Architettura degli elaboratori - lezione 10

Appunti di Davide Vella 2024/2025

Claudio Schifanella
claudio.schifanella@unito.it
Link al moodle :
https://informatica.i-learn.unito.it/course/view.php?id=3106

07/04/2025

Contenuti

Overflow

Per capire se c'é stato un overflow, ci sono due modi :

- Se la somma di due numeri positivi fanno un numero negativo. Se la somma di due numeri negativi fanno un numero positivo.
- Se l'ultimo riporto ottenuto è discorde al penultimo riporto ottenuto.

Bnegate

Come visto nella scorsa lezione, ogni volta che vogliamo che l'ALU esegua sottrazioni, dobbiamo asserire (porre a 1) sia CarryIn sia Binvert

Operazione	Funzione	Binvert	CarryIn0
0	AND	0	0
1	OR	0	0
2	ADD	0	0
2	SUB	1	1
3	SLT	1	1

Notando che Binvert è ad 1, quando CarryIn0 (ovvero il CarryIn della prima ALU a 1 bit) è ad 1, possiamo creare una sola linea che si chiama Bnegate, per poter risparmiare una linea (la quale porta ad una minore grandezza del bus, meno pin in un chip, meno costi...).

Beq, funzionamento

Beq realizza un salto se due registri sono uguali. Avviene quindi una sottrazione tra i due registri e se il risultato è uguale a 0, avviene il salto, altrimenti no.

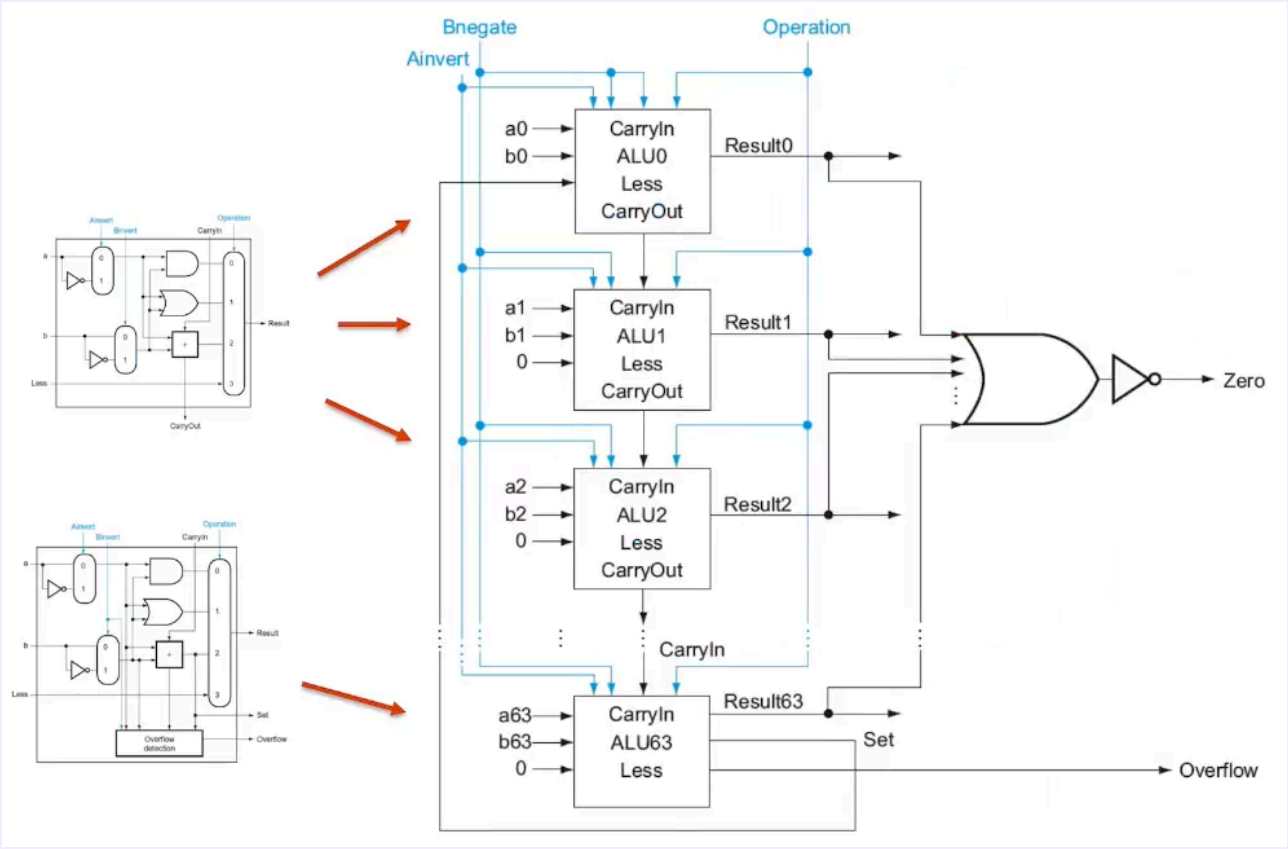
$$(a - b) == 0$$

L'operazione sopra viene eseguita nel seguente ordine :

