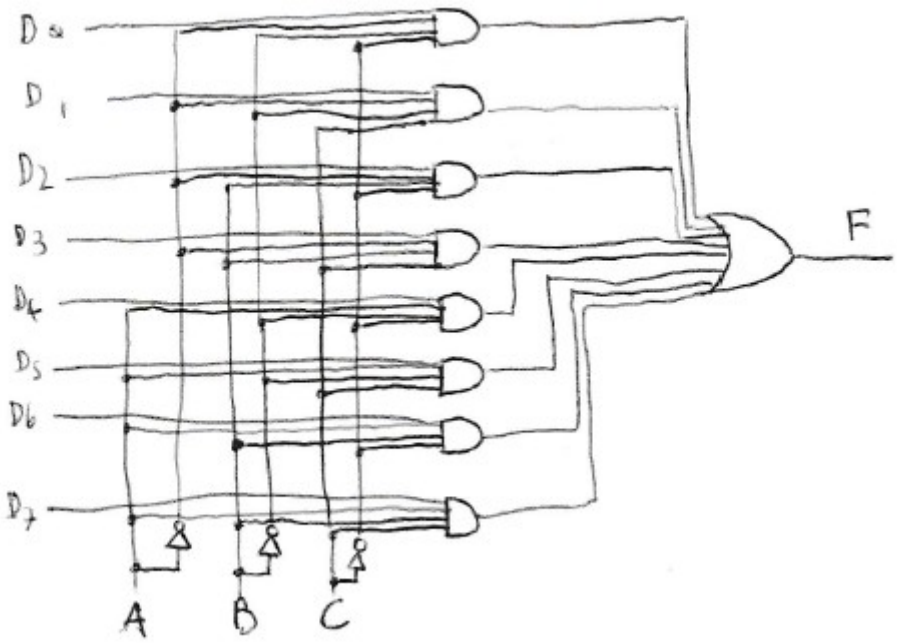
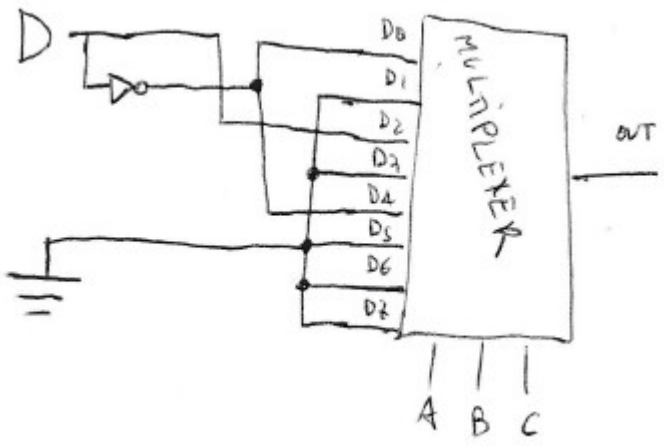
Esercitazione 2 – Architettura dei Sistemi di Elaborazione – 19/12/2016
Soluzioni

1. **Si costruisca un circuito multiplexer con 8 dati in input, un output e 3 input di controllo, che sia effettivamente in grado di calcolare il valore di verità di una funzione booleana a *quattro* variabili.**
   La funzione da calcolare è la seguente: $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C}\,D + A\,\overline{B}\,\overline{C}\,\overline{D}$.

È noto che un multiplexer a 3 input di controllo può rappresentare qualsiasi tavola di verità di una funzione booleana a 3 variabili. Per forzarlo a calcolare una funzione booleana a *quattro* variabili si deve conoscere la sua struttura interna, illustrata in figura. Sul libro (**Architettura dei calcolatori**, Tanenbaum), lo trovate disegnato meglio a pagina 144.

