Logica Digitale

Logica combinatoria e sequenziale

- Una CPU è composta fondamentalmente da due classi diverse di circuiti digitali
  - Combinatori → l'uscita dipende solo dal valore degli ingressi istante per istante
    - es. Operazioni aritmetiche
  - Sequenziali → L'uscita dipende anche da uno **stato** interno del circuito
    - es. Memorie, registri
- Entrambi si possono realizzare con gli stessi elementi base, ovvero **porte logiche**. Il loro funzionamento si basa sull'algebra di Boole.

Algebra di Boole

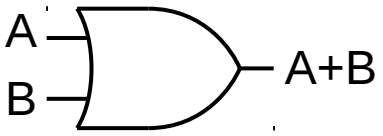
- Algebra di riferimento per lo studio di circuiti logici digitali
- Un'espressione booleana è isomorfa ad un circuito digitale
  - un circuito digitale può essere espresso univocamente tramite un'espressione booleana e viceversa
- È definita da:
  - Un insieme di 2 valori
    - {0, 1}, {vero, falso}, {alto, basso}, …

    } Rappresentabile con 1 bit
  - 3 operazioni elementari:
    - OR → congiunzione, `A+B=1` se `A=1` *oppure* `B=1`
    - AND → disgiunzione, `A•B=1` se `A=1` *e* `B=1`
    - NOT → negazione $\overline{A}=1$ se `A=0`, $\overline{A}=0$ se `A=1`

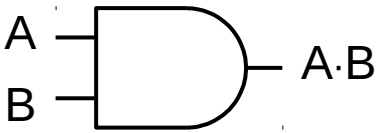
    } Equivalenti alle operazioni bitwise

Porte logiche elementari

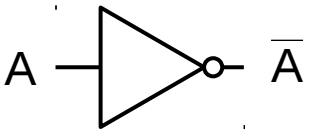
- Circuiti digitali che implementano le operazioni booleane elementari



OR		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1



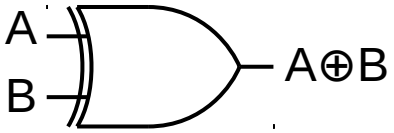
AND		
A	B	A·B
0	0	0
0	1	0
1	0	0
1	1	1



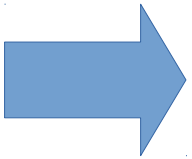
NOT	
A	$\overline{A}$
0	1
1	0

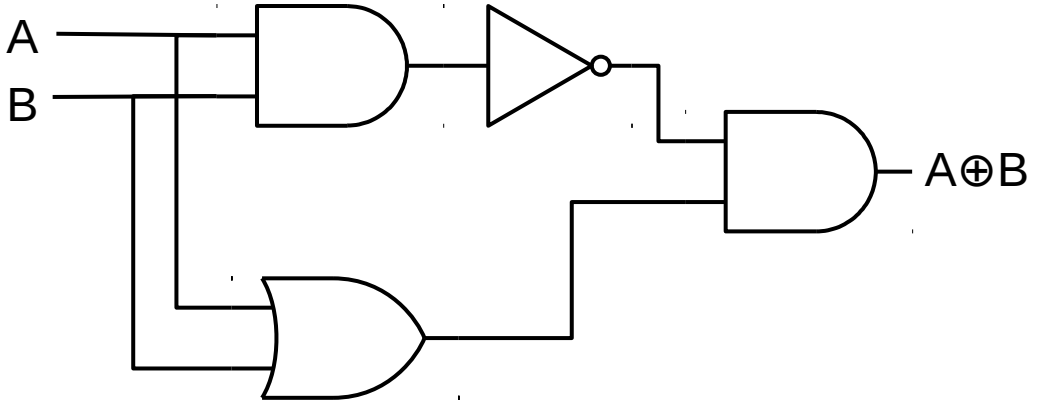
Porte logiche aggiuntive

- Risulta comodo costruire la funzione di OR esclusivo



XOR		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

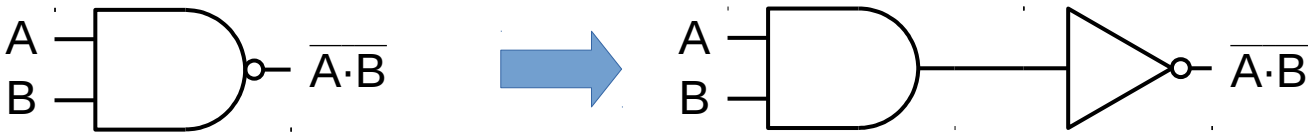




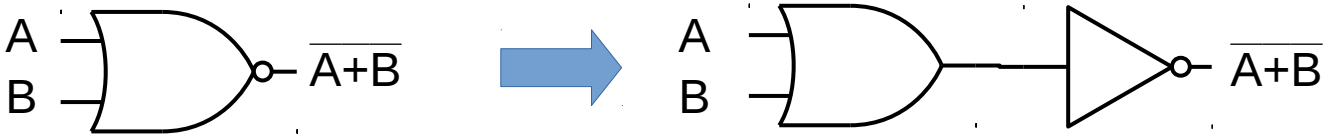
Porte logiche aggiuntive

- Un pallino su un ingresso o uscita indica una negazione:

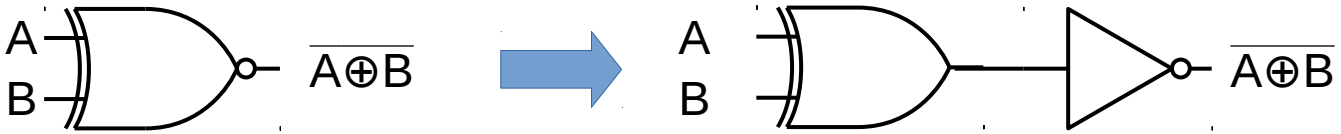
NAND



NOR



XNOR



Proprietà algebra booleana

- Identità:
  - $A + 0 = A$
  - $A \cdot 1 = A$
- Assorbimento:
  - $A + 1 = 1$
  - $A \cdot 0 = 0$
- Inverso:
  - $A + \overline{A} = 1$
  - $A \cdot \overline{A} = 0$
- Idempotenza:
  - $A + A = A$
  - $A \cdot A = A$
- Commutativa:
  - $A + B = B + A$
  - $A \cdot B = B \cdot A$
- Associativa:
  - $A + (B + C) = (A + B) + C$
  - $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributiva:
  - $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
  - $A + (B \cdot C) = (A + B) \cdot (A + C)$
- Assorbimento:
  - $A + \overline{A} \cdot B = A + B$
- Doppia negazione
  - $\overline{\overline{A}} = A$

Forme canoniche e teoremi di De Morgan

- Un’espressione booleana si dice in forma canonica (o normale) quando è rappresentata come:
  - Somma di prodotti (forma disgiuntiva)
  - Prodotto di somme (forma congiuntiva)
- Esistono due teoremi che stabiliscono delle regole di conversione tra somma e prodotto, che si possono usare per passare da una forma canonica all’altra:

$$\overline{A \bullet B} = \overline{A} + \overline{B}$$

$$\overline{A + B} = \overline{A} \bullet \overline{B}$$

- Si possono ovviamente estendere a più variabili, e si possono usare per semplificare una funzione booleana

Funzioni booleane

- Normalmente in fase di progetto si parte da una tabella di verità per poi ricavare una funzione booleana, che poi va semplificata

A	B	C
0	0	0
0	1	1
1	0	0
1	1	1

- Ciò si può fare in maniera metodica, costruendo o una **somma di prodotti** o un **prodotto di somme**

Funzioni booleane

A	B	C
0	0	0
0	1	1
1	0	0
1	1	1

- Supponendo di volere una **somma di prodotti**, costruisco un'espressione per ogni combinazione in cui l'uscita vale 1, e sommo le varie espressioni
- Ogni espressione viene costruita moltiplicando tutte le variabili Z in ingresso, prendendo Z se l'ingresso deve essere 1 e $\overline{Z}$ se l'ingresso deve essere 0
- Dalla tabella sopra risulta:

$$C = \overline{A} \bullet B + A \bullet B$$

Funzioni booleane - semplificazione

- Una volta ottenuta una funzione booleana, si può semplificare appilcando le regole dell'algebra booleana:

  $C = \overline{A} \bullet B + A \bullet B$

  $C = B \bullet (\overline{A} + A)$ → proprietà distributiva

  $C = B$ → idempotenza

- Otteniamo che la variabile C è indipendente dall'ingresso A; questo si può esprimere con una "X" (don't care) nella tabella di verità

A	B	X
X	0	0
X	1	1

Funzioni booleane (2)

A	B	C
0	0	0
0	1	1
1	0	0
1	1	1

- Supponendo di volere un **prodotto di somme**, costruisco un'espressione per ogni combinazione in cui l'uscita vale 0, e moltiplico le varie espressioni
- Ogni espressione viene costruita sommando tutte le variabili Z in ingresso, prendendo Z se l'ingresso deve essere 0 e $\overline{Z}$ se l'ingresso deve essere 1
- Dalla tabella sopra risulta:

$$C = (A + B) \bullet (\overline{A} + B)$$

Funzioni booleane – semplificazione (2)

- Una volta ottenuta una funzione booleana, si può semplificare appilcando le regole dell'algebra booleana:

  $C = (A + B) \bullet (\overline{A} + B)$

  $C = B + (\overline{A} \bullet A)$ → proprietà distributiva

  $C = B$ → idempotenza

- Otteniamo lo stesso risultato ottenuto tramite la somma di prodotti (c.v.d.)
- Per ogni funzione si può scegliere la forma canonica più conveniente, ad esempio:
  - quella in cui compaiono meno termini
  - quella più facile da semplificare
  - …
- Esistono algoritmi per la semplificazione di funzioni booleane, usati per automatizzare la sintesi dei circuiti digitali, ad esempio:
  - Mappe di Karnaugh
  - Metodo di Quine-McClusky
- Per brevità non li trattiamo

Funzioni booleane

- Ovviemente il procedimento si può estendere a più variabili

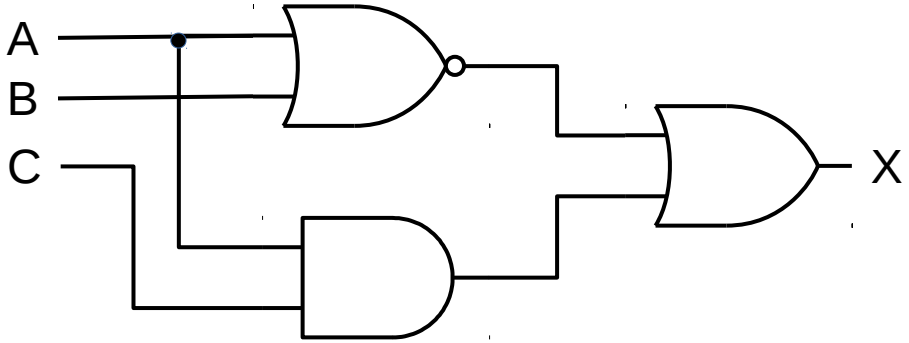
A	B	C	X	Y
0	0	0	1	1
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	0

$$X = (\overline{A} \bullet \overline{B} \bullet \overline{C}) + (\overline{A} \bullet \overline{B} \bullet C) + (A \bullet \overline{B} \bullet C) + (A \bullet B \bullet C) = \overline{A} \bullet \overline{B} + A \bullet C = \overline{A+B} + A \bullet C$$