1. Complemento a 1 di b
2. Aggiungo 1 a b (adesso ho - b)
3. Eseguo la somma tra 'a' e '-b'
4. Adesso ho il risultato. Devo controllarlo. Se il risultato vale 0, dobbiamo saltare, altrimenti no. Per far saltare però, dobbiamo dare come output '1'.
5. Neghiamo il risultato
6. Facciamo l'or. Se c'é un 1, vuol dire che $a == b$

> **Example**
>
> Come possiamo vedere sotto (una possibile soluzione), prendiamo l'or del risultato di tutti i bit. Se tutti valgono 0, allora passa 0 dalla OR e diventa 1 con la NOT e il risultato è zero = 1 (zero = 1 se i due numeri sono uguali!), altrimenti ogni altro valore di un result diverso da

zero farà deventare zero = 0.



Controllo operazioni ALU

Per riassumere, nella nostra ALU avremo :

- 2 bit per il controllo dell'operazione (operation).
- 1 bit per l'ingresso di Ainvert (che usiamo per fare la nor).
- 1 bit per l'ingresso di Binvert (che usiamo per fare una sottrazione, slt e nor).

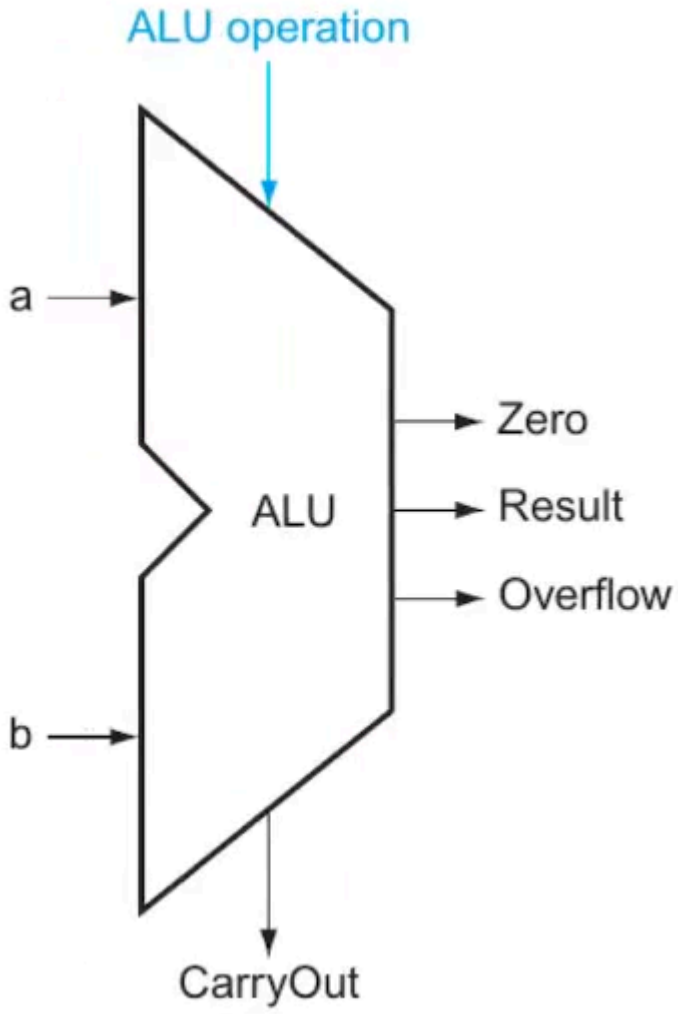
Ainvert	Bnegate	Operation
0	0	00
0	0	01
0	0	10
0	1	10
0	1	11
1	1	00

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

E ancora avremo :

- 32 bit di ingresso per A (architettura a 32 bit)
- 32 bit di ingresso per B
- 32 bit di uscita per result