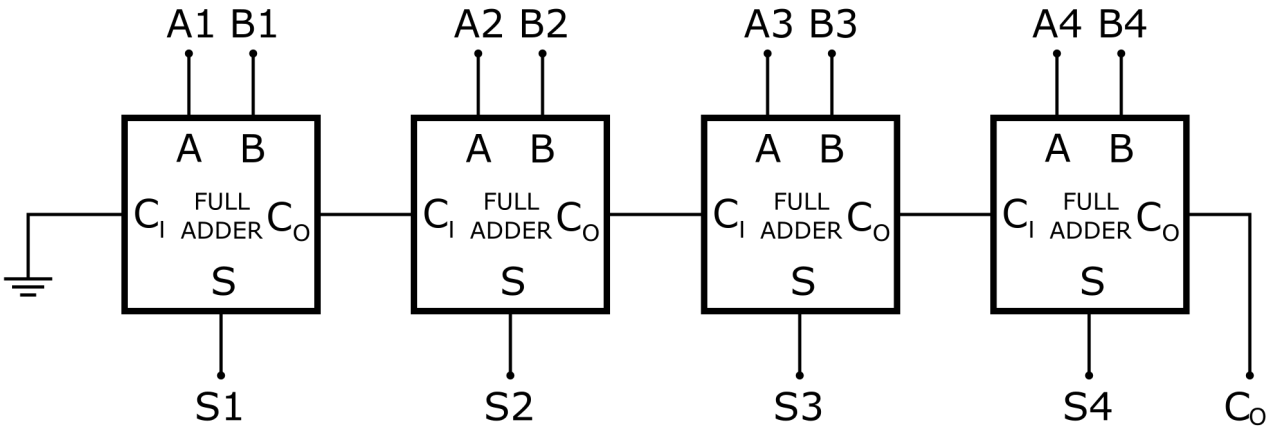


La soluzione al quesito è semplice, basta inserire all'interno del circuito la variabile D (e il suo corrispettivo $\overline{D}$) e collegarla agli input corretti, in modo tale da soddisfare la funzione richiesta. Tutti gli altri input, a questo punto, vanno collegati a terra. Quindi vogliamo collegare D all'ingresso D_2, corrispondente alla porta relativa a $\overline{A}\,B\,\overline{C}$, e $\overline{D}$ agli ingressi D_0 e D_4, corrispondenti alle porte relative a $\overline{A}\,\overline{B}\,\overline{C}$ e $A\,\overline{B}\,\overline{C}$. Il risultato sarà, dunque, quello illustrato in figura.

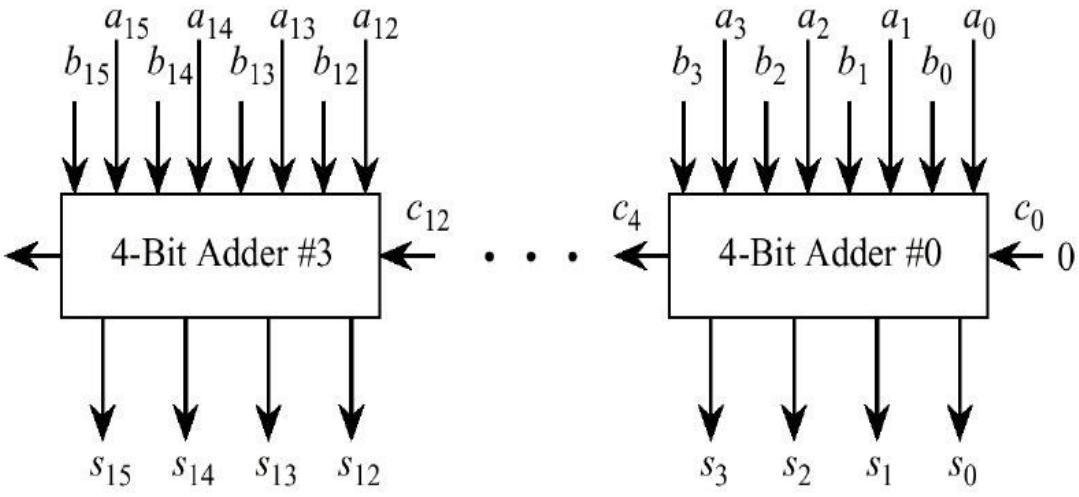


2. **Un chip MSI molto comune è il sommatore a 4 bit. È possibile agganciare quattro di questi chip per ottenere un sommatore a 16 bit? Disegnarlo, se possibile. Quanti pin avrà il nuovo sommatore?**

Come si può vedere in figura, l'adder a 4 bit altro non è che la giustapposizione di 4 circuiti full-adder a un bit (sul libro, si tratta di questo argomento alle pagine 149-150).