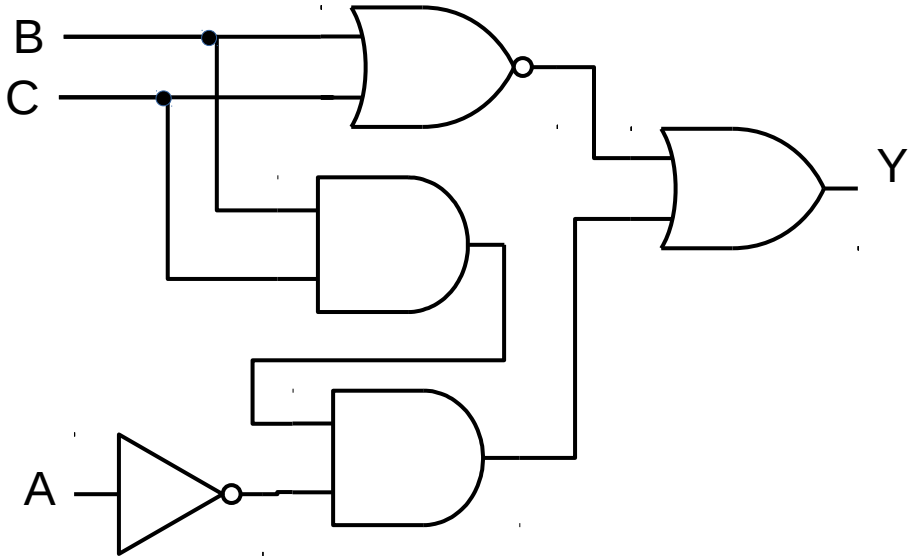
$$Y = (\overline{A} \bullet \overline{B} \bullet \overline{C}) + (\overline{A} \bullet B \bullet C) + (A \bullet \overline{B} \bullet \overline{C}) = \overline{B} \bullet \overline{C} + \overline{A} \bullet B \bullet C = \overline{B+C} + \overline{A} \bullet B \bullet C$$

Funzioni booleane → reti logiche

- Una funzione booleana è facilmente traducibile in reti logiche sostituendo ogni operazione con la corrispondente porta logica e tenendo a mente le regole di precedenza delle operazioni
- Dall'esempio precedente risulta:



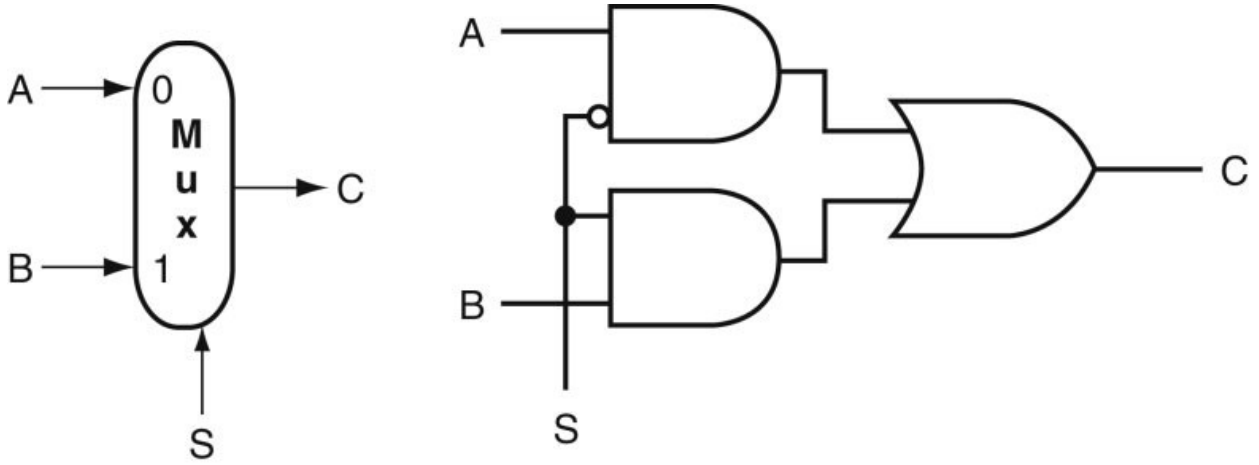
$X = \overline{A+B} + A \bullet C$



$Y = \overline{B+C} + \overline{A} \bullet B \bullet C$

Multiplexer

- Un componente molto utile è il multiplexer, che permette di "selezionare" una particolare variabile in ingresso e riportarla in uscita



- La variabile selezionata dipende dal valore del bit di controllo S:

  S = 0 → C = A

  S = 1 → C = B

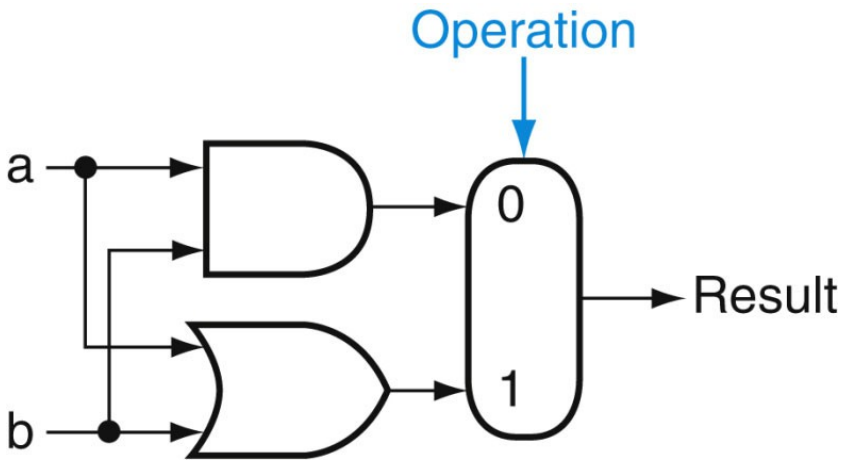
- Con N bit di controllo posso scegliere 2^N variabili in input.

ALU

- La ALU (Arithmetic and Logic Unit) o unità aritmetico-logica è un componente hardware che svolge operazioni aritmetiche e logiche.
- L'ALU è una componente fondamentale della CPU.
- Nelle prossime slides vedremo una ALU semplificata in grado di eseguire le seguenti operazioni su interi signed a 32 bit:
  - somma e sottrazione
  - AND
  - OR
  - altro?
- Obiettivo: costruire una ALU per l'architettura MIPS

ALU – operazioni logiche

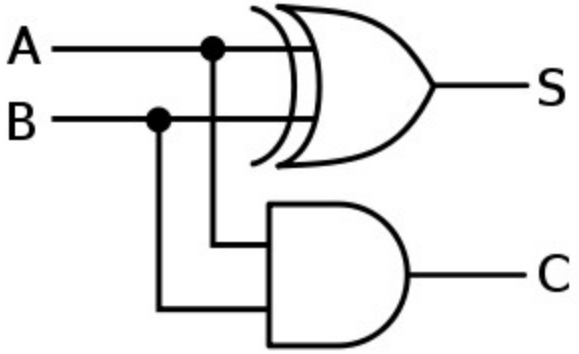
- Le operazioni AND e OR nella ALU sono svolte dalle corrispondenti port logiche.
- Un multiplexer sceglie quale delle due operazioni deve essere svolta.



ALU – operazioni aritmetiche

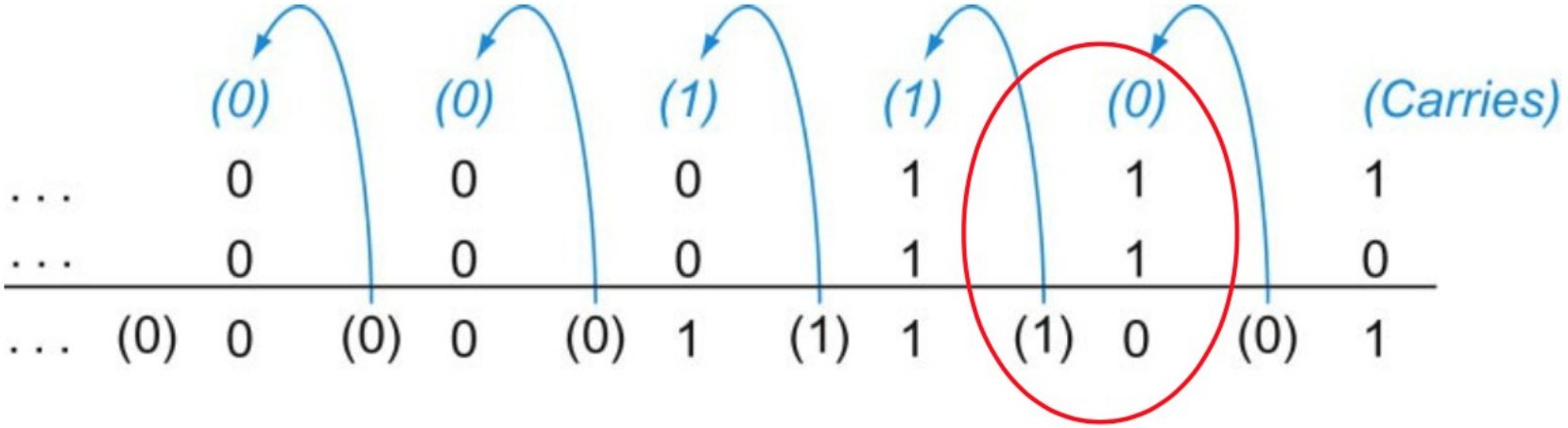
- Come posso implementare operazioni aritmetiche tra due bit?
  - OR → somma logica
  - AND → prodotto logico e aritmetico
  - XOR → somma aritmetica (senza riporto)
- **Half adder**
  - Somma (S) di due bit con riporto (C)

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



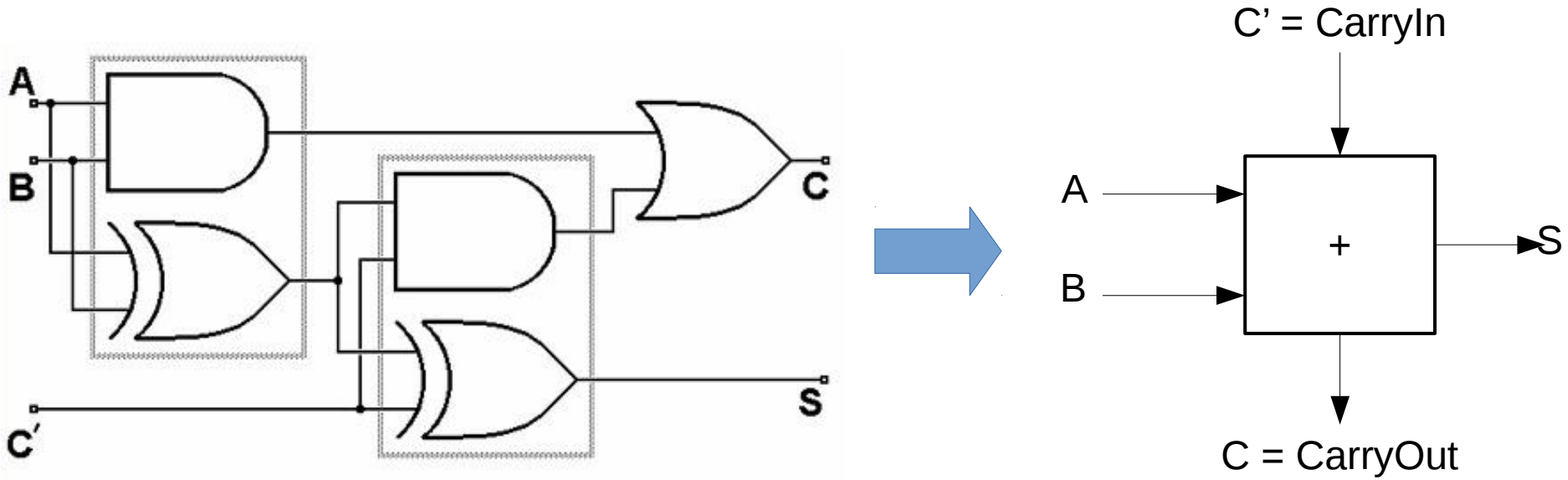
ALU – somme a più bit

- L’addizione di numeri composti da più di 1 bit comporta un problema, ovvero bisogna tener conto del bit di riporto
- questo riporto diventa un terzo input della addizione (Carry In)