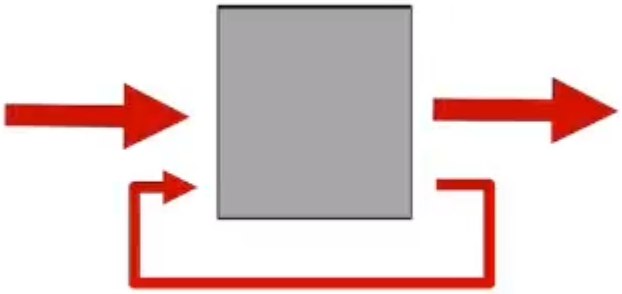
- 1 bit per zero
- 1 bit per overflow
- 1 bit per il CarryOut



Circuiti sequenziali

I circuiti sequenziali sono dispositivi in grado di calcolare funzioni che dipendono anche da uno stato interno, quindi dipendono anche da informazioni memorizzate in elementi di memoria interni. In generale, la funzione calcolata dal circuito ad un dato istante dipende dalla sequenza

temporale dei valori in input al circuito.



Latch

Latch, un dispositivo a 1 bit che ha due ingressi (i e β) ed un'uscita (Q). Mantiene uno stato interno (s). La logica del latch è la seguente, se β = 0, lo stato futuro è uguale allo stato passato (non viene sovrascritto il bit salvato). Invece, se β = 1, lo stato futuro varia in base all'input i. Quindi :

- b = 0 : hold, si mantiene il contenuto della memoria
- b = 1 : store, si riscrivere il contenuto della memoria

Stato corrente Stato futuro

β	i	s	s'=o
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

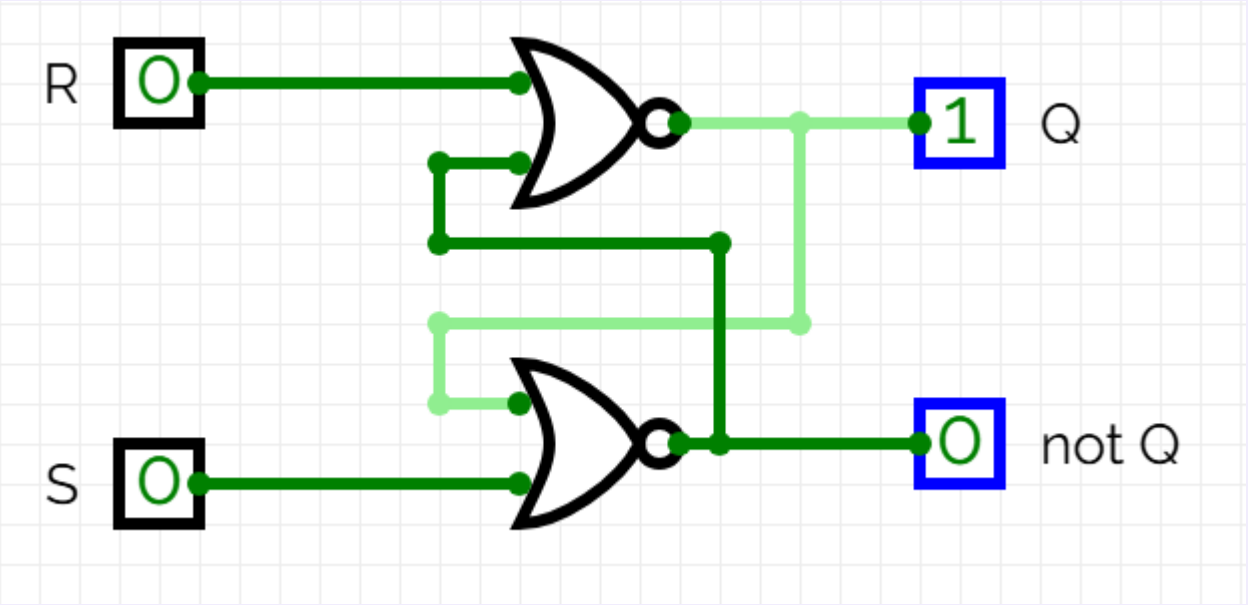
Latch SR

Latch SR (set-reset). Logicamente simile al latch visto sopra. Le fasi sono :

- Hold, quando R = S = 0, allora si mantiene lo stato salvato all'interno
- Set (store 1), S = 1 e R = 0, porta il latch allo stato 1
- Reset (store 0), R = 1 e S = 0, porta il latch allo stato 0

