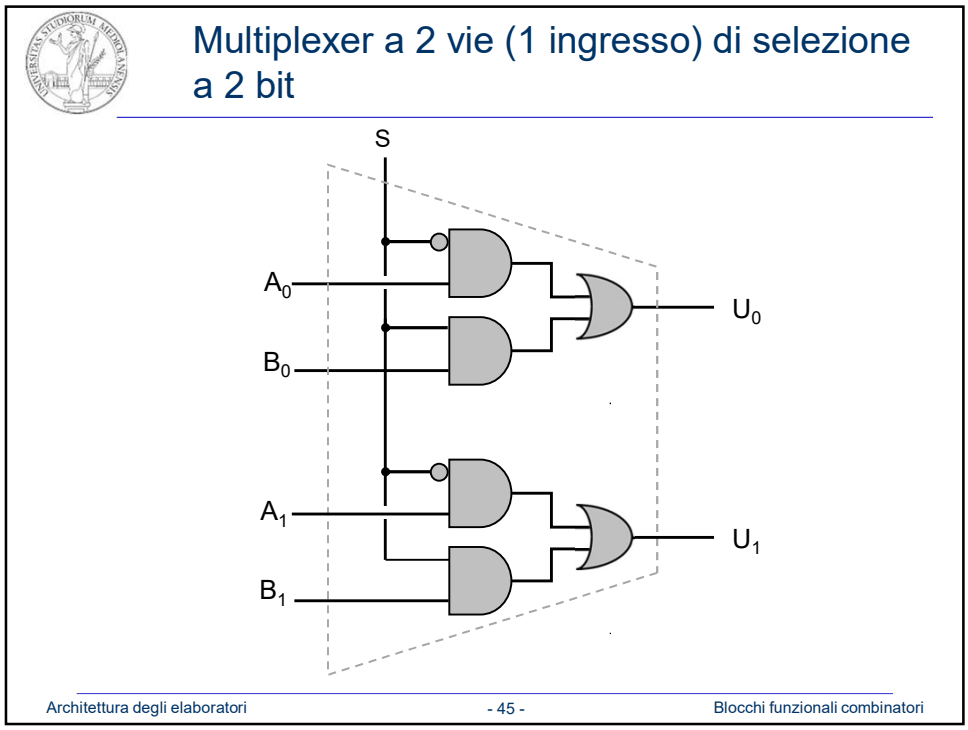
41

Multiplexer a 2 vie (1 ingresso) di selezione a 1 bit

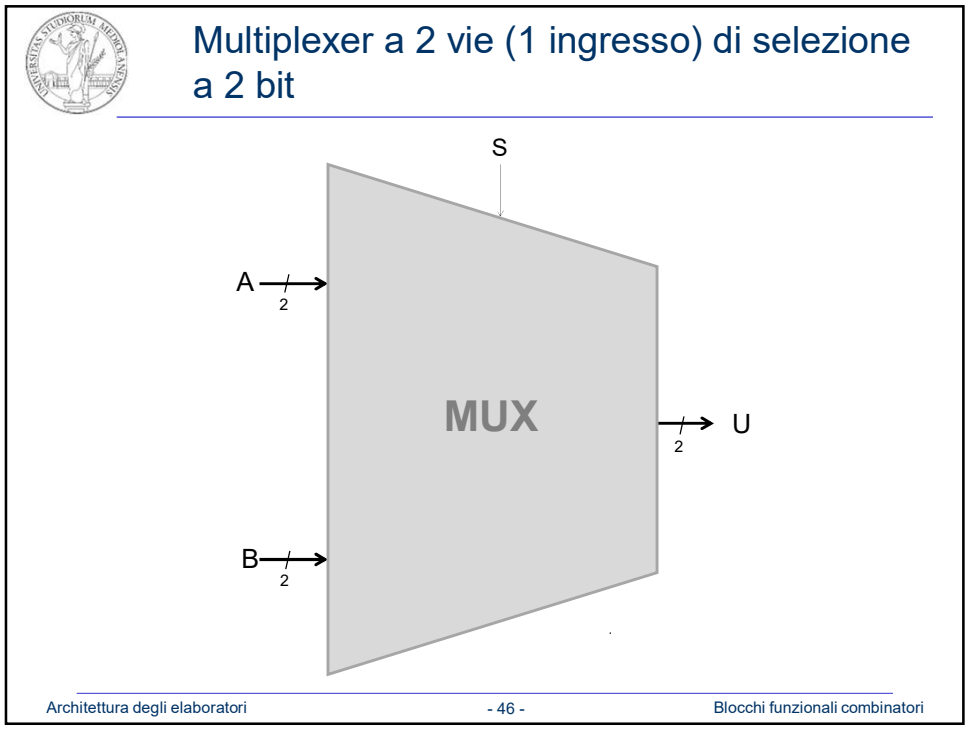
INPUT			OUTPUT
S	A	B	U
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



42

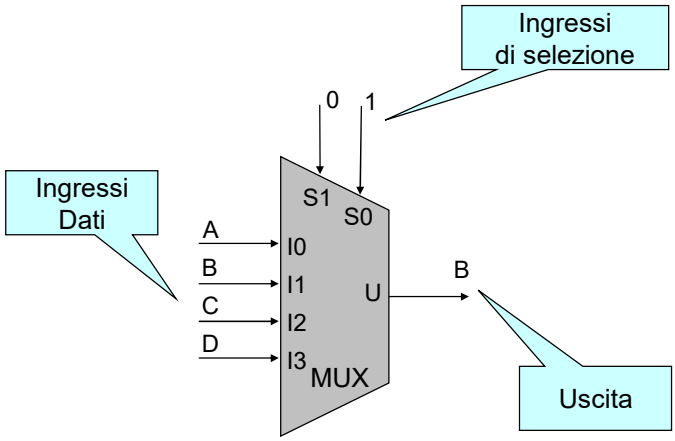

Multiplexer a 2 vie (1 ingresso) di selezione a 2 bit
S
A0
B0
A1
B1
U0
U1
Architettura degli elaboratori
- 45 -
Blocchi funzionali combinatori

45


Multiplexer a 2 vie (1 ingresso) di selezione a 2 bit
S
A
2
B
2
MUX
U
2
Architettura degli elaboratori
- 46 -
Blocchi funzionali combinatori

46

Multiplexer a 2 ingressi di selezione



Multiplexer a 2 ingressi di selezione

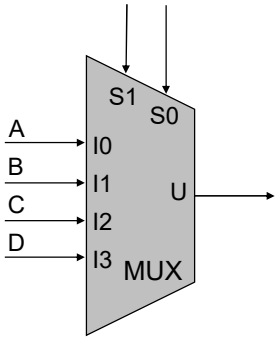
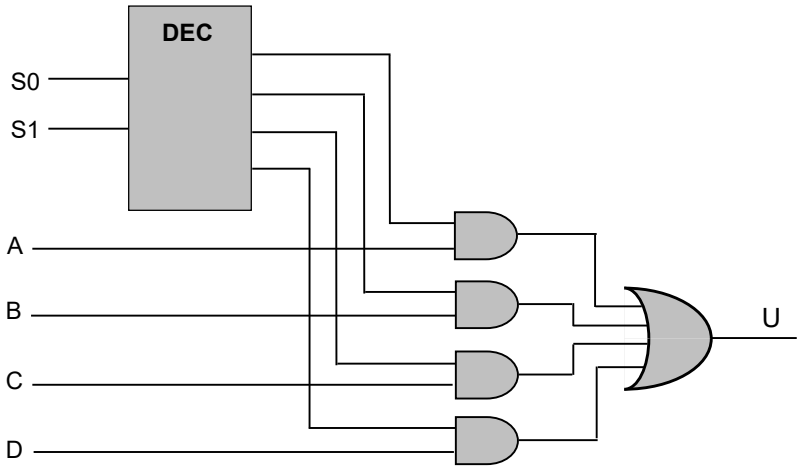