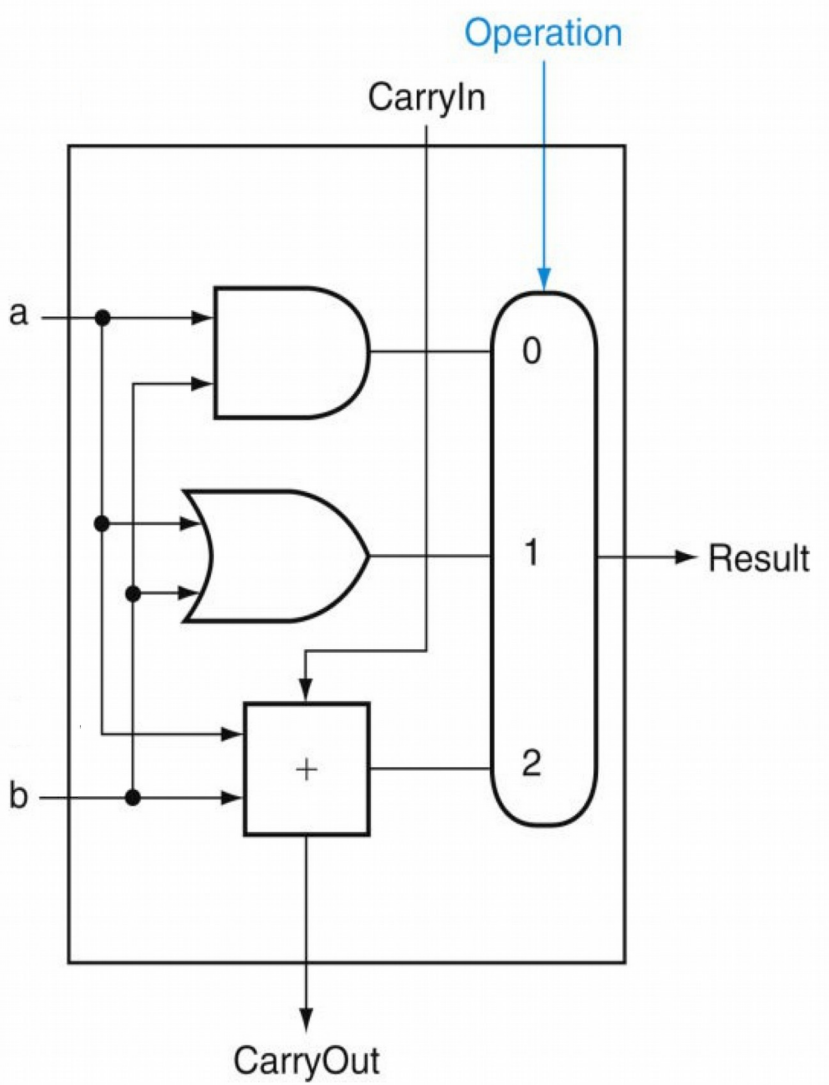
ALU – Full Adder

- **Full Adder**
  - Tiene conto del Carry In, permette addizione di numeri interi a N bit
- E’ composto da due half adder in cascata più un OR che somma i due Carry Out.



ALU a 1 bit

- 3 operazioni disponibili:
  - OR
  - AND
  - Somma con riporto
- *Operation* deve essere almeno a 2 bit

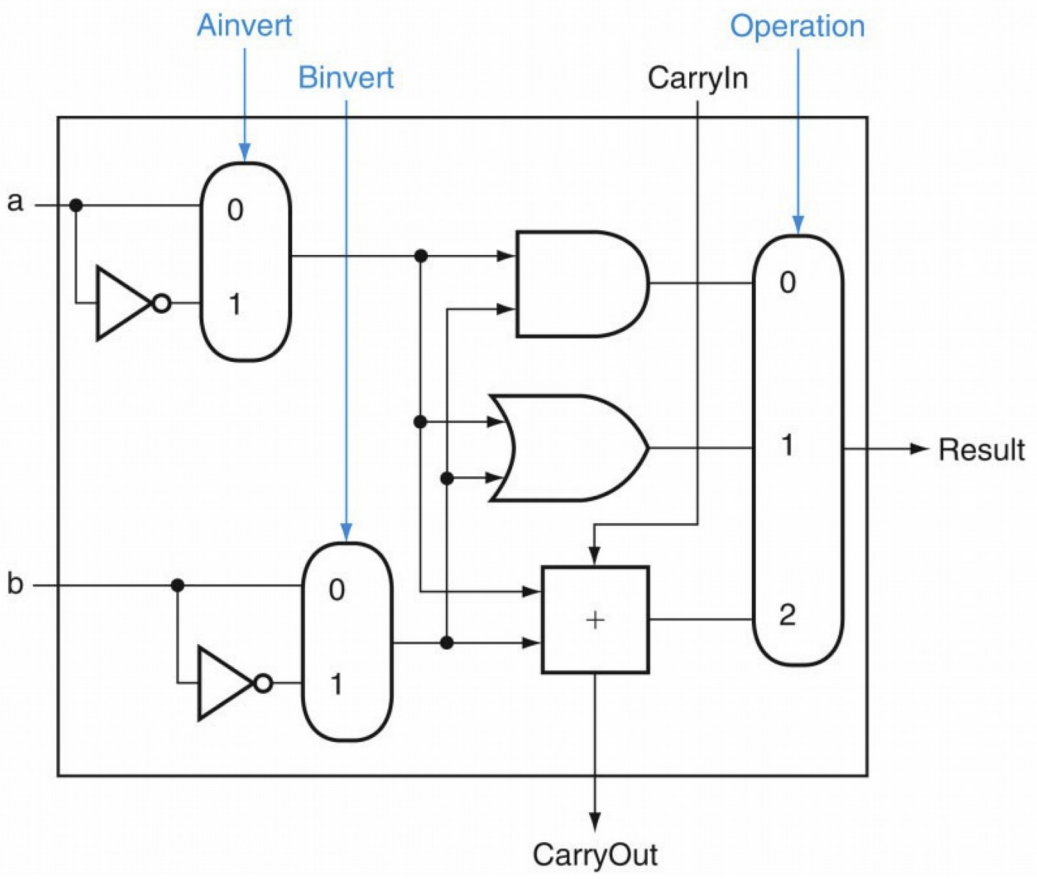


ALU a 1 bit

- Utile aggiungere la possibilità di negare gli ingressi, utile per implementare la sottrazione attraverso il complemento a 2:

  $a - b = a + \overline{b} + 1$

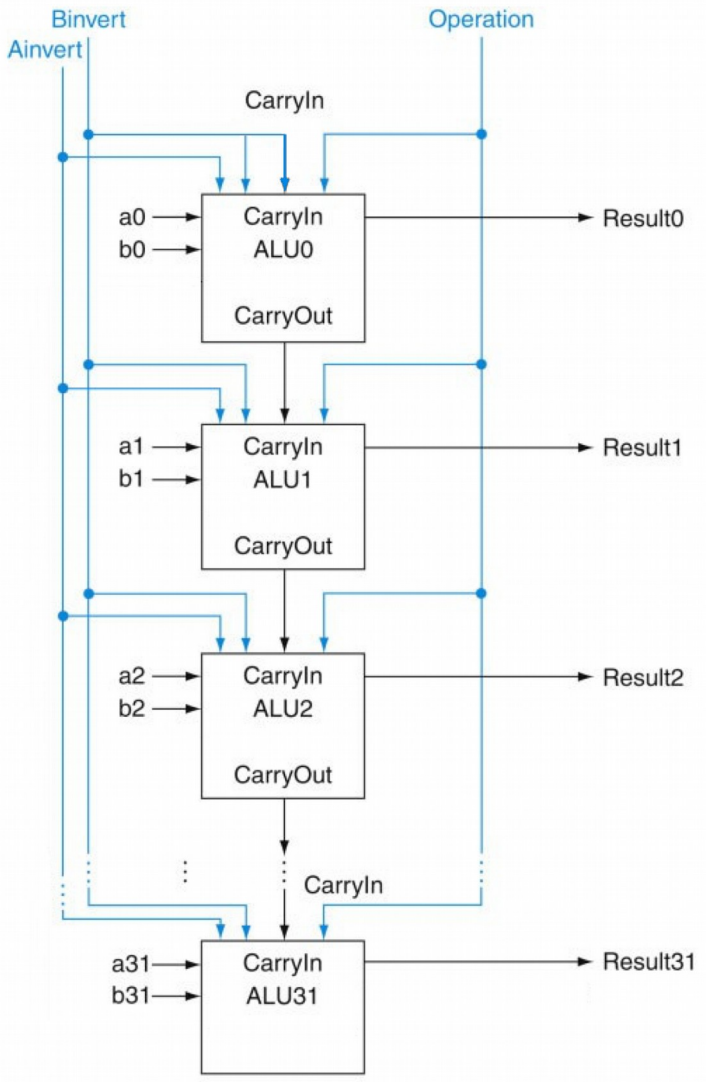
- Ainvert e Binvert sono due multiplexer a 1 bit



ALU a 32 bit

- Mettendo in cascata 32 ALU a 1 bit, ottengo un'ALU in grado di operare su 32 bit
  - AND bitwise
  - OR bitwise
  - Somma con riporto
  - Sottrazione
    - Binvert collegato all'ingresso CarryIn del bit 0 per implementare