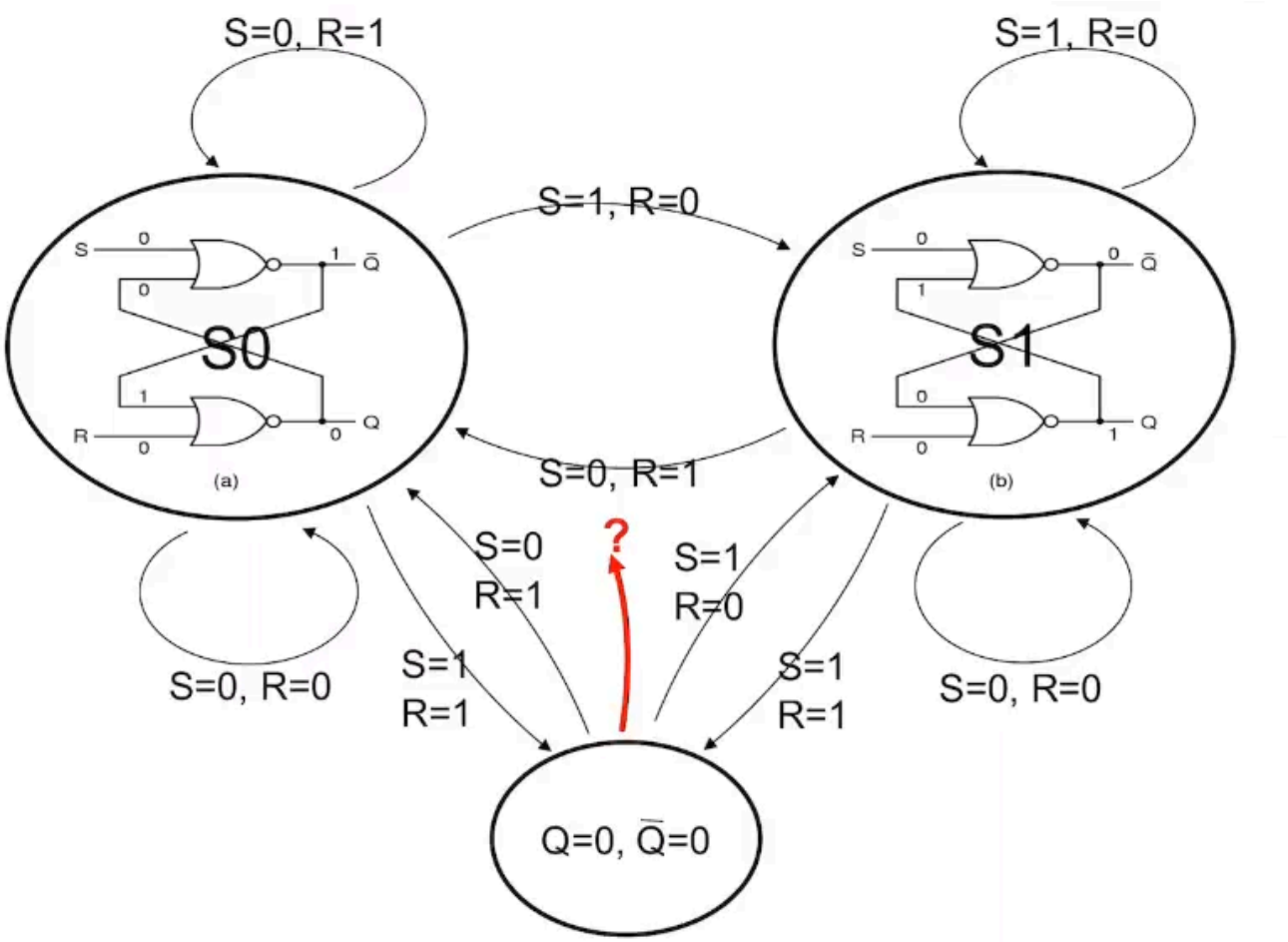
Example

Sotto un latch SR con stato salvato 1 e in fase di HOLD



Avere S = R = 1 è una cosa che non dovrebbe mai accadere. Lo stato non è stabile, per tanto Q diventa impredicibile e potrebbe nascere un'oscillazione.