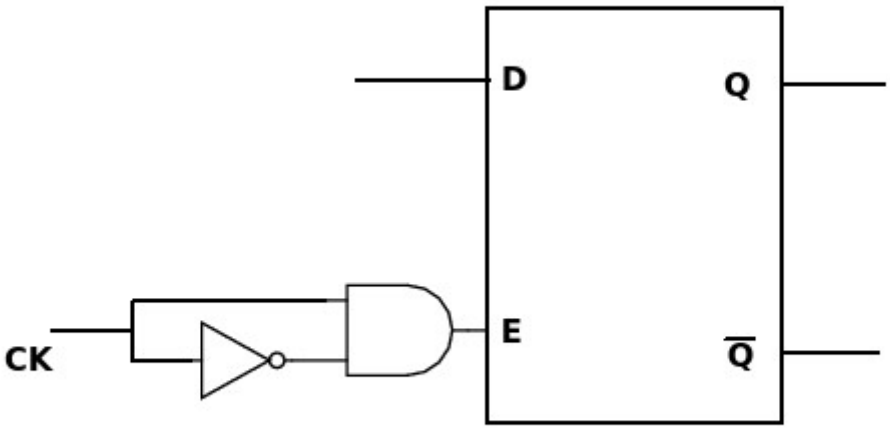


I valori in ingresso sono indicati come le coppie di 4 pin A_1, A_2, A_3, A_4 e B_1, B_2, B_3, B4, che rappresentano i bit dal meno al più significativo dei valori A e B. Gli output sono i 4 bit risultanti e marcati con le tag S_1, …, S_4. I riporti, invece sono rappresentati dalle tag C_I per quello in ingresso e C_O per quello in uscita.
Ora, è ovviamente possibile applicare la stessa idea usata per creare un sommatore a 4 bit per crearne uno a 16 bit. Il numero totale di pin finale è facilmente calcolabile: ne occorrono 32 per i dati in input, 16 per la somma in output, 1 per il riporto in ingresso iniziale (collegato, come nella figura sovrastante, alla terra) e 1 per il riporto in uscita finale. La somma è pari a 50. Il circuito risultante è mostrato in figura.