      $a - b = a + \overline{b} + 1$

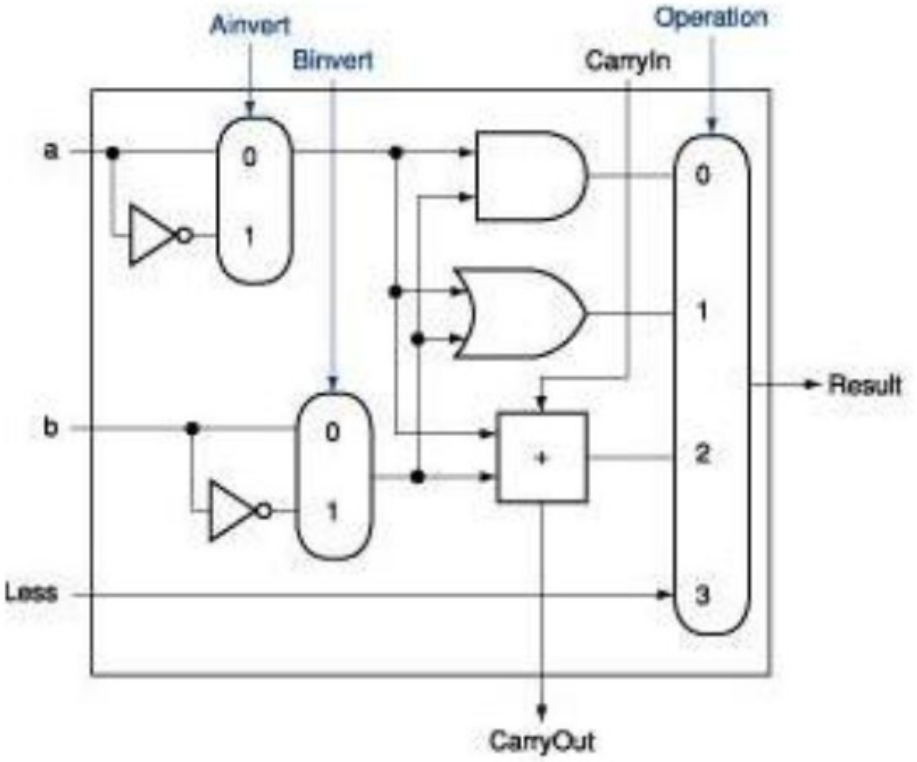


ALU - confronto

- Aggiungiamo operazioni per confrontare due numeri interi
  - a < b → Necessario per istruzione `slt` (set-if-less-than)
  - a == b → Necessario per istruzione `beq` (branch-if-equal)
- Idea: sfutto l'operazione di sottrazione
  - a < b → è sufficiente controllare il bit di segno del risultato
  - a == b → devo avere tutti i bit del rsultato a 0, aggiungo uscita aggiuntiva

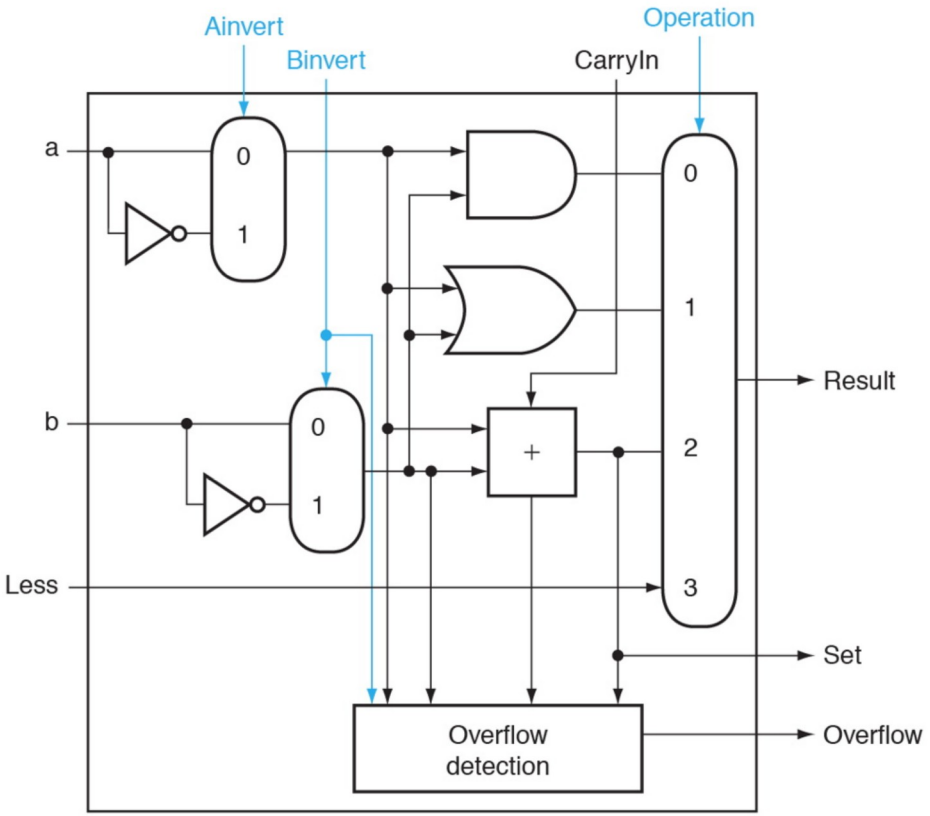
ALU - Confronto `slt` (bit 0-30)

- L'istruzione deve dare come risultato:
  - 0x00000001 se a < b
  - 0x00000000 se a >= b
- Cambia solo il bit 0, quindi posso forzare a 0 tutti gli altri attraverso una linea di ingresso aggiuntiva (Less)
  - andrà forzata a 0 per i bit da 1 a 31
  - Conterrà l'MSB del risultato per il bit 0



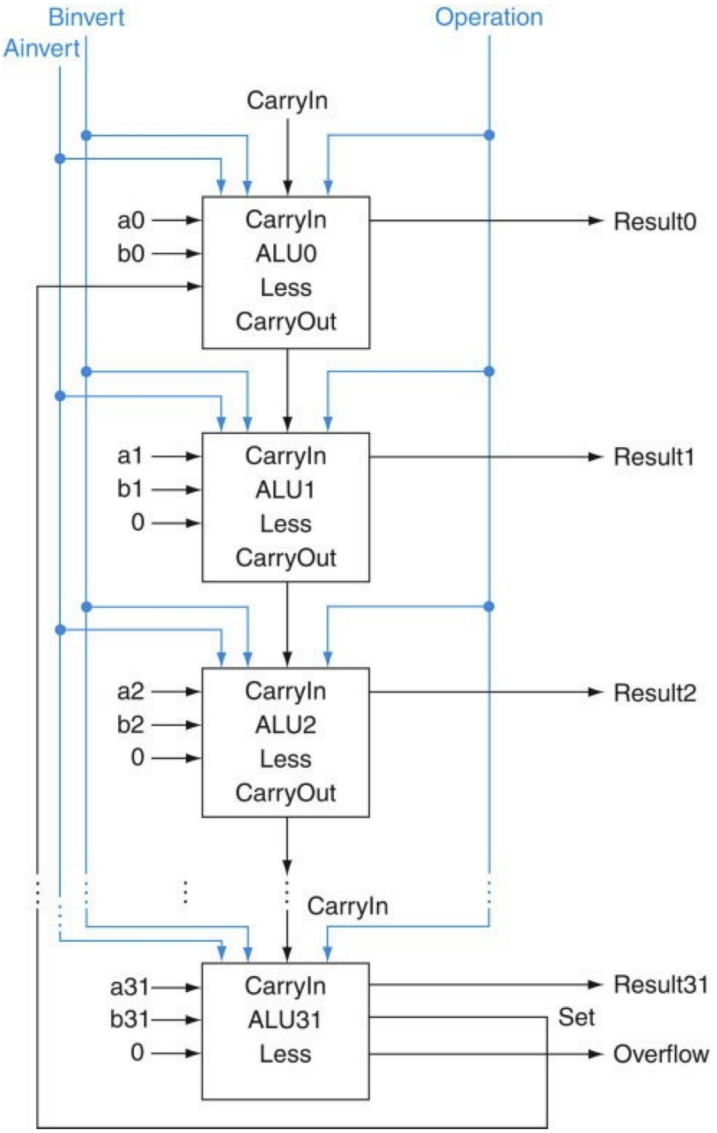
ALU - Confronto `slt` (bit 31)

- L'istruzione deve dare come risultato:
  - 0x00000001 se a < b
  - 0x00000000 se a >= b
- Il risultato del full adder (bit di segno) va sulla linea Set:
  - serve un output aggiuntivo, ossia il risultato del full adder (Set), che diventa l'input Less per la ALU del bit 0
  - E' aggiunta anche una parte per determinare la condizione di Overflow.



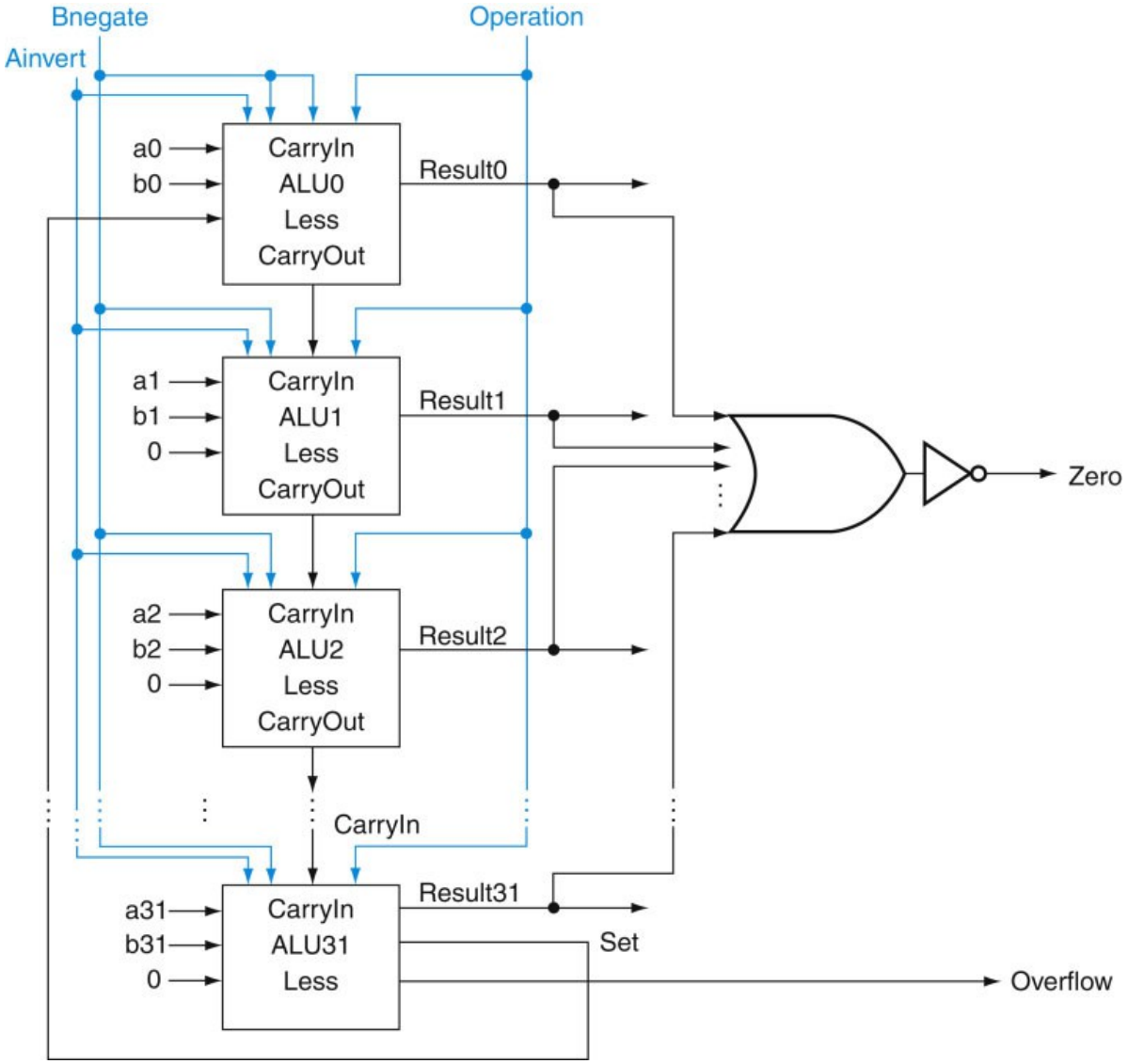
ALU - Confronto `slt`

- L'istruzione deve dare come risultato:
  - 0x00000001 se a < b
  - 0x00000000 se a >= b
- Schema completo con il feedback dal bit 31 al bit 0