Tabella delle verità

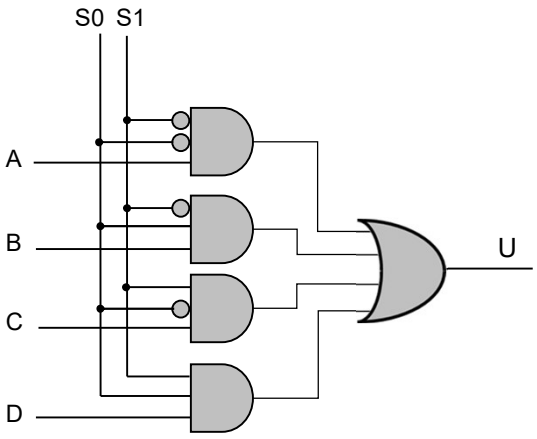
S1	S0	I0	I1	I2	I3	U
0	0	A	X	X	X	A
0	1	X	B	X	X	B
1	0	X	X	C	X	C
1	1	X	X	X	D	D

Multiplexer a 2 ingressi di selezione: realizzazione

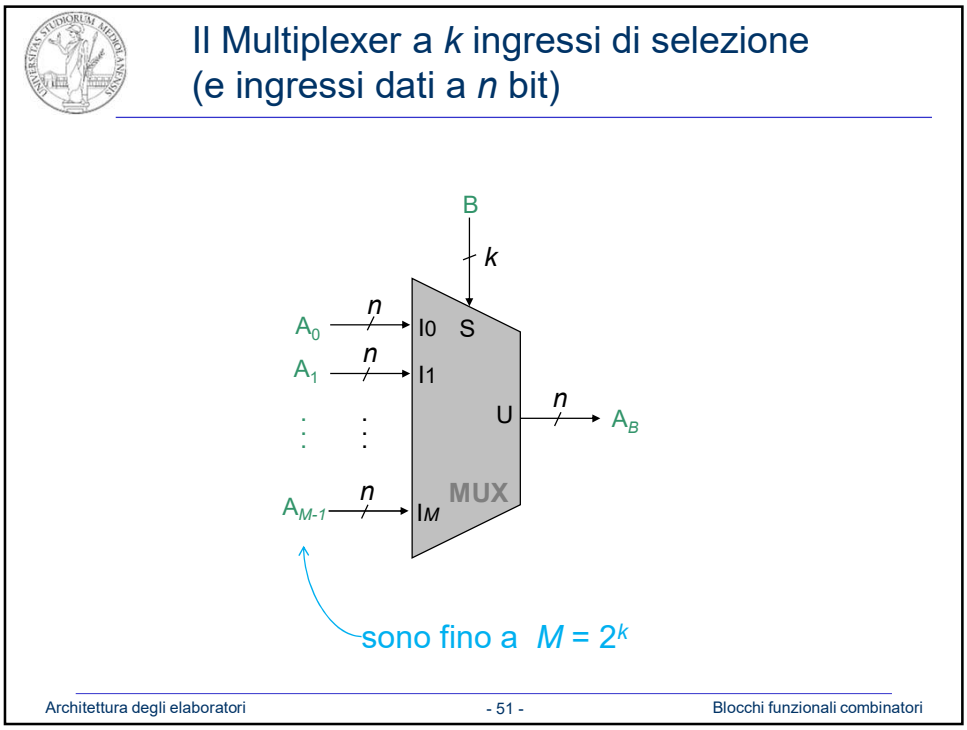


49

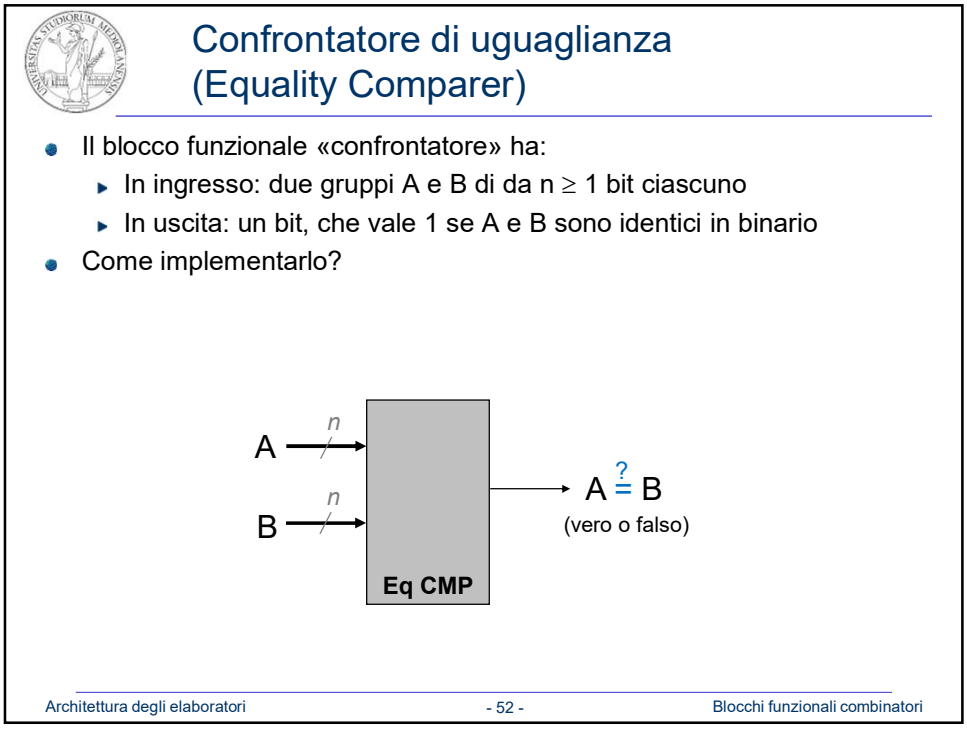
Multiplexer a 2 ingressi di selezione: realizzazione alternativa



50


Il Multiplexer a k ingressi di selezione
(e ingressi dati a n bit)
B
k
n
A0
I0
S
n
A1
I1
U
n
AB
MUX
n
AM-1
IM
sono fino a M = 2k
Architettura degli elaboratori
- 51 -
Blocchi funzionali combinatori

51


Confrontatore di uguaglianza
(Equality Comparer)
Il blocco funzionale «confrontatore» ha:
In ingresso: due gruppi A e B di da n ≥ 1 bit ciascuno
In uscita: un bit, che vale 1 se A e B sono identici in binario
Come implementarlo?
n
A
n
B
Eq CMP
A ? = B
(vero o falso)
Architettura degli elaboratori
- 52 -
Blocchi funzionali combinatori

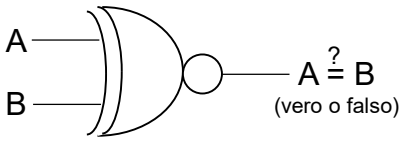
52

Ripasso: uno dei significati intuitivi dell'operatore booleano NXOR

NXOR
(«fa il contrario di XOR»)

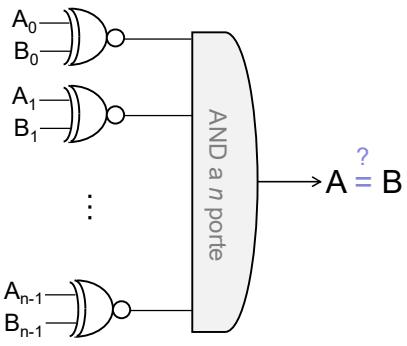
A	B	X	
0	0	1	*vero*
0	1	0	*falso*
1	0	0	*falso*
1	1	1	*vero*

uguali



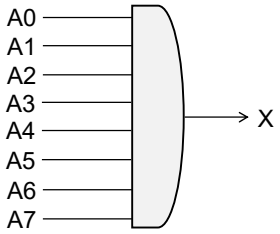
Confrontatore di uguaglianza (Equality Comparer)

- Evidentemente, due numeri sono uguali se corrispondono bit a bit
- Cioè, se hanno un match fra i loro 1mi bit,
  E ANCHE fra loro due 2ndi bit, E ANCHE i loro 3zi bit, etc…
- A circuito, per *n* bits:

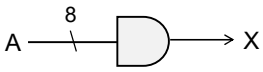


Porta AND a 8 input (1/2)

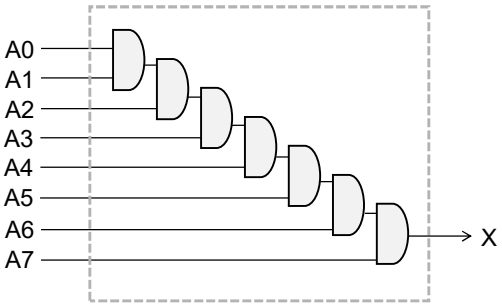
Ad alto livello:



oppure:



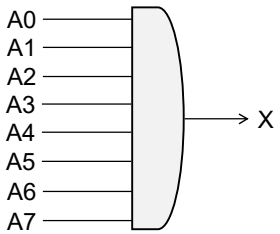
Implementazione 1:
(sequenziale)



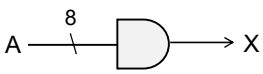
((((((A0·A1) · A2) · A3) · A4) · A5) · A6) · A7

Porta AND a 8 input (1/2)

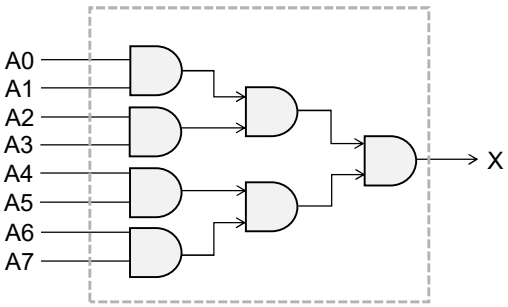
Ad alto livello:



oppure:



Implementazione 2:
(parallela)



((A0·A1) · (A2·A3)) · ((A4·A5) · (A6·A7))

Parallelismo = Ottimizzazione (dei tempi)
E' un concetto generale in HW! Esempio:

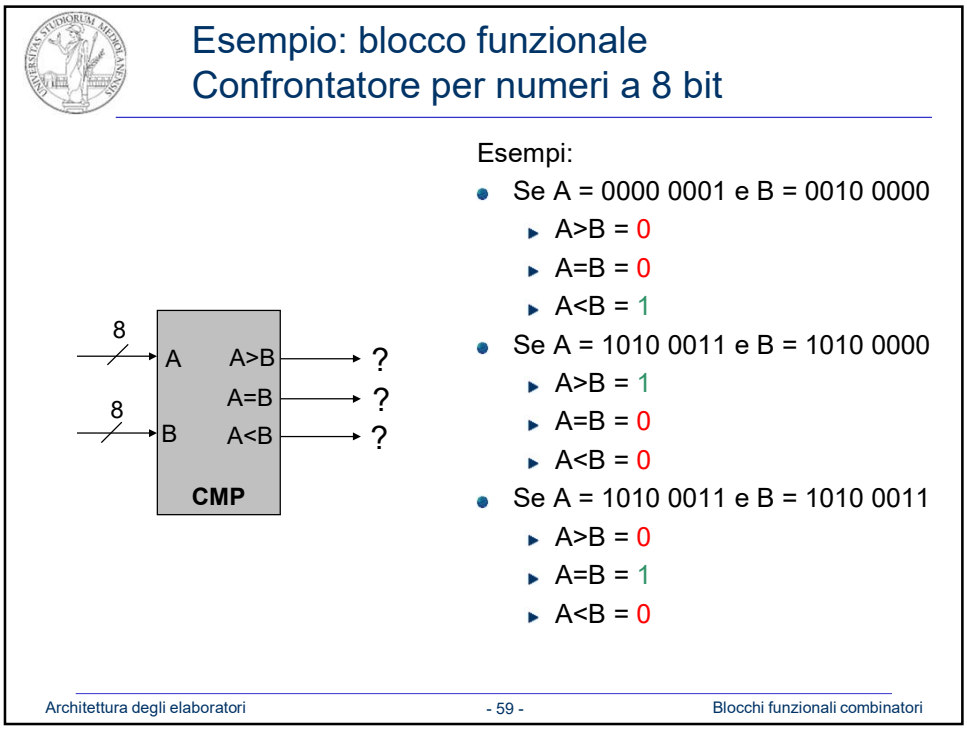
- Le porte AND a molti ingressi possono essere considerati mini-blocchi funzionali, implementabili con porte AND a 2 ingressi
  - La stessa cosa vale per le porte OR, XOR, NOR, NAND, etc, cioè tutte le porte degli operatori associativi
- Confronta le due implementazioni proposte di un AND a 8 input
  - Risultato computato : identico.
    I due circuiti sono "funzionalmente equivalenti",
    cioè calcolano la stessa funzione booleana («1 sse input tutti 1»)
    - Le espressioni booleane sono diverse, ma equivalenti
  - Numero di porte binarie usato : identico (questo si riflette nel costo monetario di realizzazione, nel consumo, nel riscaldamento):
  - Ma la 2a implementaz. sfrutta la capacità dell'HW di elaborare **in parallelo:** es, le prime 4 porte lavorano *contemporaneamente*
  - L'implementazione in parallelo è molto più efficiente in velocità; ha un tempo di commutazione men che dimezzato

57

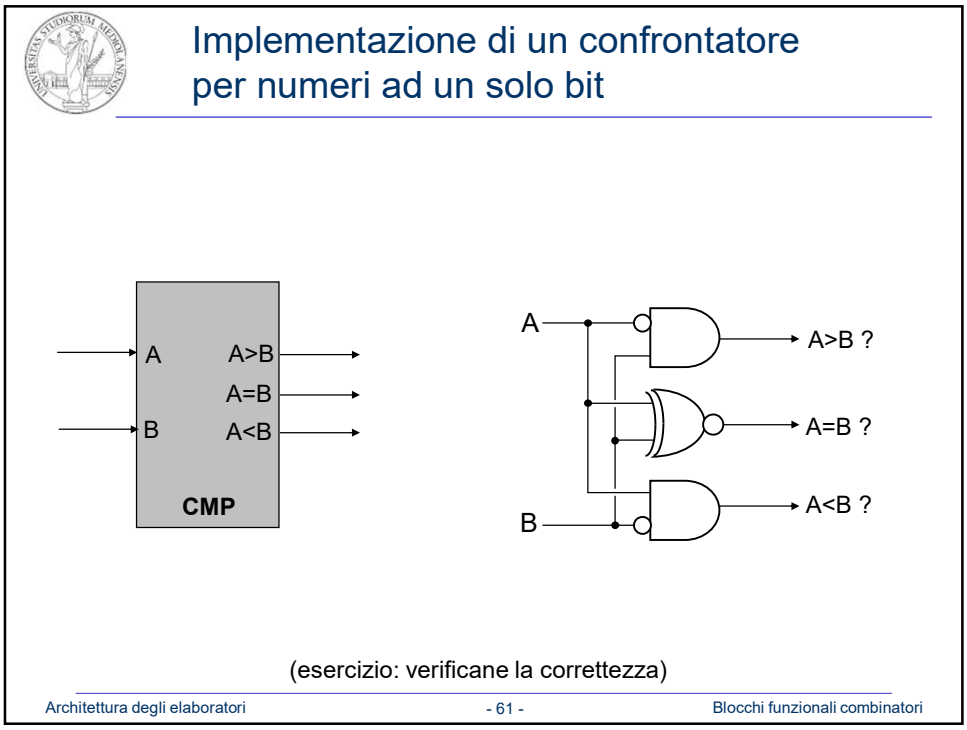
Confrontatore [completo]
(Comparer, o CMP)

- Il blocco funzionale confrontatore ha:
  - due gruppi A e B di ingressi da $n \geq 1$ bit ciascuno
  - tre uscite di un bit:
    - $A < B$
    - $A = B$
    - $A > B$
- Il blocco confronta i due numeri binari A e B da *n* bit presenti sui due gruppi di ingressi, e
  - attiva (mette a 1, in inglese, «to set») l'uscita corrispondente all'esito del confronto
  - azzera (mette a 0, to «reset»,) le altre due uscite, che corrispondono a condizioni false
- Nota: un confrontatore per numeri in complemento a 2 è diverso da uno in numeri senza segno
  - (vediamo quest'ultimo)