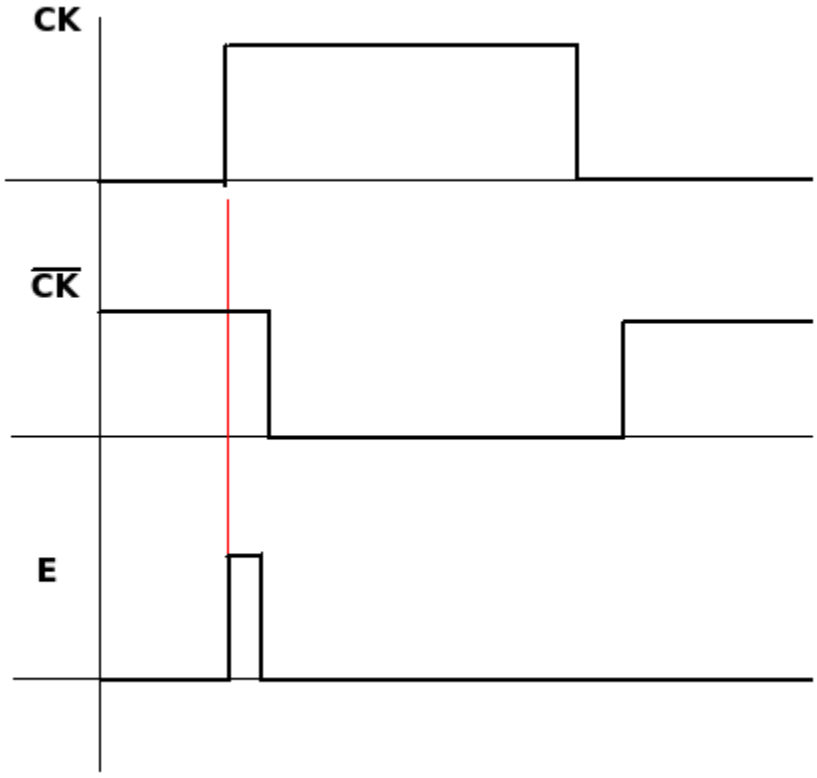


3. **Si studi un circuito flip-flop pilotato dal fronte di salita del clock. Lo si modifichi in modo tale che sia pilotato dal fronte di discesa del clock.**

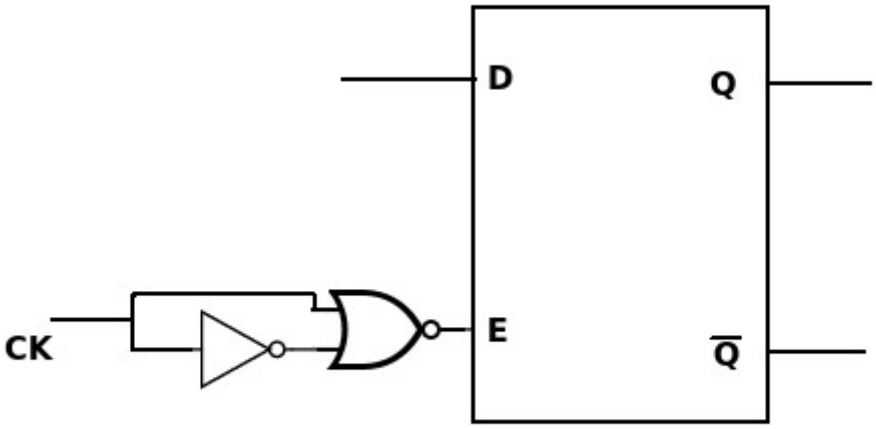
In primo luogo, vi illustro il flip-flop pilotato dalla salita del clock. Per tutti gli aspetti teorici relativi al funzionamento e all'utilizzo del flip-flop, vi consiglio di vedere, sempre sul Tanenbaum, la pagine 157 e successive. Considerate che, nella domanda, si fa riferimento al flip-flop temporizzato, indicato sul libro come flip-flop D. Il circuito è mostrato in figura.



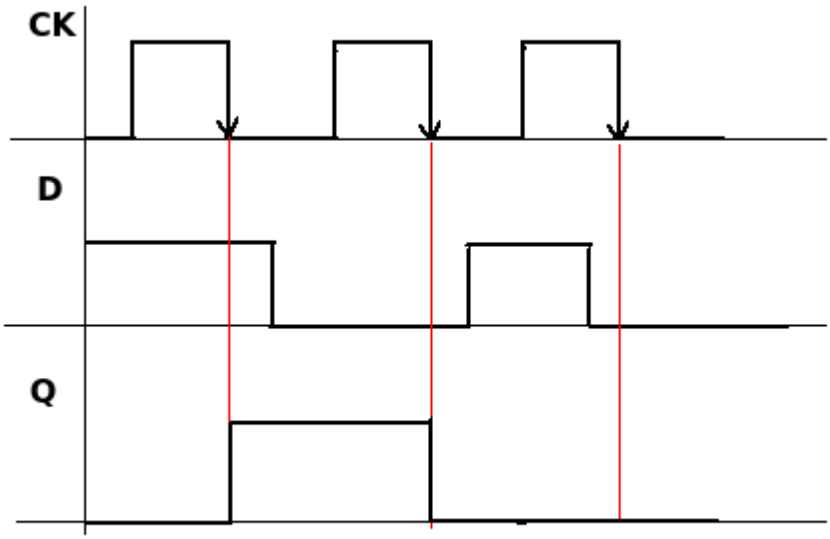
Ora, può sembrare che la porta E riceva sempre un segnale negativo, ma questo non è completamente vero, grazie al ritardo nella propagazione del segnale causato dalla presenza della porta NOT. La porta E, quindi, avrà valore 1 per un breve istante nel momento in cui CK (il clock) passa dal valore 0 al valore 1 (fronte di salita del clock).
Il funzionamento è mostrato qui sotto.