ALU – controllo `beq`

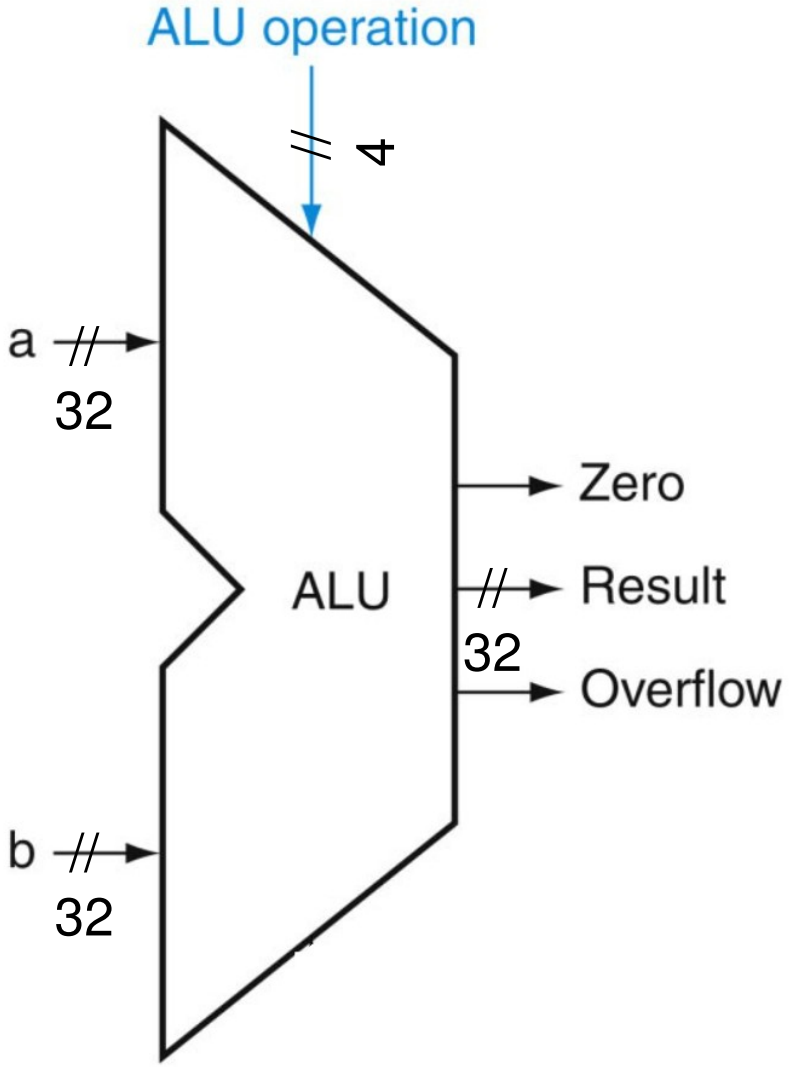
- Aggiungo linea dedicata per segnalare il valore 0x00000000 in uscita
  - Uso porta NOR a 32 ingressi



ALU - sommario

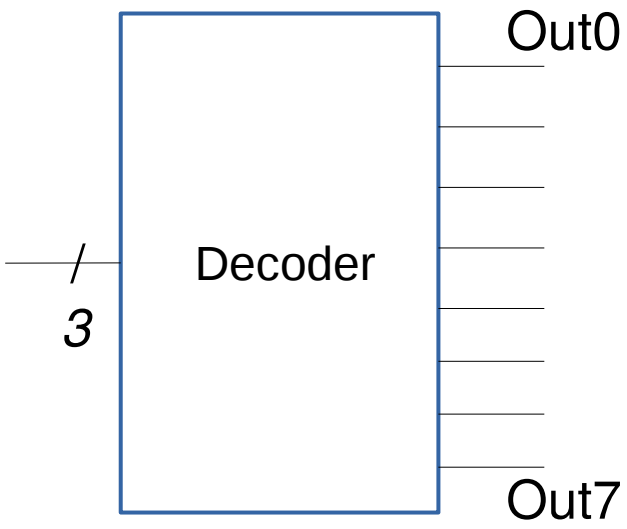
- L'ALU costruita finora permette di svolgere diverse operazioni, a seconda dei bit di controllo

Operazione ALU			Funzione
Aneg	Bneg	Op	
0	0	00	AND bitwise
0	0	01	OR bitwise
0	0	10	Addizione
0	1	10	Sottrazione
0	1	11	Confronto `slt`
1	1	00	NOR bitwise
1	1	01	NAND bitwise



Altri blocchi utili: Decoder $n \rightarrow 2^n$

- In uscita ho 2^n bit di cui uno solo è settato
- In ingresso ho *n* bit che rappresentano un numero, e “scelgono” il bit da settare in uscita



Ingressi			Uscite							
I1	I2	I3	O0	O1	O2	O5	O6	O7	O6	O7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Shifter (traslatore)

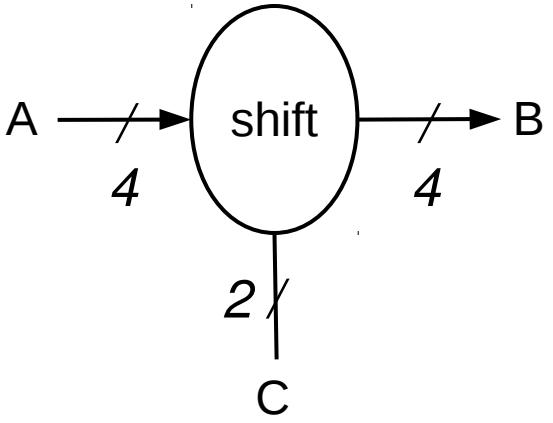
- Esegue l'operazione di shift logico a destra o sinistra
  - Si può implementare con una catena di multiplexer
  - Es. shifter a 4 bit → 2 bit di controllo

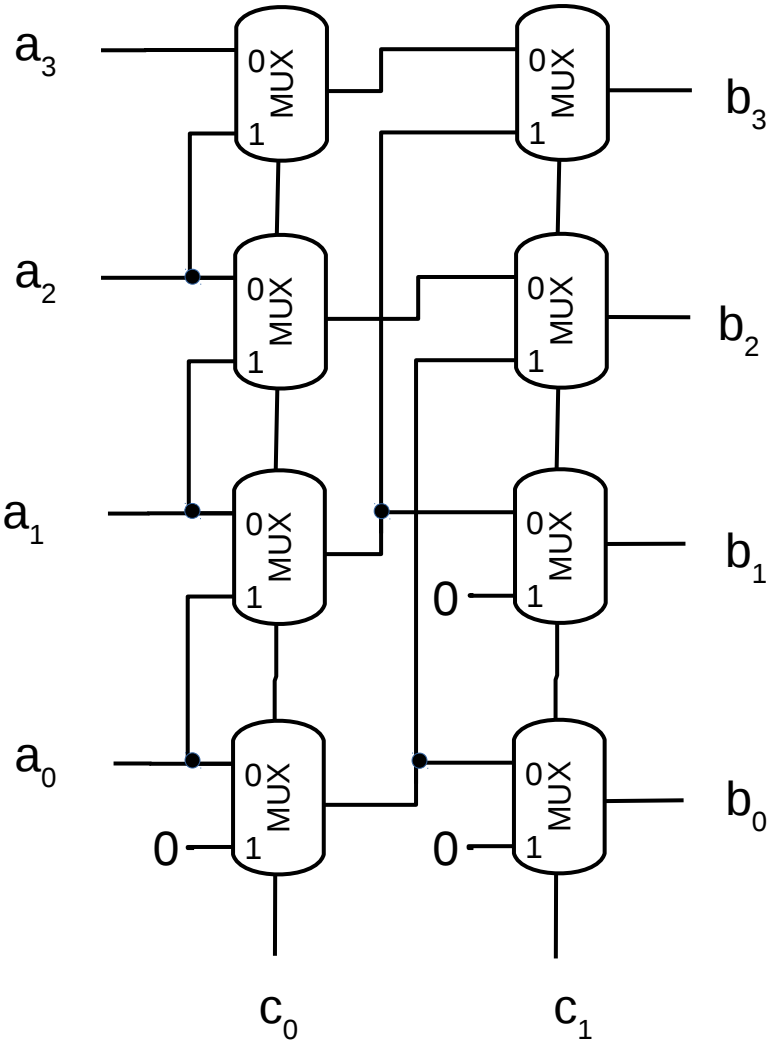
$A = a_3a_2a_1a_0$

$B = b_3b_2b_1b_0$

$C = c_1c_0$

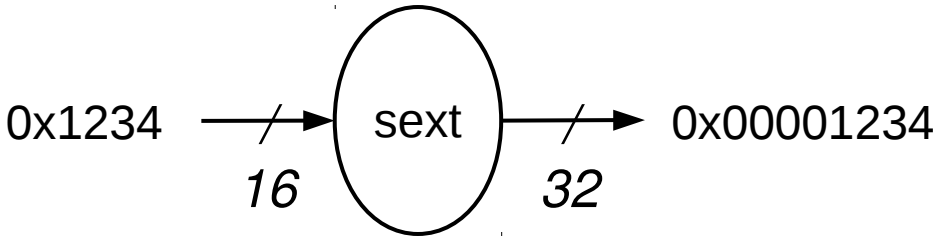
$B = A << C$





Bit extender

- Implementa l’estensione del segno (sign-extend)
- Converte N bit in M, con N < M
  - Aggiunge bit a 0 a sinistra se MSB è 0
  - Aggiunge bit a 1 a sinistra se MSB è 1



Latch S-R (Set-Reset)

- Circuito ***bistabile*** – capace di mantenere 2 stati diversi:

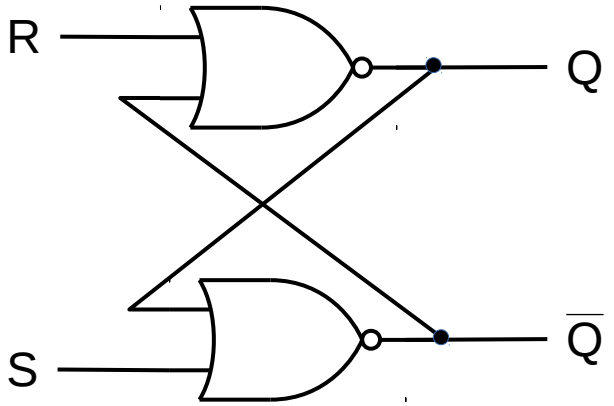
  $Q = 1 \quad (\overline{Q} = 0)$

  $Q = 0 \quad (\overline{Q} = 1)$

- Lo stato viene deciso attivando (portando a 1) un ingresso:

  $S = 1, R = 0 \rightarrow Q = 1$

  $S = 0, R = 1 \rightarrow Q = 0$



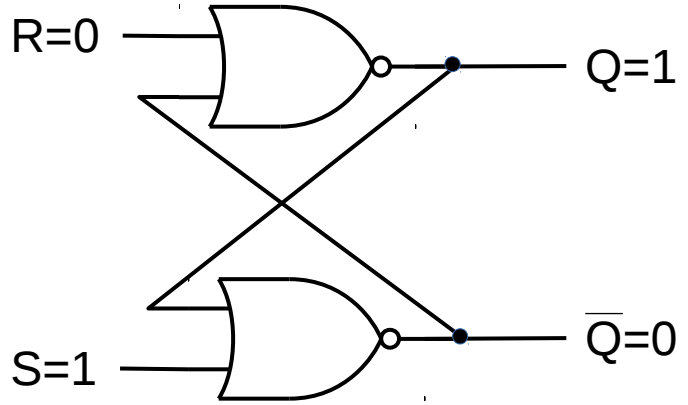
Latch S-R – stato SET

- Poniamo:

  S = 1

  R = 0

- Trattandosi di porte NOR, l'uscita $\overline{Q}$ sarà sicuramente a 0 perché S=1
- L'uscita Q invece dipende da $\overline{Q}$ perché R=0, quindi va a 1



NOR		
0	0	1
0	1	0
1	0	0
1	1	0

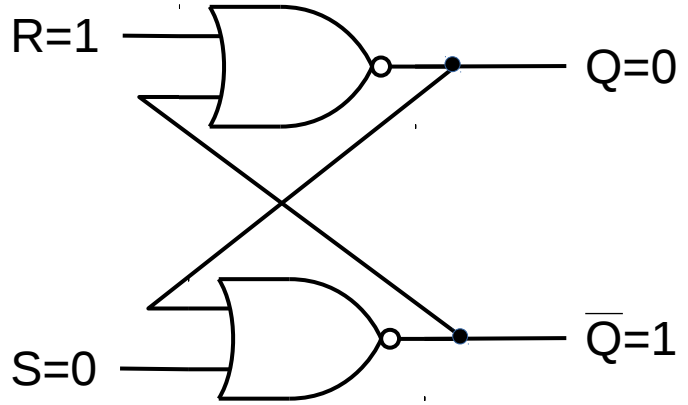
Latch S-R – stato RESET

- Poniamo:

  S = 0

  R = 1

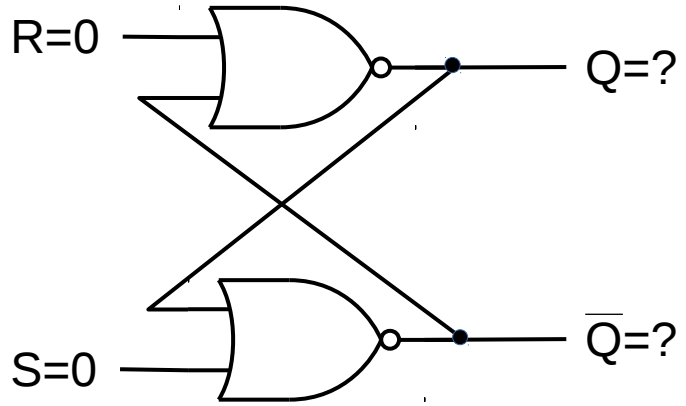
- Trattandosi di porte NOR, l'uscita Q sarà sicuramente a 0 perché R=1
- L'uscita $\overline{Q}$ invece dipende da Q perché S=0, quindi va a 1



NOR		
0	0	1
0	1	0
1	0	0
1	1	0

Latch S-R – stato di riposo

- Poniamo:
  - S = 0
  - R = 0
- Cosa ottengo su Q e $\overline{Q}$?
  - $Q = \overline{R + \overline{Q}} = \overline{0 + \overline{Q}} = \overline{\overline{Q}} = Q$
  - $\overline{Q} = \overline{S + Q} = \overline{0 + Q} = \overline{Q}$
- Esistono due soluzioni possibili, ovvero i due stati
  - Q = 1 e $\overline{Q}$ = 0
  - Q = 0 e $\overline{Q}$ = 1
- Dipende dai valori precedenti di S e R
  - Il latch “ricorda”, attraverso il feedback, l’ultimo ingresso che è stato settato



NOR		
0	0	1
0	1	0
1	0	0
1	1	0

Latch S-R – stato indeterminato

- Poniamo:

  S = 1

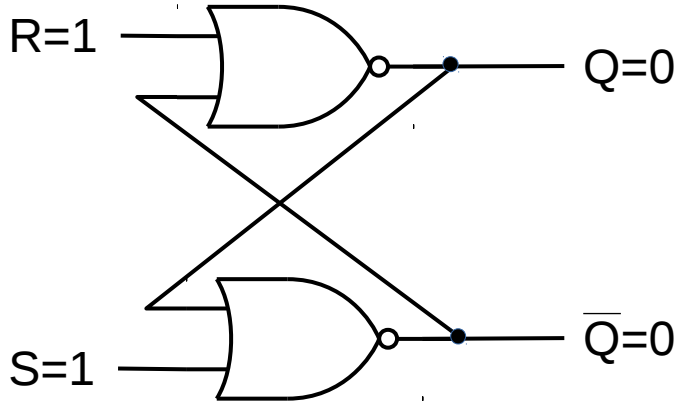
  R = 1

- Cosa ottengo su Q e $\overline{Q}$?

  $Q = \overline{R + \overline{Q}} = \overline{1 + \overline{Q}} = 0$

  $\overline{Q} = \overline{S + Q} = \overline{1 + Q} = 0$

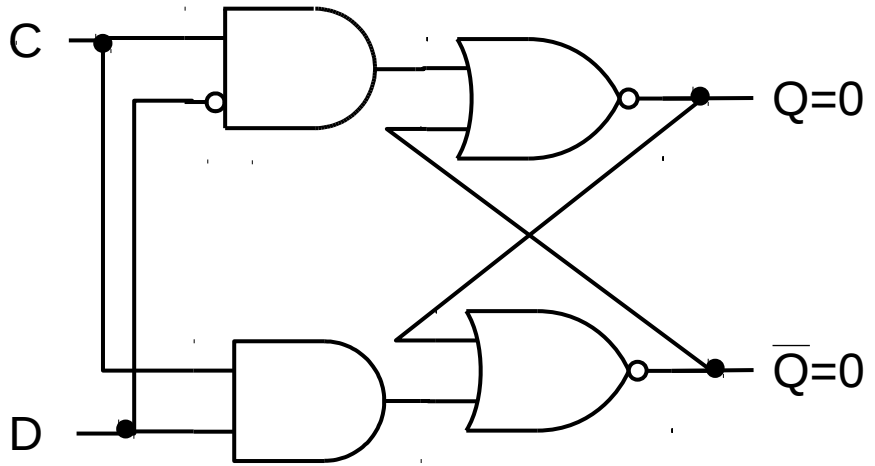
- Anche se stabile, questa configurazione solitamente non viene ammessa perché $Q = \overline{Q}$!
- Cosa succede se S e R passano da 1 a 0 *contemporaneamente*?



NOR		
0	0	1
0	1	0
1	0	0
1	1	0

Latch D

- Evita lo stato indeterminato grazie ad un segnale di abilitazione, chiamato Clock
- Quando C=1, il valore di D viene riportato su Q (e l'inverso su $\overline{Q}$)
  - D=1 corrisponde al Set
  - D=0 corrisponde al reset
- Quando C=0 vene mantenuto lo stato
  - Stato indeterminato non si verifica mai
- Circuito **Level Triggered**
  - La transizione di stato può avvenire in corrispondenza di un *livello logico*

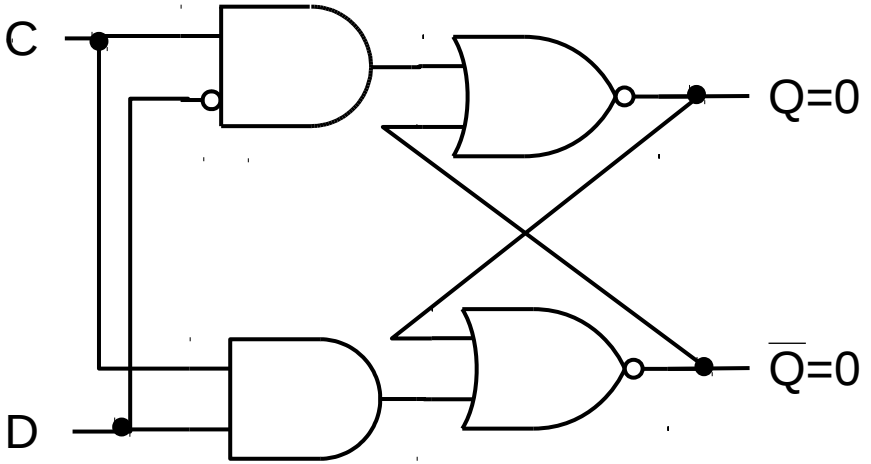


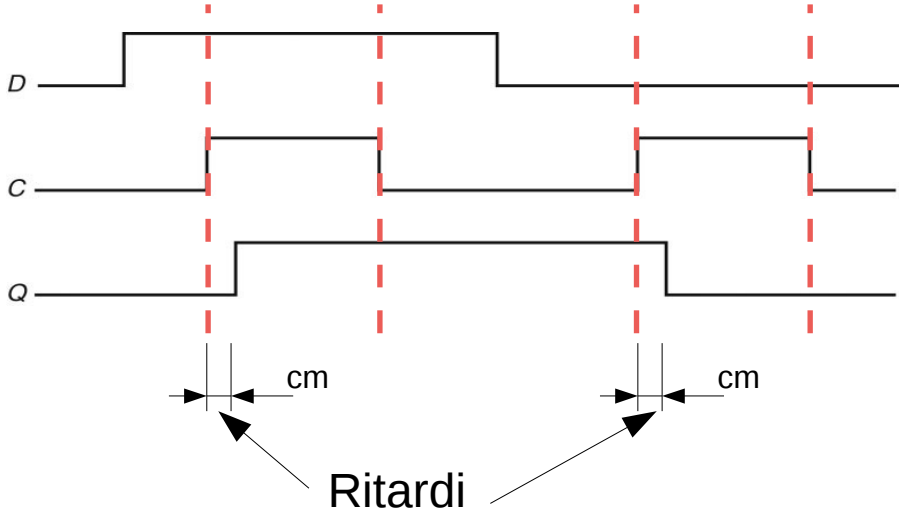
AND		
0	0	0
0	1	0
1	0	0
1	1	1

NOR		
0	0	1
0	1	0
1	0	0
1	1	0

Latch D

- Per tutto il tempo in cui C=1, il latch D è “trasparente”, ovvero riporta in uscita lo stato D (a meno di ritardi di propagazione)
  - Non vengono memorizzati tutti i cambiamenti che avvengono mentre C=1
  - Non è un comportamento che vogliamo!