Automa a stati finiti di Latch SR



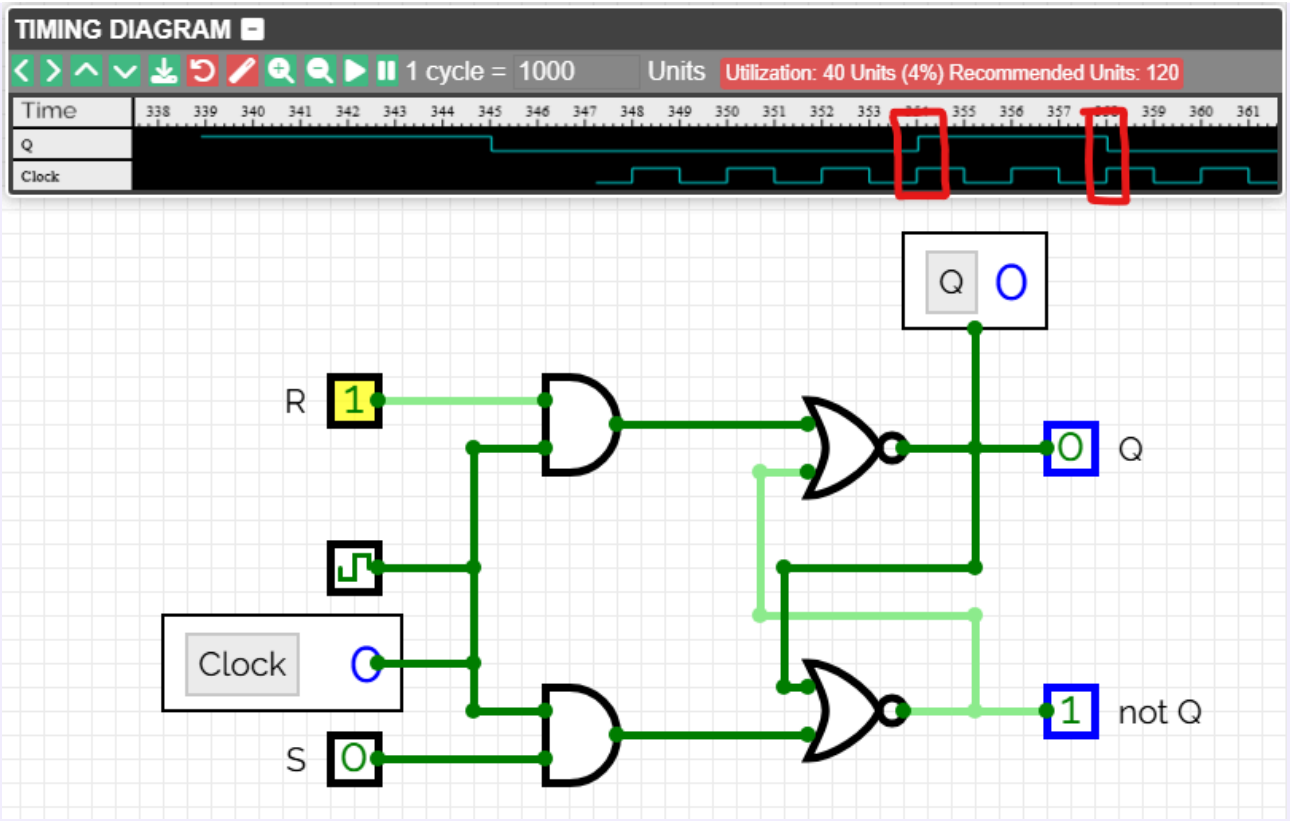
La freccia rossa indica il momento in cui passiamo da R = S = 1 a R = S = 0, cosa che rende appunto impredicibile lo stato e quindi che non deve accadere.
Visto che non dobbiamo mai arrivare ad avere S = R = 1, dobbiamo cercare una soluzione in cui dobbiamo salvare il risultato di S e R sì, ma solo quando il loro segnale è stabile (S e R sono calcolati da circuiti combinatori, è possibile che il loro segnale non sia immediatamente stabile per varie ragioni), questa soluzione è il **clock**.

Latch SR sincronizzato

Un clock garantisce che il latch cambi stato solo in certi momenti specifici (momenti scelti cambiando il circuito. Si può decidere di salvare in fronte di salita, in discesa, in stato alto o stato basso). Aggiungendo quindi un clock e delle porte AND, possiamo fare in modo che i dati vengono salvati solo durante lo stato alto :