58


Esempio: blocco funzionale
Confrontatore per numeri a 8 bit
Esempi:
Se A = 0000 0001 e B = 0010 0000
A>B = 0
A=B = 0
A<B = 1
Se A = 1010 0011 e B = 1010 0000
A>B = 1
A=B = 0
A<B = 0
Se A = 1010 0011 e B = 1010 0011
A>B = 0
A=B = 1
A<B = 0
8
A
8
B
A>B
A=B
A<B
CMP
?
?
?
Architettura degli elaboratori
- 59 -
Blocchi funzionali combinatori

59


Implementazione di un confrontatore
per numeri ad un solo bit
A
B
A>B
A=B
A<B
CMP
A
B
A>B ?
A=B ?
A<B ?
(esercizio: verificane la correttezza)
Architettura degli elaboratori
- 61 -
Blocchi funzionali combinatori

61

Estensione di un numero: cioè aggiunta di cifre a sinistra

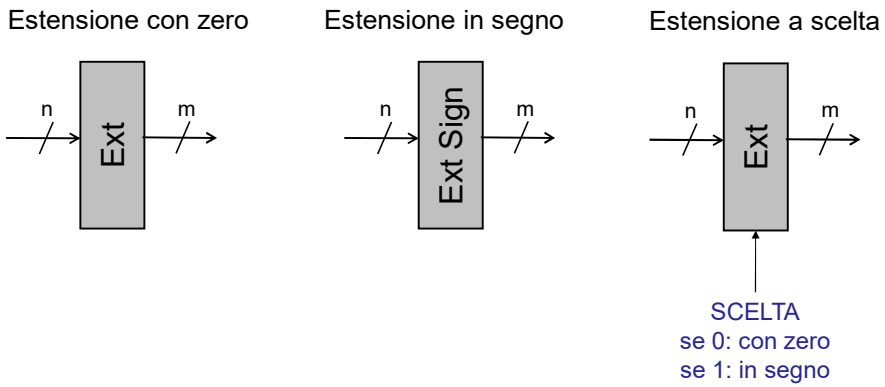
- A volte, vogliamo passare da una rappresentazione di un numero a *n* bit ad una rappresentazione dello stesso a *m* bit, con *m* > *n*
- Come si sa, *aggiungere zeri a sinistra* non modifica un numero:
  - 11011_2 = 27
  - $000\,11011_2$ = ancora 27
  - Questa operazione si chiama **estensione con zero**
- Ma attenzione: per estendere un numero in **CP2** occorre aggingere a sinistra: bit **0** se il MSB è 0, ma bit **1** se il MSB è 1. Esempi:
  - 01101_2 = +13
  - $000\,01101_2$ = ancora +13

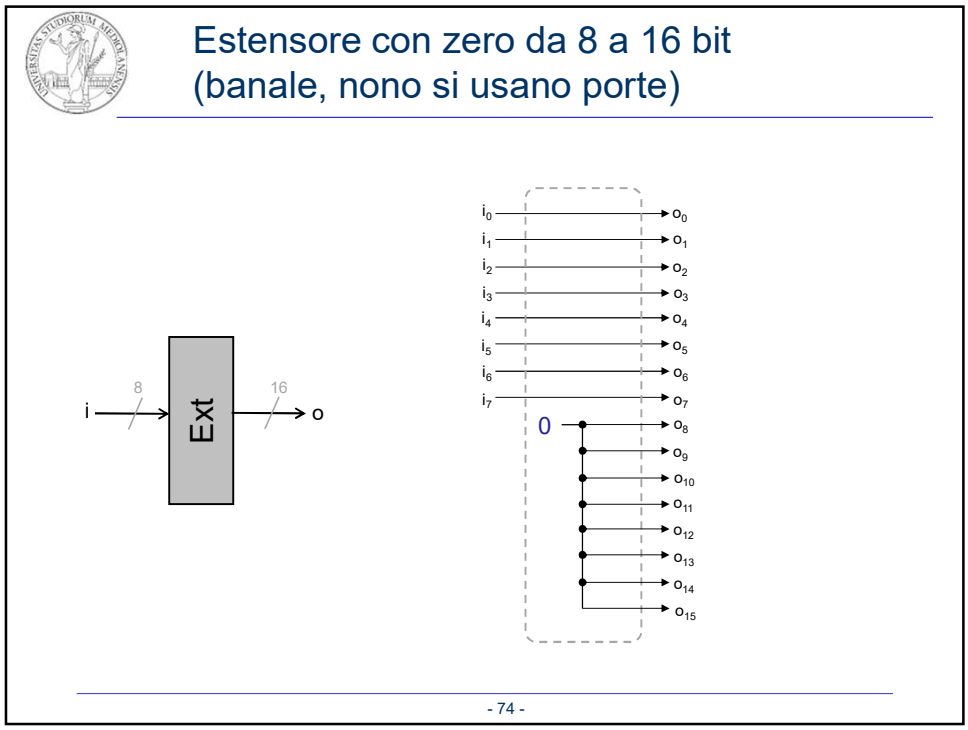
  Most Significant Bit, il bit più a sinistra
  - 11001_2 = -7
  - $111\,11001_2$ = ancora -7

  ←verificare!
  - Questa operazione si chiama **estensione in segno**
- Vediamo circuiti per implementare queste due operazioni

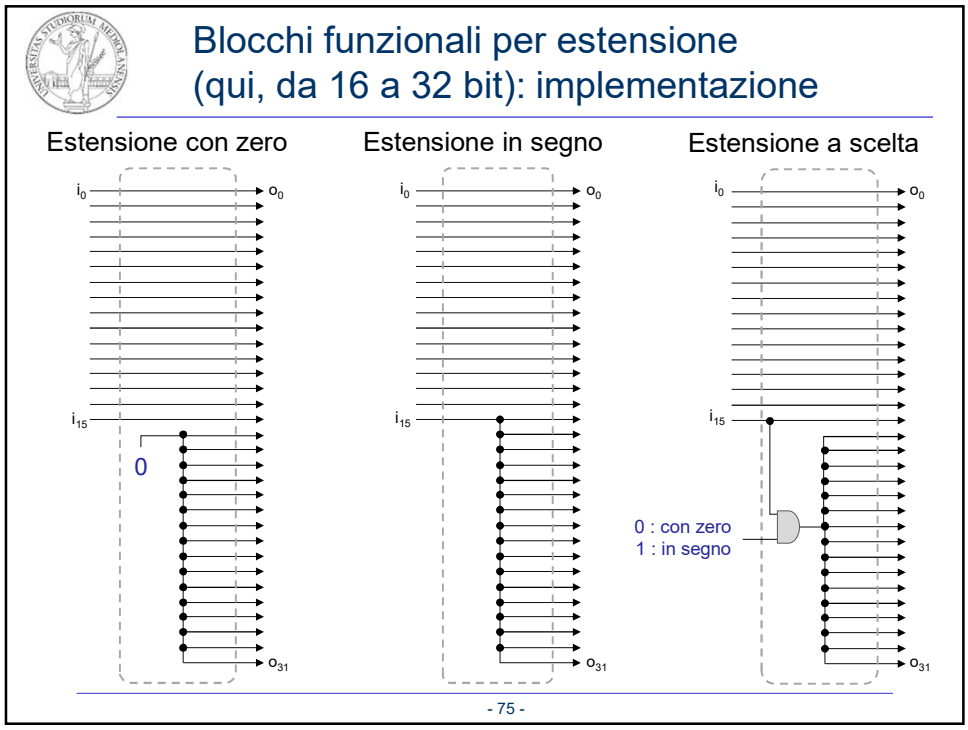
Blocchi funzionali per estensione

m > n




Estensore con zero da 8 a 16 bit
(banale, nono si usano porte)
i
8
Ext
16
o
i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7
o_0 o_1 o_2 o_3 o_4 o_5 o_6 o_7
0
o_8 o_9 o_{10} o_{11} o_{12} o_{13} o_{14} o_{15}
- 74 -