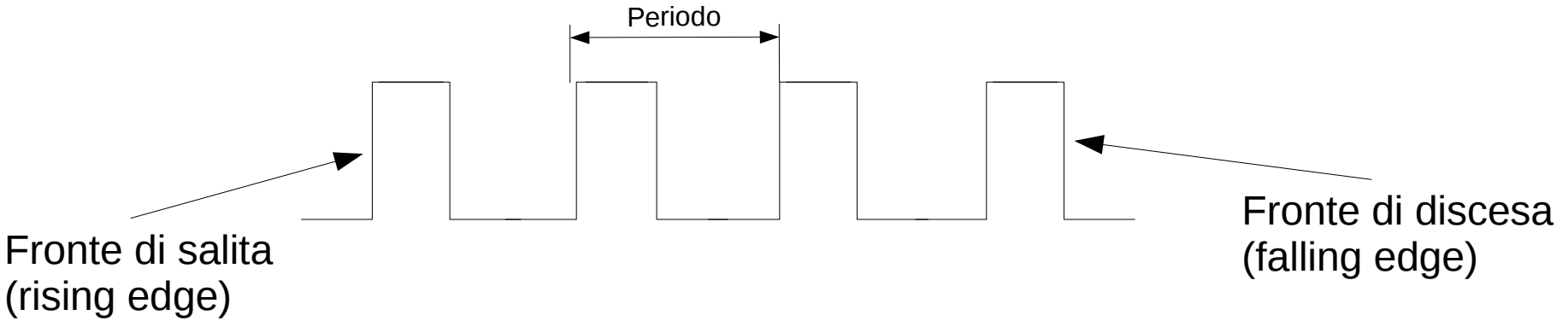


AND		
0	0	0
0	1	0
1	0	0
1	1	1

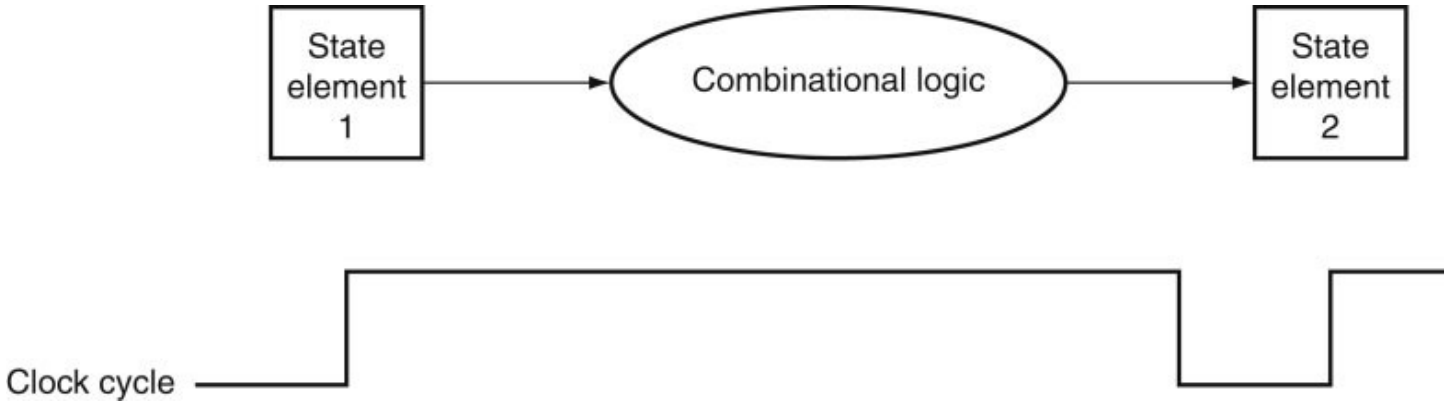
NOR		
0	0	1
0	1	0
1	0	0
1	1	0

Clock

- I segnali di Set e Reset solitamente arrivano da altri circuiti digitali, e per motivi fisici possono non essere perfettamente sincronizzati
  - Ogni porta logica introduce un ritardo di propagazione
  - I segnali S e R diventano stabili solo dopo un certo tempo
- Per avere un comportamento predicibile, è necessario ***sincronizzare*** le transizioni di stato
- Questo sincronismo è dato da un segnale di **clock:**



Clock



- Il primo elemento di stato fornisce gli input al circuito logico combinatorio.
- La durata di un ciclo di clock deve essere sufficientemente lungo affinché i segnali in input riescano a propagarsi all'interno del circuito combinatorio e generare un segnale di uscita stabile.
- il clock è unico per ogni sotto-sistema (CPU, memoria, …) il periodo deve essere sufficiente affinché ogni circuito logico abbia il tempo di stabilizzare il proprio segnale di output.

Clock

- Il periodo (o ciclo) di clock dev'essere abbastanza lungo da assicurare la stabilità degli output del circuito
  - Spesso la tecnologia di realizzazione limita la frequenza massima (ma non è l'unico parametro)
- Il clock abilita la scrittura nei latch, evitando lo stato indeterminato
- Il clock determina il ritmo dei calcoli e delle relative operazioni di memorizzazione
- Il circuito diventa sincrono rispetto al segnale di clock

Clock

- In un circuito digitale ci possono essere diversi clock a diversa frequenza
  - In genere derivano tutti da un clock unico, generato da un circuito ***oscillatore***
  - Clock secondari sono derivati attraverso dei divisori di frequenza (prescaler) o moltiplicatori di frequenza (PLL, Phased-Locked Loop)
  - Ci possono comunque essere più oscillatori indipendenti (e.g. per basso consumo)
- Es. clock CPU a 1.2 GHz → periodo di ~ 0.75 ns

  clock memoria a 666 MHz → periodo di ~ 1.5 ns

Mantenimento dello stato

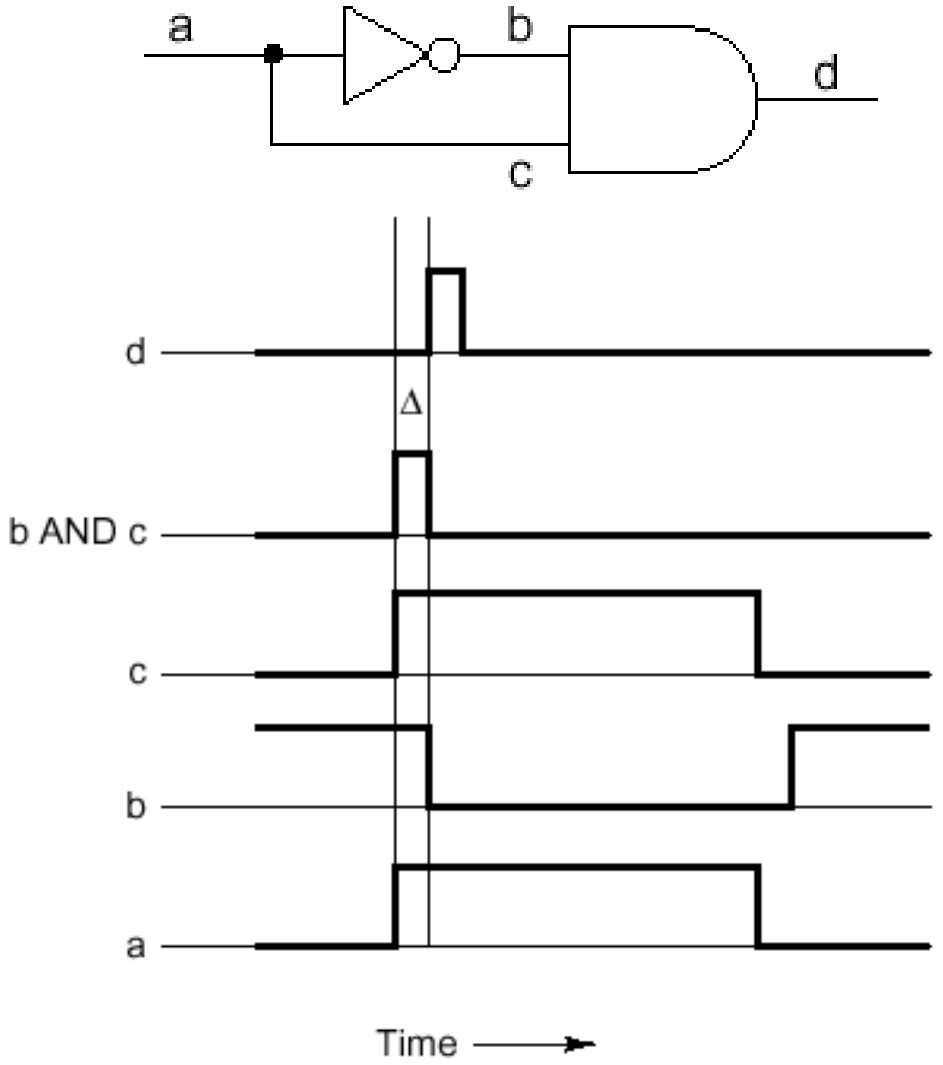
- Vorremmo che, ad ogni ciclo di clock
  - Il circuito combinatorio esegue un'operazione avente come ingresso lo stato memorizzato in un circuito sequenziale
  - Il risultato deve diventare lo stato di un altro circuito sequenziale (o lo stato futuro dello stesso)
  - In questo modo riesco a costruire delle macchine a stati il cui stato si evolve in sincronia con il clock
- Una CPU non è altro che un'enorme macchina a stati finiti composta da
  - Memorie, registri, latch → mantengono lo stato
  - ALU, multiplexer e altre reti logiche → eseguono operazioni logiche e aritmetiche

Metodologia di timing

- La memorizzazione può avvenire in vari istanti rispetto al clock:
  - **level-triggered**: avviene sul **livello** alto (o basso) del clock
    → il D latch ne è un esempio
  - **edge-triggered**: avviene sul **fronte** di salita (o discesa) del clock
    → è quello che ci serve!
- I latch visti finora sono tutti level-triggered
- Per avere elementi edge-triggered esistono diverse soluzioni, ad esempio l'utilizzo di un generatore di impulsi
- Gli elementi edge-triggered sono chiamati **flip-flop**

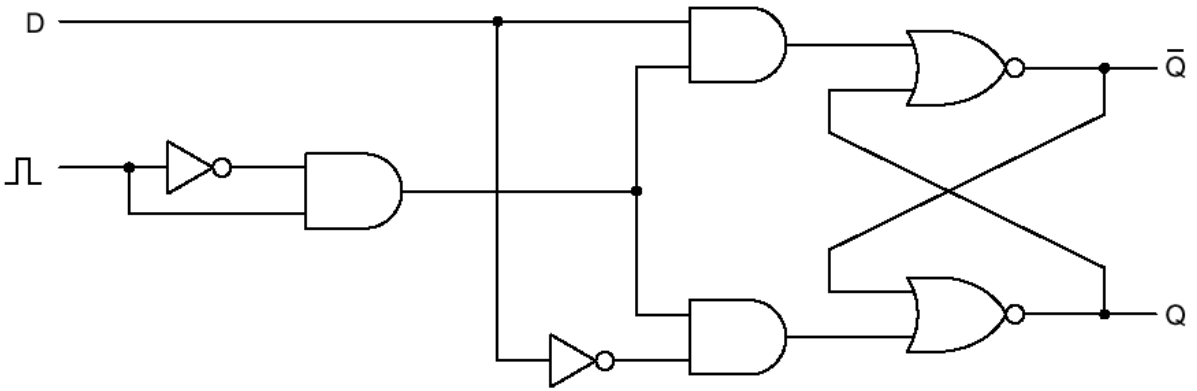
Generatore di impulsi

- Converte un segnale di clock in una sequenza di impulsi, sfruttando il ritardo di propagazione Δ delle porte logiche
- Può essere usato per mantenere lo stato “trasparente” di un latch D per un tempo brevissimo