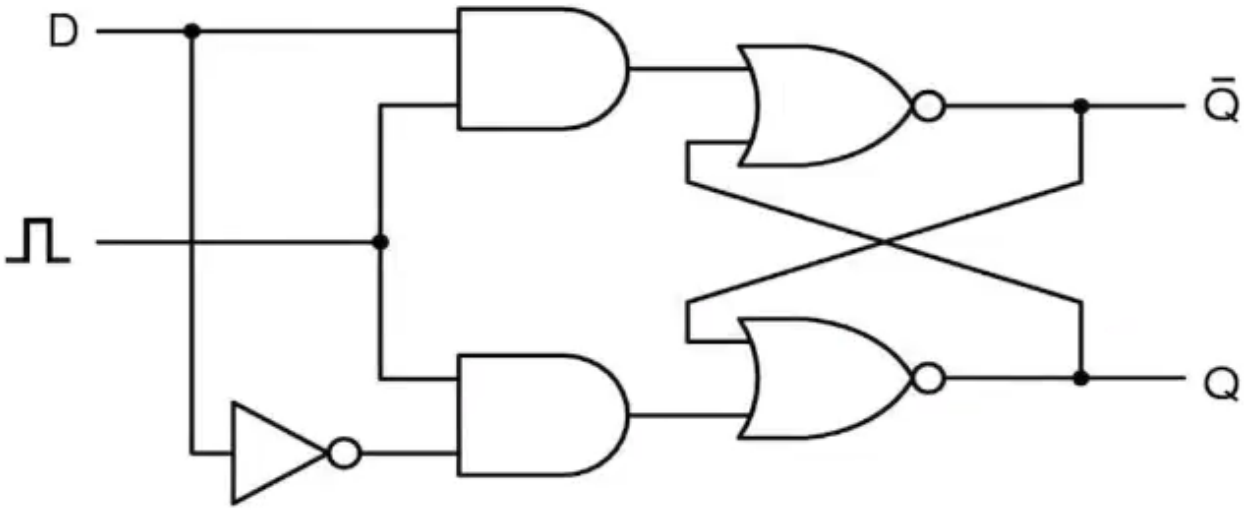
Example

Di seguito un esempio realizzato con CircuitVerse. Possiamo notare sopra che il cambiamento in Q viene eseguito solo quando il Clock è HIGH (evidenziato in rosso).



Latch di tipo D sincronizzato

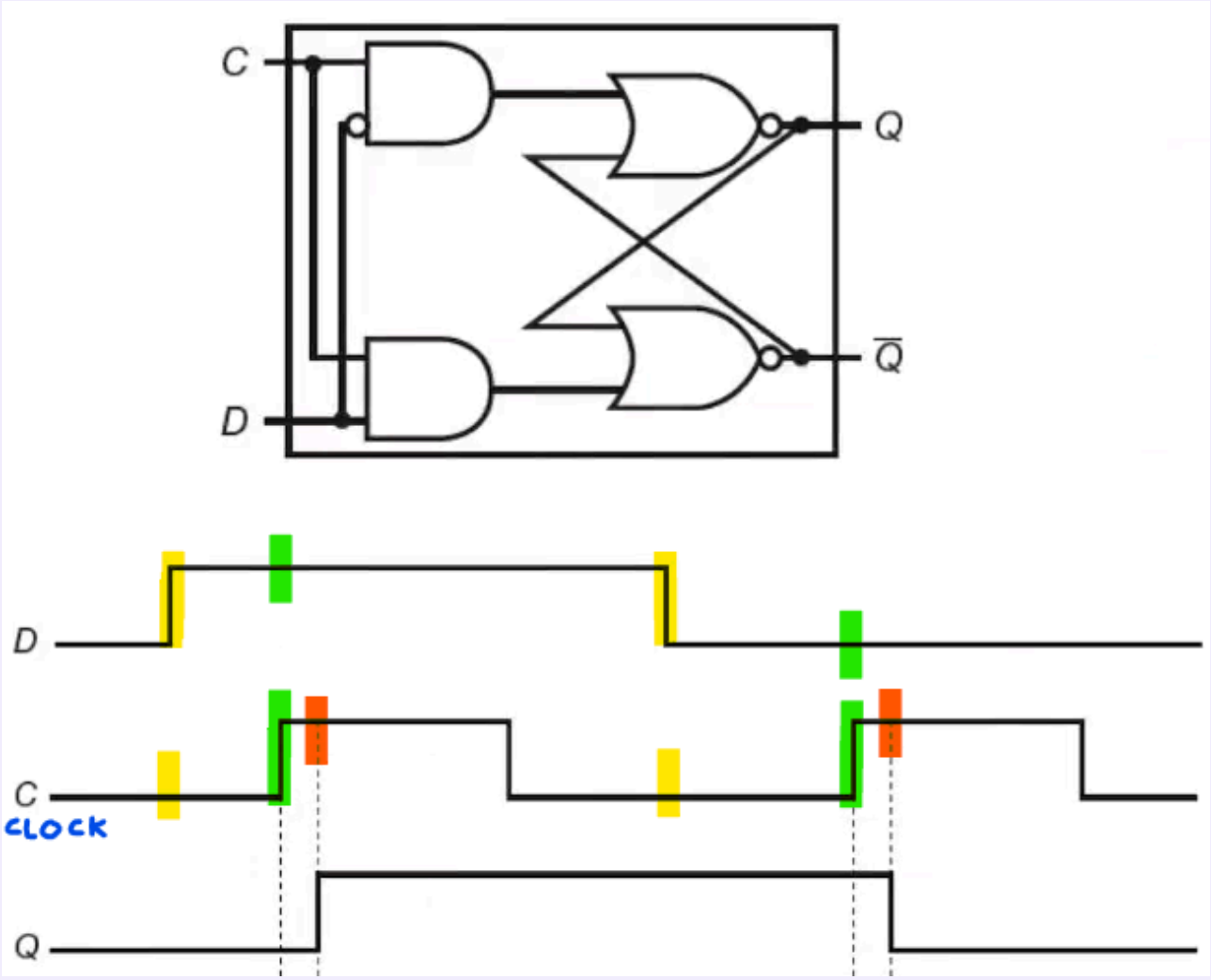
Il latch di tipo D sincronizzato è un miglioramento del latch SR. Il latch di tipo D rimuove il problema di poter avere sia R che S ad 1 contemporaneamente, come? Semplicemente avendo un input "D" che va in S e che va in maniera negata in R. Essendo sincronizzato, presenta anche lui il clock.