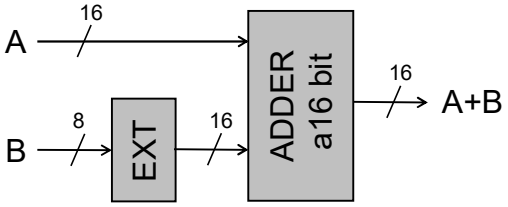
74


Blocchi funzionali per estensione
(qui, da 16 a 32 bit): implementazione
Estensione con zero
Estensione in segno
Estensione a scelta
i_0 i_{15} o_0 o_{31}
0
0 : con zero
1 : in segno
- 75 -

75

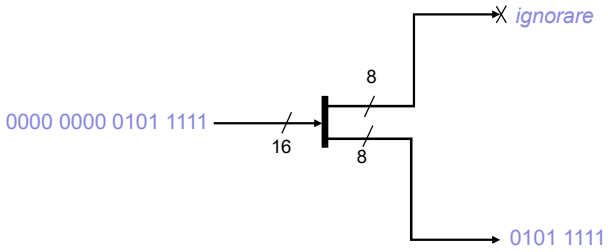
Esempio di uso dell'estensore: sommare valori interi di tipo diverso

- Come sommare (senza segno) un numero A a 16 bit ad un numero B a 8 bit?
- Risposta: posso usare un addizionatore a 16 bit, se…



Trancare un numero

- L'inverso di estendere un numero è troncarlo, cioè considerare solo le sue cifre meno significative.
  - scartando quelle più significative, e assumendo che siano zero
- Non abbiamo bisogno di un blocco funzionale per rappresentare questo procedimento. Basta scrivere (ad esempio):



Ripasso: lo shift a sinistra (moltiplicazione per 2, 4, 8 …)

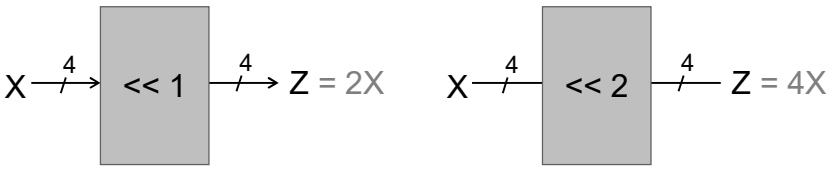
analogo in base 10:
1320 = 132 x 10
13200 = 132 x 100

- *Left-shift* (di *n*): spostare le cifre binarie *n* posti verso sinistra
  - *n* le cifre più a sinistra scompaiono, e da destra compaiono *n* **zeri**
- Ricorda:
  uno *shift* a sinistra in base 2 di 1 = raddoppio del numero
  uno *shift* a sinistra in base 2 di *n* cifre = moltiplicazione per 2^n
- Notazione algebrica: A << B («applica ad A lo shift a sin di B bits»)
- Esempi:
  00011011_2 = 27
  00011011_2 << 1 = 00110110_2 = 54 (= 27x2)
  00011011_2 << 2 = 01101100_2 = 108 (= 27x4)
  00011011_2 << 3 = 11011000_2 = 216 (= 27x8)
- Lo stesso vale anche in CP2! Esempi:
  11111011_2 = -5
  11111011_2 << 1 = 11110110_2 = -10
  11111011_2 << 2 = 11101100_2 = -20

←verificare!

Left Shift (con secondo parametro costante)

- Blocchi funzionale per shift: (usano…. zero porte!)
- Esempi per 4 bit:

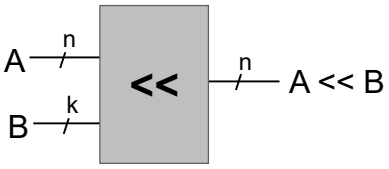


Left-Shift
domande e esercizi

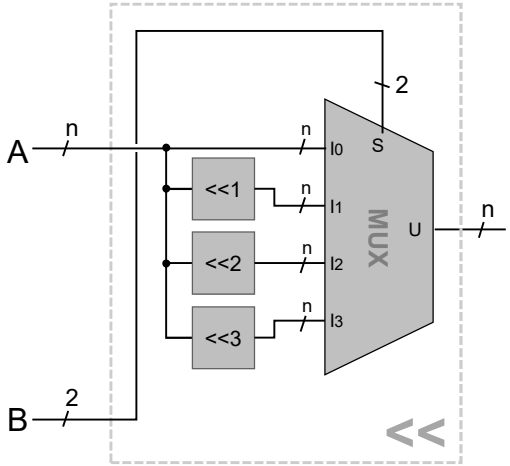
- **Tempo di commutazione**?
  - quali sono i tempi dei blocchi funzionali visti ("<<1", "<<2") ?
- **Overflow**?
  essendo una moltiplicazione per 2^n, il left-shift può generare overflow!
  - come accorgersi se "**<<1**" fa overflow, per **naturali** (no segno) ?
  - come accorgersi se "**<<2**" fa overflow, per **naturali** (no segno) ?
  - come accorgersi se "**<<1**" fa overflow, per **CP2** ?
  - come accorgersi se "**<<2**" fa overflow, per **CP2** ?
  - realizza un blocco funzionale che prevede un bit ulteriore di uscita "overflow", che vale 1 sse l'operazione ha generato un overflow.
- **Left shift a 2 parametri**?
  - per ora, abbiamo visto un blocco funzionale (pur senza porte) che shifta con secondo parametro costante.
  - Realizziamone uno con secondo paramero variabile, cioè che prende in input A (*n* bit), e B (2 bit), e dà in output A<<B

80

Left Shift
con secondo parametro variabile



Una realizzazione con K=2



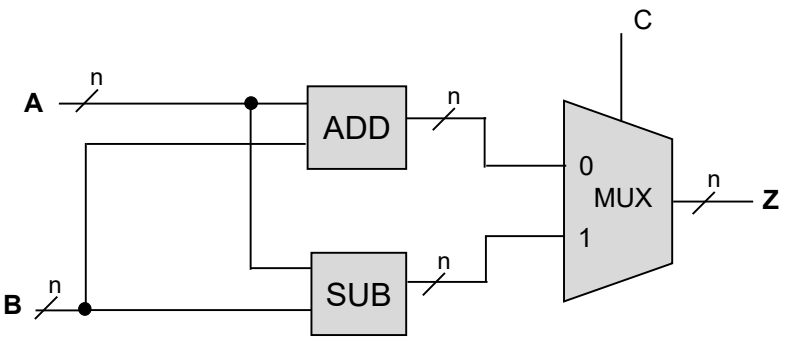
81

Un esempio di progetto con blocchi funzionali

- Si chiede di progettare un circuito digitale combinatorio, che abbia:
  - in ingresso due numeri binari interi (in CP2) A e B da *n* bit ciascuno
  - in ingresso un segnale di comando C da un bit
  - in uscita un numero binario intero Z (in CP2) da *n* bit tale che
    - Z = A + B , quando C = 0
    - Z = A – B , quando C = 1
- Si trascurano gli overflow

- In pratica:
  - C codifica il comando (ed A e B i suoi operatori), scelto in minuscolo insieme di istruzioni composto solo da { somma , sottrazione }
  - il dispositivo esegue l'istruzione richiesta, codificata da C
  - è il nostro primo passo verso un Instruction Set eseguibile da un elaboratore:-)

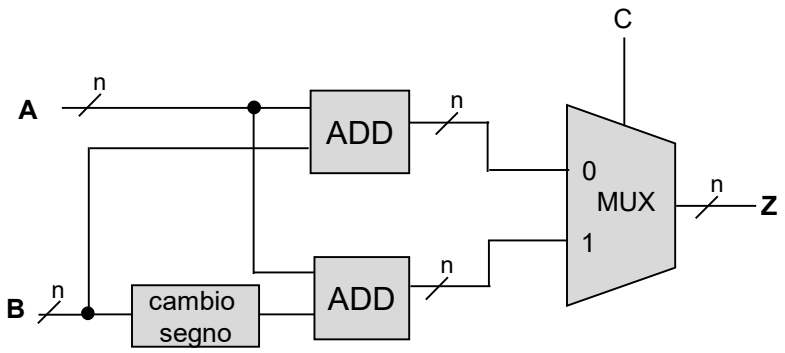
Soluzione 0



- Ma come realizzare la sottrazione ("SUB" - subtract)?