Ora, quello che viene richiesto, dunque, è che la porta E riceva il segnale alto quando CK passa da 1 a 0, invece che da 0 a 1. La soluzione è abbastanza semplice: si vuole che E abbia valore 1 esclusivamente quando sia CK che $\overline{CK}$ (opportunamente ritardato) abbiano valore 0. Intuitivamente, è facile vedere (anche dalla tavola di verità) che la porta logica che fa al caso nostro è una porta NOR. Il circuito risultante, dunque, è quello illustrato in figura.



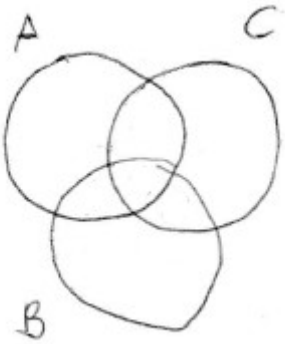
Questa seconda figura mostra l'andamento dei segnali, giusto per verificare che quanto abbiamo detto sia corretto.



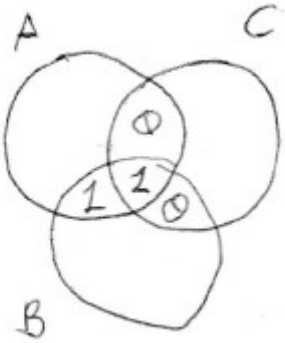
4. **Si ha una parola a 4 bit, 1100. Utilizzando il codice d'errore basato sulla distanza di Hamming, quanti e quali sono i bit del codice di correzione corrispondente? Quanti sarebbero per una parola a 8 bit?**

Cominciamo col dire che tutta la teoria relativa a questa domanda la potete trovare alle pagine 70 e seguenti del Tanenbaum (paragrafo 2.2.4.). Sul libro trovate esattamente l'esempio che vi ho richiesto in questo esercizio. L'algoritmo per determinare i 3 bit di correzione (che chiameremo A, B e C) è il seguente:

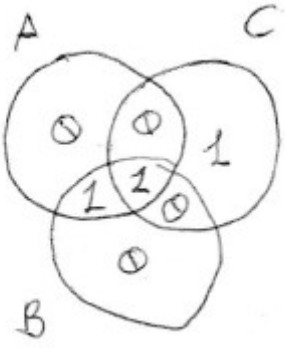
- Disegnare tre cerchi intersecati, uno per ogni bit di correzione.



- Ordinare le intersezioni per ordine alfabetico (o lessicografico, se preferite), ottenendo dunque AB, ABC, AC, BC.
- Scrivere, un bit alla volta, la parola all'interno delle intersezioni, seguendo l'ordine determinato al punto precedente.



- Scrivere, all'interno della parte non intersecata dei tre cerchi, il valore 0 o 1, in modo tale che il numero di 1 all'interno del singolo cerchio sia pari. Nel nostro caso A ha due 1 al suo interno, quindi scriveremo 0; B ne ha sempre due, quindi scriveremo 0; C ha un solo 1 al suo interno e quindi si scrive un 1.



Pertanto i bit d'errore saranno 001. Nel caso in cui abbiamo 8 bit, invece, possiamo usare l'algoritmo corrispondente per creare un codice di correzione a 7 bit, ma questo valore può essere ridotto fino a 4 bit.