Con lo schema sopra, se D varia mentre il clock è 1 (HIGH), allora varia anche lo stato. Se D varia mentre il clock è 0, il primo cambiamento verrà salvato quando il clock torna ad 1.

Example

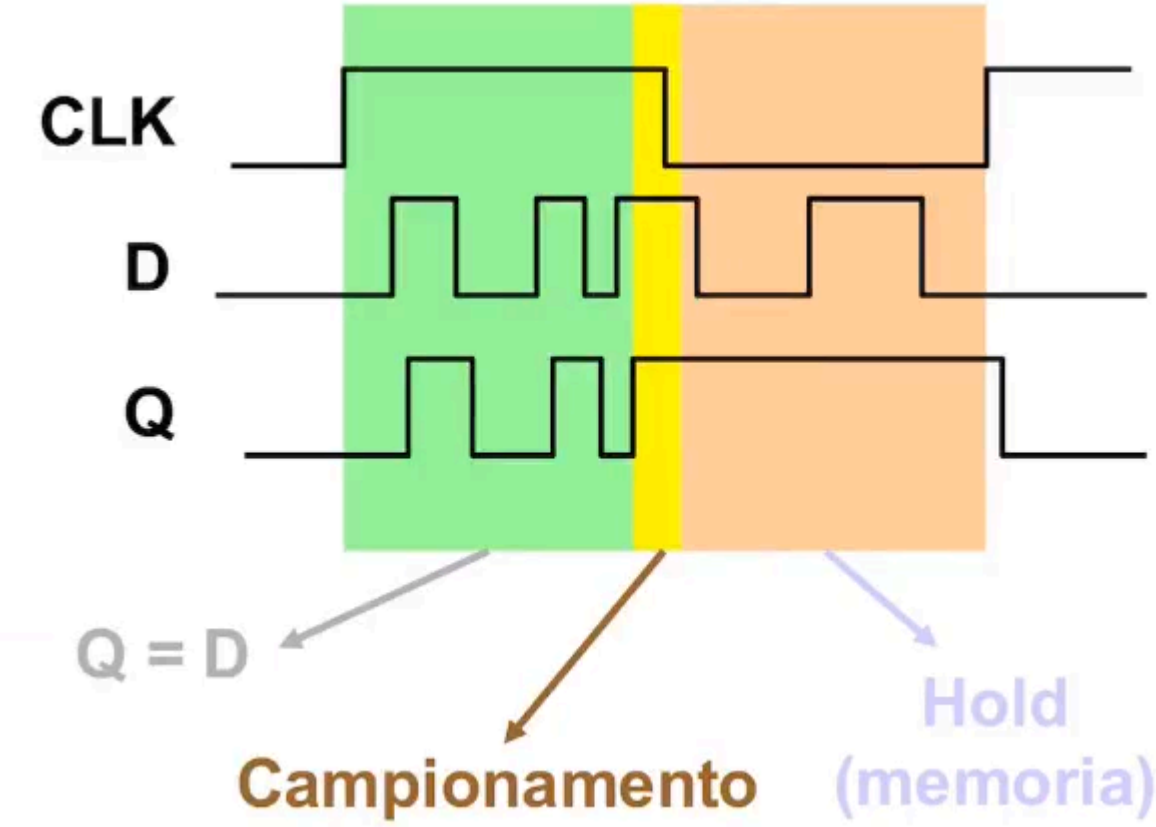
Qui un esempio. Troviamo in giallo il cambiamento di D, in verde lo stato 1 del clock nel quale verrà salvato il cambiamento di D in Q e in rosso il momento effettivo in cui viene cambiato Q.