Soluzione 1

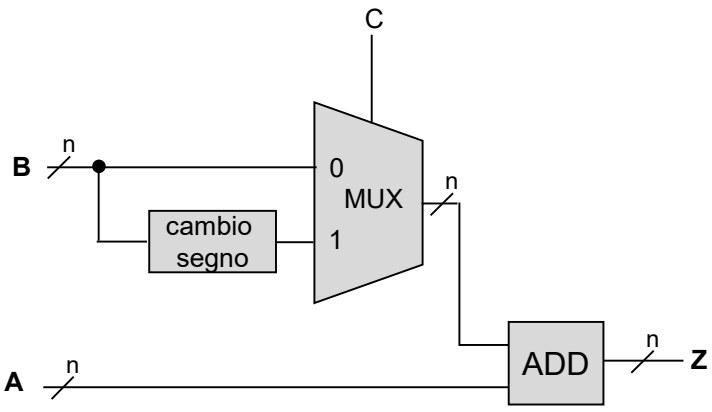
- Idea: usare un sommatore anche per la sottrazione… A – B = A + (– B)



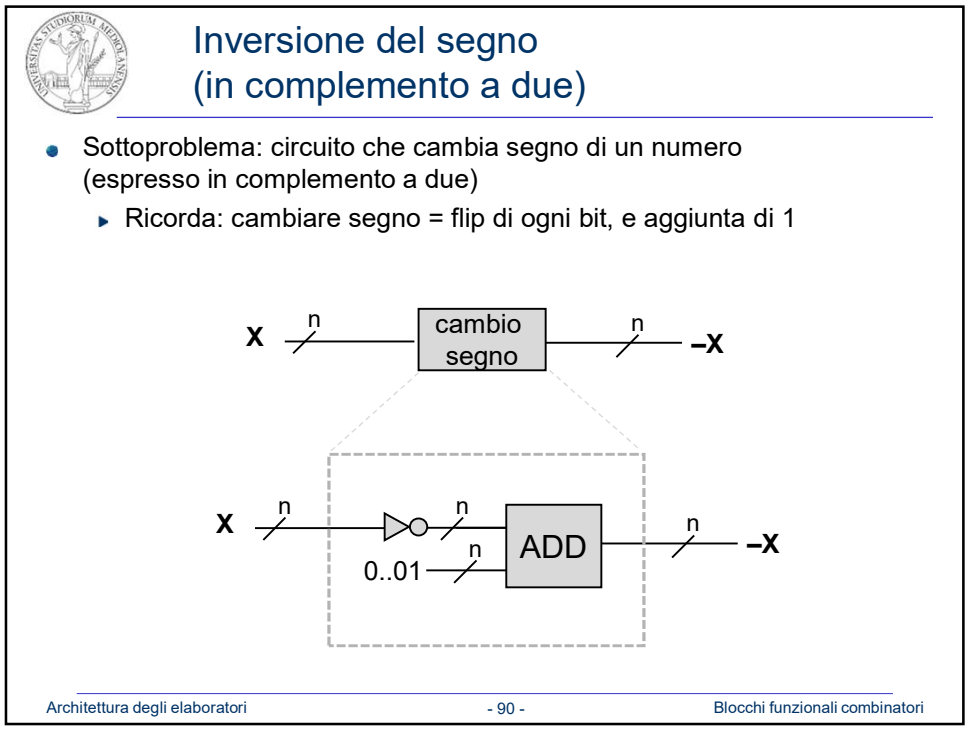
88

Soluzione 2

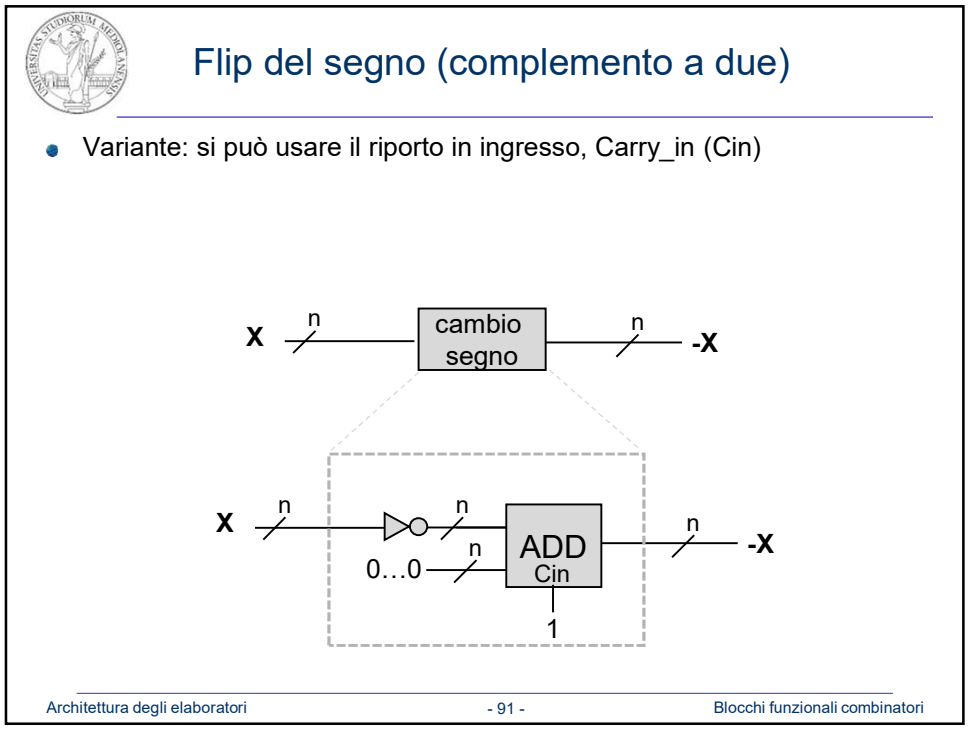
- Meno costosa della precedente: abbiamo risparmiato un sommatore!



89

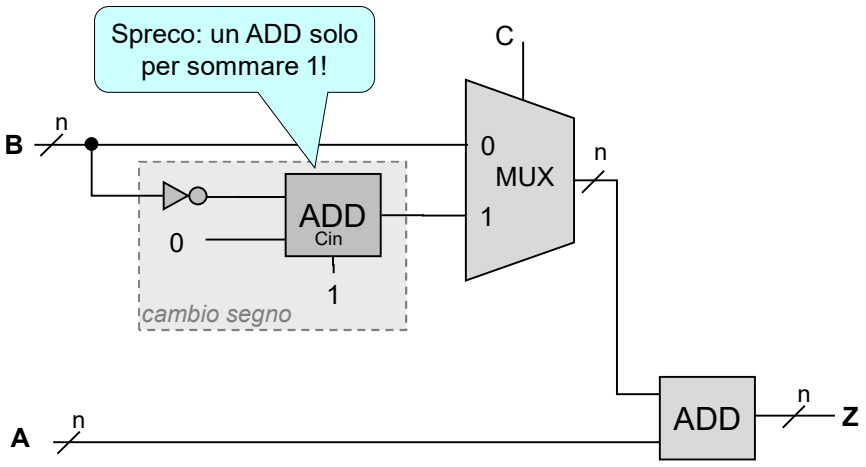

Inversione del segno
(in complemento a due)
Sottoproblema: circuito che cambia segno di un numero (espresso in complemento a due)
Ricorda: cambiare segno = flip di ogni bit, e aggiunta di 1
X
n
cambio segno
n
–X
X
n
n
ADD
n
0..01
n
–X
Architettura degli elaboratori
- 90 -
Blocchi funzionali combinatori