5. **Si scriva in notazione polacca inversa le seguenti espressioni**
   - **6*(4-3)**
   - **(7/3)/((1-4)*2)+1**
   - **(5*2+7)-4/2+1**

La notazione polacca inversa è piuttosto semplice: invece di scrivere nella maniera classica OPERANDO1 OPERAZIONE OPERANDO2 (esempio: 2 + 3), si riordinano gli elementi secondo la struttura OPERANDO1 OPERANDO2 OPERAZIONE. Per comodità, d'ora in poi chiameremo gli elementi OPERANDO1 OP1, OPERANDO2 OP2 e OPERAZIONE OP.
Si guardano le espressioni da sinistra a destra, andando a vedere quale operazione va svolta per prima. Nel primo caso, ad esempio avremo OP1 = 6, OP2 = (4-3), OP = *. OP2 va ancora diviso secondo la stessa regola: $OP1^1 = 4$, $OP2^1 = 3$, $OP^1 = -$. Scriveremo quindi 6 4 3 - *
Le altre soluzioni alla domanda, dunque, sono:

- 73/14-2*/1+
- 52*7+4-21+/

6. **Scrivere le istruzioni Assembly utili per ottimizzare l'utilizzo della CPU per il calcolo della della moltiplicazione tra il numero 19 e un valore *n* intero e non negativo. Verificare , ponendo *n*=3, che l'operazione proposta sia corretta.**

Comincio con l'indicarvi a quale parte del libro dovete fare riferimento per rispondere a questa domanda: le pagine da considerare sono la 367 e seguenti, e in particolar modo il paragrafo 5.5.3, Operazioni Unarie.
In questo paragrafo è ben spiegato che la moltiplicazione tra un qualsiasi numero *n* e un numero come 2^k, può essere ottimizzato come uno shift del numero verso sinistra di k cifre. Ad esempio, 3 (rappresentazione binaria a 8 bit 00000011), può essere moltiplicato per 2 ($2=2^1$), shiftandolo di 1 bit verso sinistra, ottenendo, quindi, 00000110, che equivale proprio al numero 6 in base 10. Nel caso dell'esercizio, abbiamo il numero 19, che può essere scritto come somme successive di numeri del tipo 2^k. Abbiamo dunque 19 = 16 + 2 + 1, cioè $19 = 2^4 + 2^1 + 2^0$. Quindi i risultati dei tre shift andranno sommati tra loro, per fornirci il risultato effettivo.
Le istruzioni – in pseuo Assembly, visto che il linguaggio cambia a seconda del processore che si sta utilizzando – saranno le seguenti:

- Caricamento del valore *n* in memoria tre volte, pari al numero di elementi che vogliamo prima shiftare e poi sommare. L'istruzione di riferimento è la MOV, che riceve come argomenti, nell'ordine, il registro su cui caricare il valore e il valore stesso.

```
MOV  R1    $n
MOV  R2    $n
MOV  R3    $n
```

- Shifting a sinistra dei tre valori presenti nei registri. L'operazione di riferimento è la SHL, che prende in ingresso, nell'ordine, il numero di bit per cui si vuole effettuare lo shift e il registro su cui lo si vuole effettuare.

```
SHL   4    R1
SHL   1    R2
```

  Come è facile vedere, non è necessario scrivere l'istruzione per lo shifting di 0 bit verso sinistra per il registro R3.
- Somma dei tre valori. L'operazione da usare è la ADD, che somma il valore nel primo registro in input al secondo, ponendo il risultato dell'operazione all'interno del secondo registro in input.

```
ADD   R2      R1
ADD   R3      R1
```

Al termine della procedura, avremo il risultato della moltiplicazione originale all'interno del registro R1.
Verifichiamo ora che la procedura sia corretta, valorizzando *n* con il valore 3.
Il risultato delle tre operazioni MOV è il seguente:

Registro	**Valore**
R1	00000011
R2	00000011
R3	00000011

Il risultato delle due operazioni shift è:

Registro	**Valore**
R1	00110000
R2	00000110
R3	00000011

Il risultato della prima somma binaria è:

Registro	**Valore**
R1	00110110
R2	00000110
R3	00000011

Il risultato della somma finale è:

Registro	**Valore**
R1	00111001
R2	00000110
R3	00000011

In R1 abbiamo il valore risultante, cioè $(00111001)_2$, che equivale a $(57)_{10}$. E casualmente 57 = 19*3