Come si può notare, c'é una discrepanza tra il momento in cui il clock sale e il momento di salvataggio. Questa discrepanza è dovuta dal ritardo delle porte. Quindi nella parti in cui il clock è basso, il circuito calcola cosa memorizzare e quando il clock è alto, memorizza quello che ha precedentemente calcolato.

Problema della "trasparenza" (importante)

Con il clock = 1, il latch è "trasparente", ovvero, l'uscita segue l'ingresso istante per instate (con un leggere ritardo dovuto, come detto prima, al tempo di propagazione attraverso il latch).



Qual è il problema quindi? Se l'ingresso cambia mentre il latch è abilitato, anche l'uscita cambia immediatamente. Questo può causare **comportamenti imprevisti o instabilità**, soprattutto se l'uscita del latch influenza altri circuiti sensibili ai cambiamenti.

Example

Pensiamo alla seguente istruzione :

```
add x1, x1, x4
```

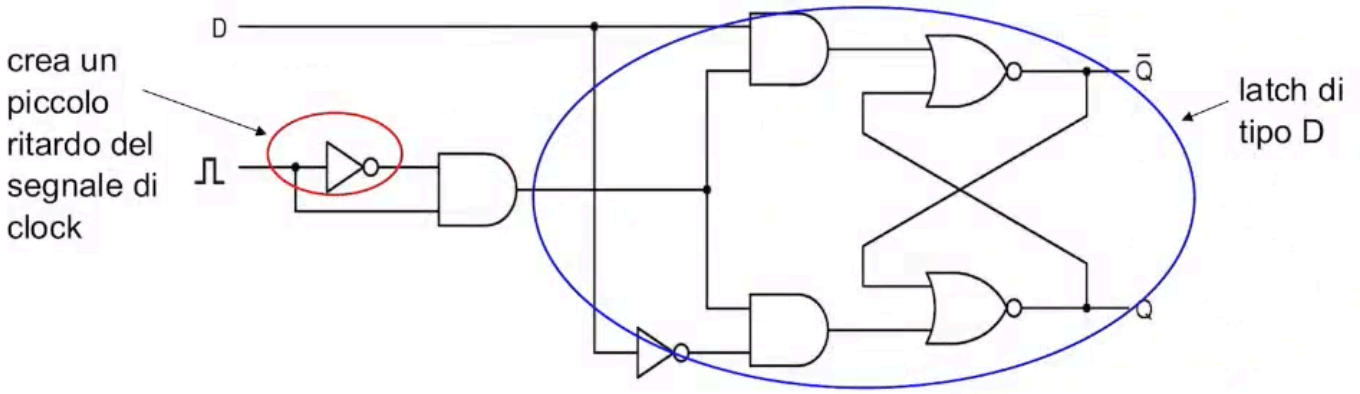
Qui prendiamo il valore di x1, lo sommiamo ad x4. Il risultato poi viene salvato in x1, il quale però può ancora tornare indietro creando un problema.

Come si può risolvere? Per evitare questo problema si preferisce usare **flip-flop edge-triggered**, che aggiornano l'uscita **solo sul fronte di salita o discesa del clock**, garantendo un comportamento più stabile e prevedibile nei sistemi digitali sincroni.

Flip-Flop di tipo D

Si aggiunge una porta NOT al segnale di clock e si sdoppia l'uscita del clock. Questi due segnali, vengono messi in AND. La porta NOT metterà ad 1 il segnale quando il clock vale 0. Questo per pochissimo tempo anche quando il valore del clock vale 1, perché c'é quel ritardo di

cui parlavamo prima. Questo fa si che la porta AND faccia passare 1 per un breve lasso di tempo.



In questo breve lasso di tempo, vogliamo salvare il risultato.
Nella condizione ottimale, il tempo in cui avviene il salvataggio dovrebbe essere solo 1, ma fisicamente è impossibile arrivare a questa ottimalità, quindi, si cerca di tenere questo tempo il più basso possibile.

Important

Da ricordare bene. Il Flip-Flop D visto, salva SOLO sul fronte di salita.