90


Flip del segno (complemento a due)
Variante: si può usare il riporto in ingresso, Carry_in (Cin)
X
n
cambio segno
n
-X
X
n
n
ADD
Cin
n
0…0
n
1
-X
Architettura degli elaboratori
- 91 -
Blocchi funzionali combinatori

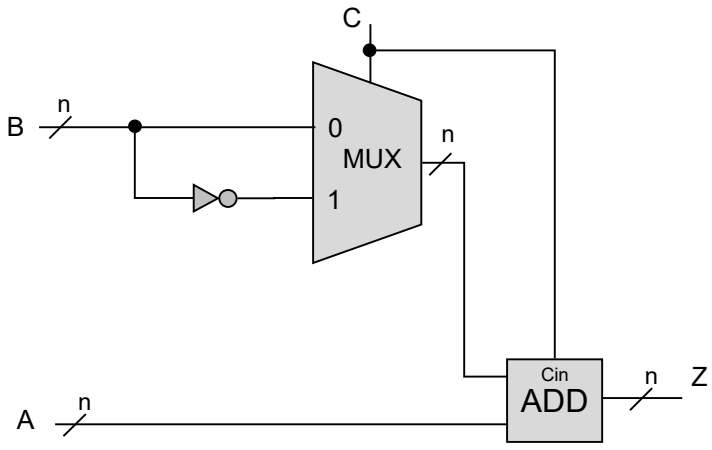
91

Soluzione 2

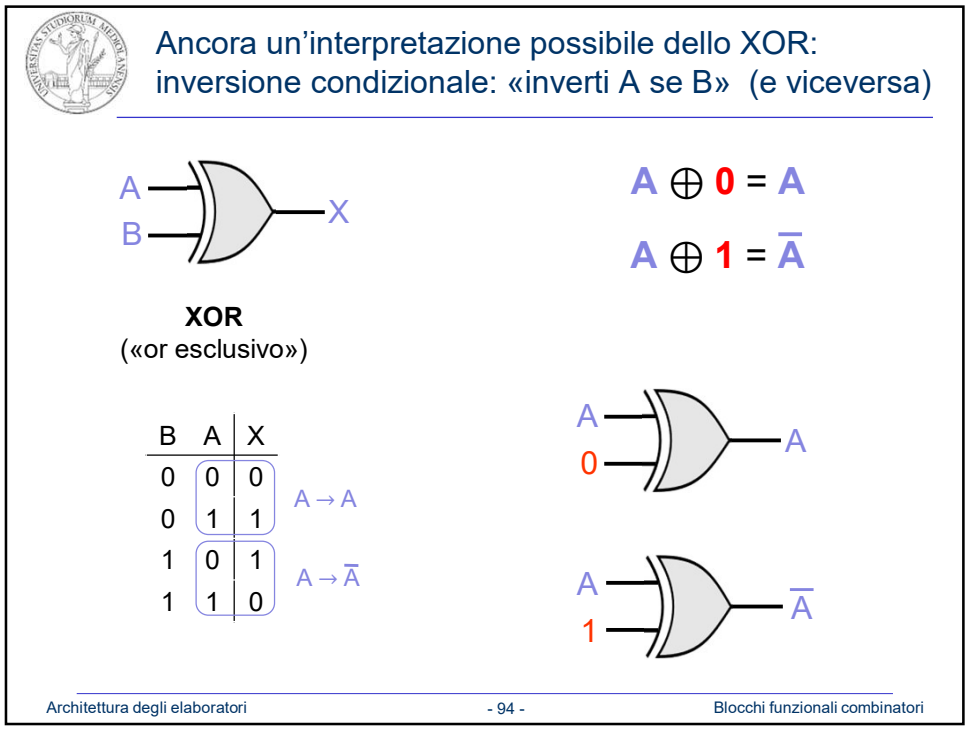


92

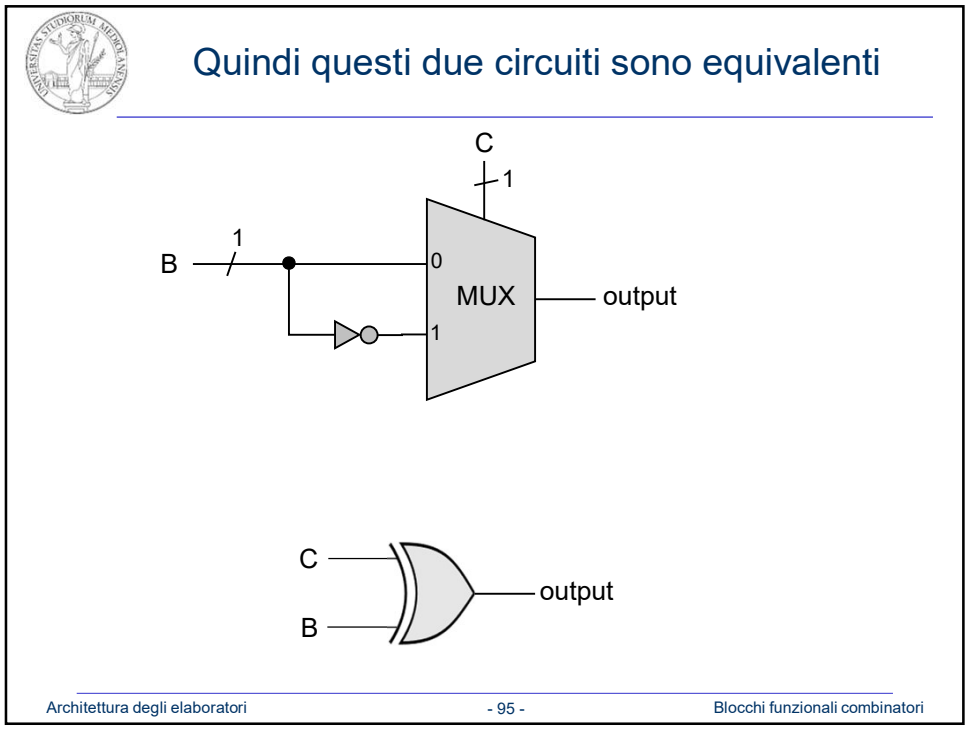
Soluzione 3 (ottimizzazione di 2)



93


Ancora un'interpretazione possibile dello XOR:
inversione condizionale: «inverti A se B» (e viceversa)
A
B
X
XOR
(«or esclusivo»)
A ⊕ 0 = A
A ⊕ 1 = Ā
B A X
0 0 0
0 1 1
1 0 1
1 1 0
A → A
A → Ā
A
0
A
A
1
Ā
Architettura degli elaboratori
- 94 -
Blocchi funzionali combinatori

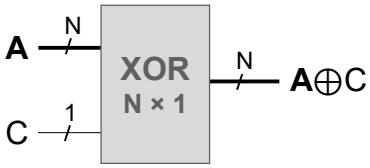
94

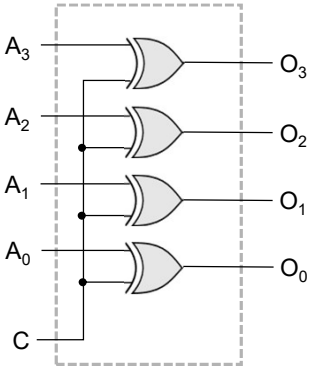

Quindi questi due circuiti sono equivalenti
C
1
B
1
0
MUX
output
1
C
B
output
Architettura degli elaboratori
- 95 -
Blocchi funzionali combinatori

95

Blocco funzionale per mettere ogni bit di un dato segnale A in XOR con un bit C

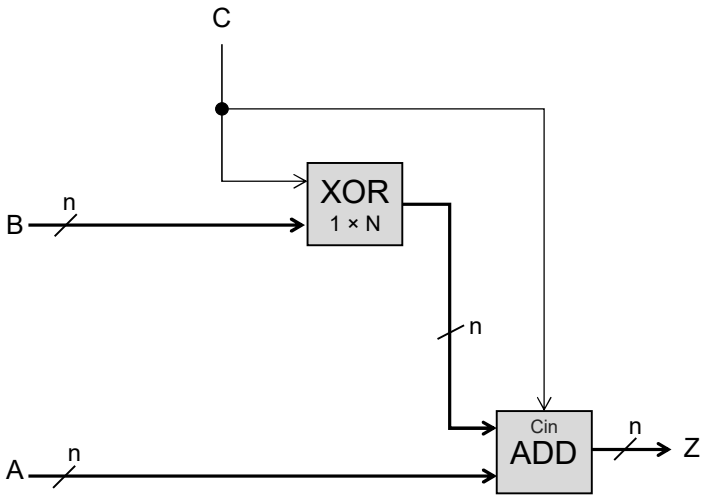
Realizzazione (banale!):
(qui, per N = 4)

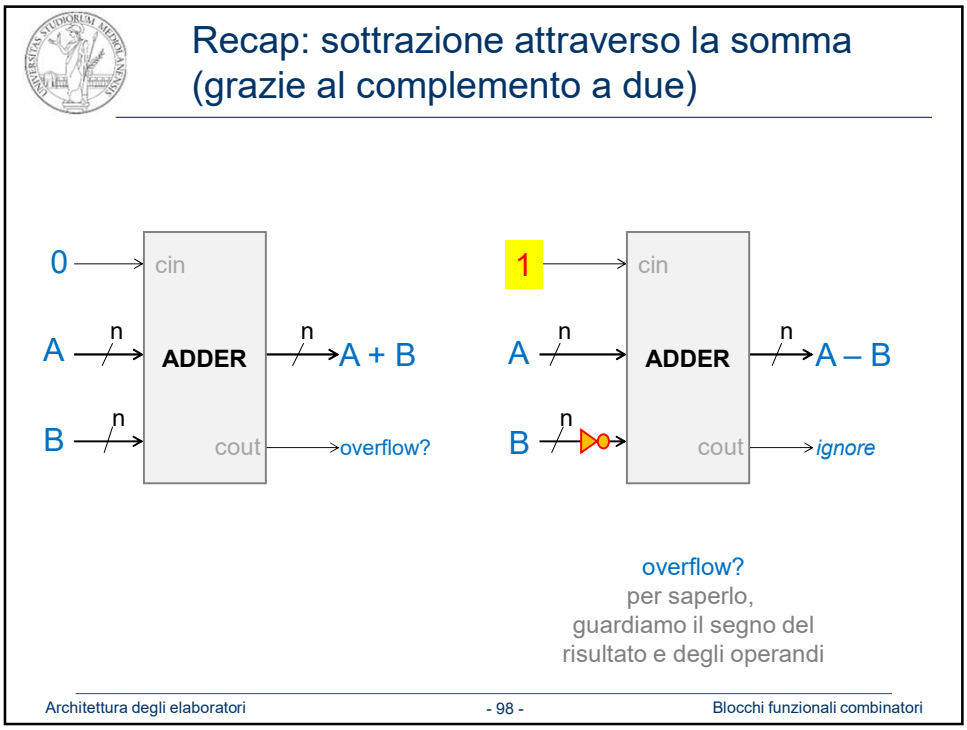




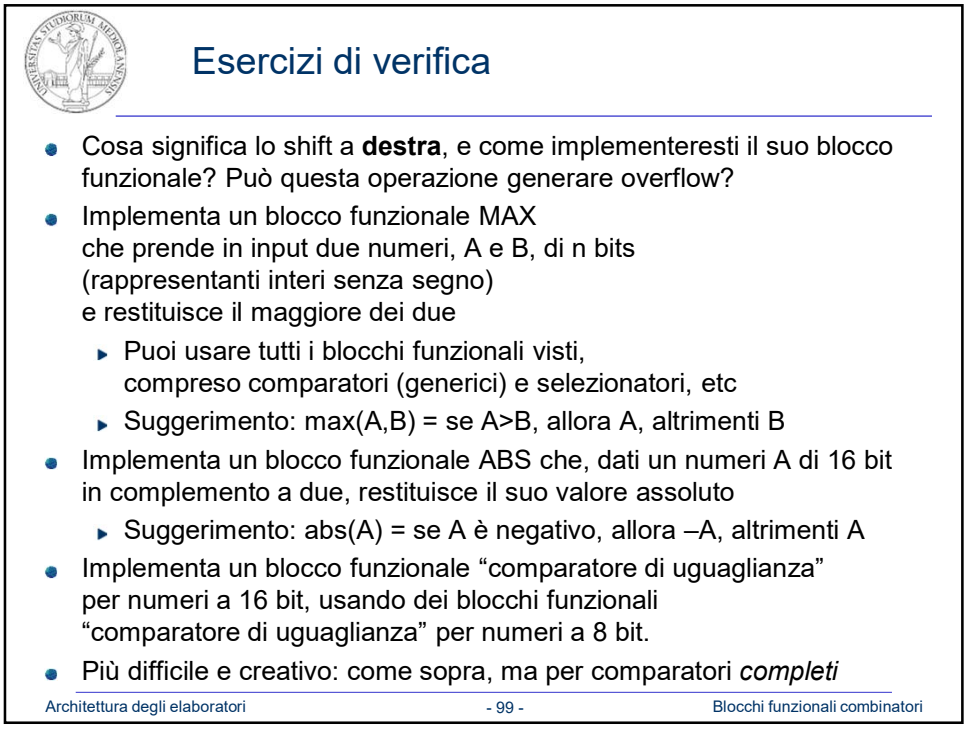
è un blocco funzionale per
invertire tutto A, quando il bit C vale 1,
e lasciare tutto A inalterato, quando C vale 0

Soluzione 4 (ottimizzazione di 3)




Recap: sottrazione attraverso la somma (grazie al complemento a due)
0
cin
A
n
ADDER
n
A + B
B
n
cout
overflow?
1
cin
A
n
ADDER
n
A – B
B
n
cout
ignore
overflow?
per saperlo,
guardiamo il segno del
risultato e degli operandi
Architettura degli elaboratori
- 98 -
Blocchi funzionali combinatori

98


Esercizi di verifica
Cosa significa lo shift a **destra**, e come implementeresti il suo blocco funzionale? Può questa operazione generare overflow?
Implementa un blocco funzionale MAX
che prende in input due numeri, A e B, di n bits
(rappresentanti interi senza segno)
e restituisce il maggiore dei due
Puoi usare tutti i blocchi funzionali visti,
compreso comparatori (generici) e selezionatori, etc
Suggerimento: max(A,B) = se A>B, allora A, altrimenti B
Implementa un blocco funzionale ABS che, dati un numeri A di 16 bit
in complemento a due, restituisce il suo valore assoluto
Suggerimento: abs(A) = se A è negativo, allora –A, altrimenti A
Implementa un blocco funzionale “comparatore di uguaglianza”
per numeri a 16 bit, usando dei blocchi funzionali
“comparatore di uguaglianza” per numeri a 8 bit.
Più difficile e creativo: come sopra, ma per comparatori *completi*
Architettura degli elaboratori
- 99 -
Blocchi funzionali combinatori

99

Circuiti per operazioni in virgola mobile (cenni)

- Sono circuiti combinatori più complessi. Tipicamente:
  - sottocircuiti separati per calcolo di: segno, esponente, mantissa
  - inoltre (per molte op): circuito finale per rinormalizzazione del risultato
  - casi speciali per gestire configurazioni speciali (attraverso dei MUX)
- Cenni sulle operazioni più comuni:
  - Comparazione: possiamo ri-usare il comparatore dei CP2 (come mai?)
    (più gestione configurazioni speciali)
  - Somma: exp: il maggiore dei due exp.
    mantissa e segno: somma (con segno) delle 2 mantisse shiftate.
    Rinormalizzazione necessaria alla fine.
  - Prodotto: exp: somma dei due exp.
    mantissa: prodotto delle due mantisse.
    segno: XOR dei due segni (come mai?)
  - Inverso (1/x): Exp: opposto. Sgno: mantenuto. Mantissa: inverso
  - Divisone : ottenibile combinando inversione e prodotto