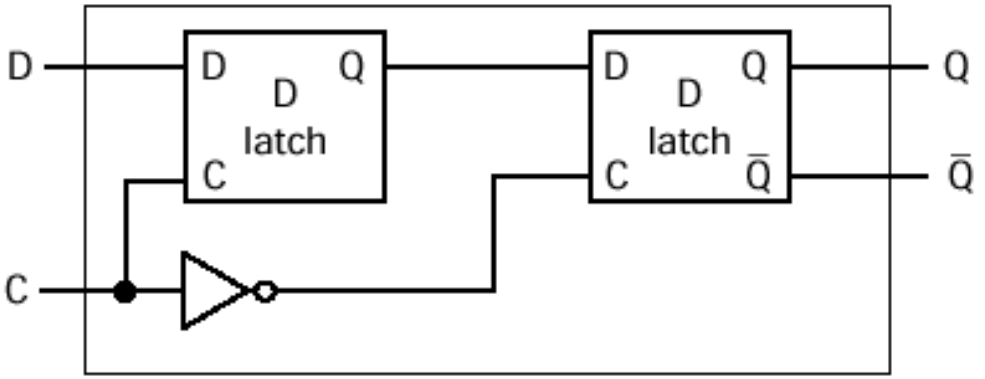
Flip-flop tipo D

- Funzionamento simile al latch D, però funzionamento edge-triggered
- Implementazione con generatore di impulsi



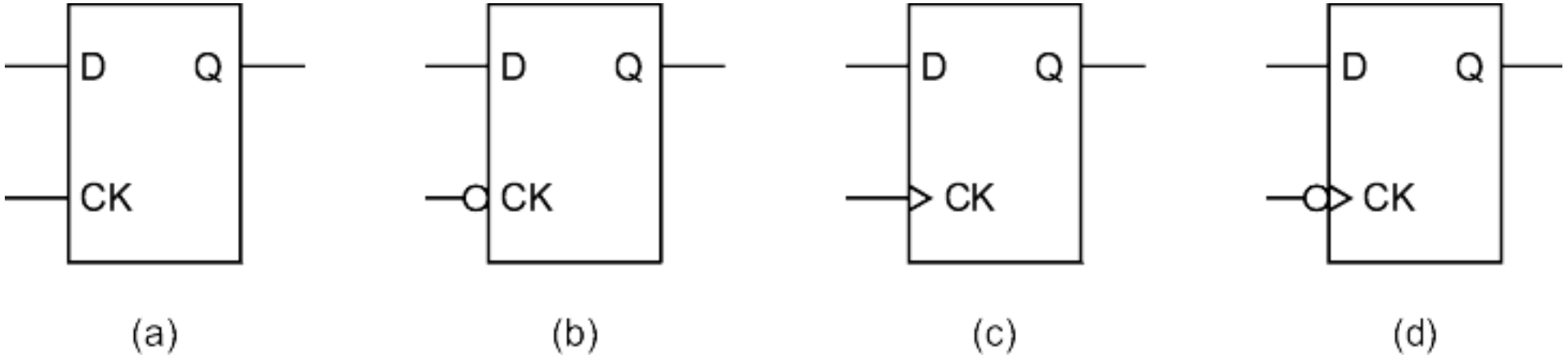
Trigger sul fronte di salita

- Implementazione con doppio latch



Trigger sul fronte di discesa

Simboli latch e flip-flop



Latch attivo alto

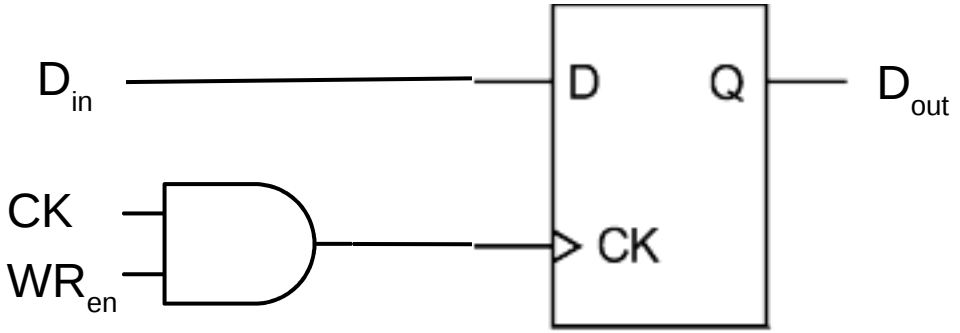
Latch attivo basso

Flip-flop attivo su fronte di salita

Flip-flop attivo su fronte di discesa

Solitamente si omette l’uscita negata se non si usa

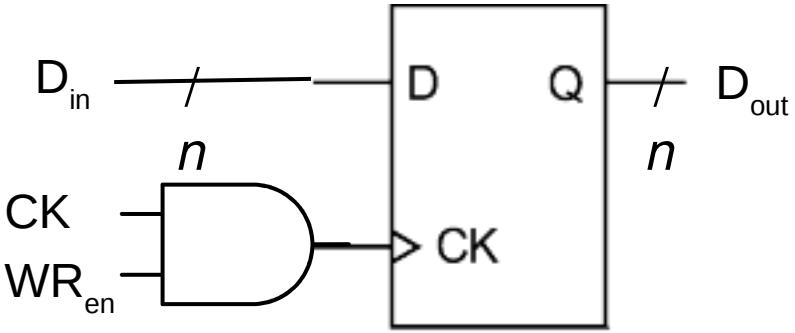
Memoria a 1 bit

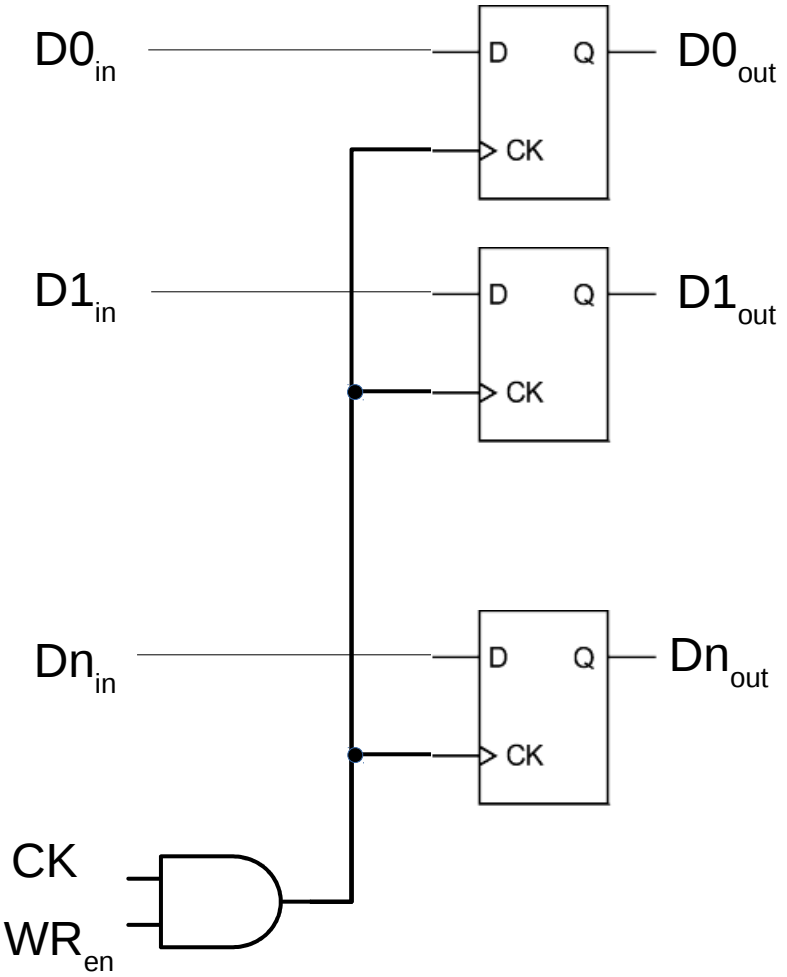


- D_{in} → usato per scrivere il dato
- D_{out} → usato per leggere il dato
- CK → clock usato per sincronizzare la scrittura del dato
- WR_{en} → abilita la scrittura
- Il dato viene memorizzato sull'uscita D

Memoria a *n* bit

- In maniera analoga a quanto fatto per l'ALU, possiamo costruire una cella di memoria a *n* bit unendo altrettante celle da 1 bit



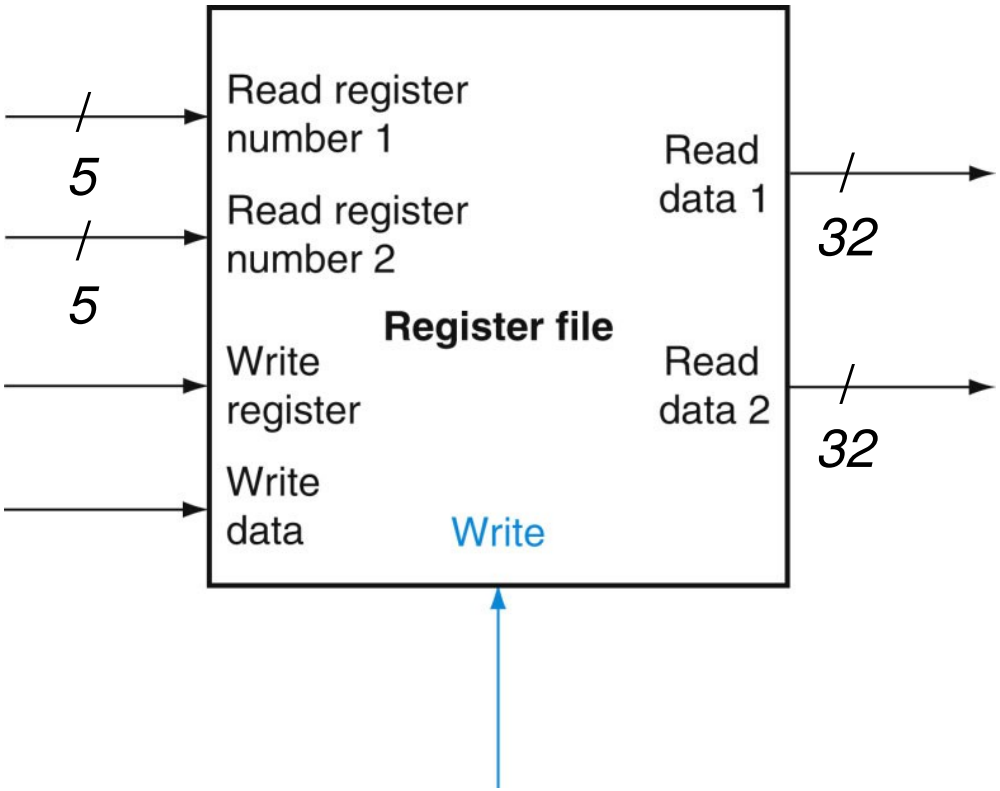


Register File

- La parte operativa della CPU (datapath) contiene alcuni registri che memorizzano gli operandi delle istruzioni aritmetico/logiche:
  - ogni registro è costituito da n flip-flop, dove n è il numero bit che costituiscono una parola
- Più registri sono organizzati in una componente nota come Register File:
  - il Register File del MIPS contiene 32 registri (32 • 32 bit = 1024 flip-flop)
- Il Register File deve permettere:
  - lettura di 2 registri
  - scrittura di 1 registro

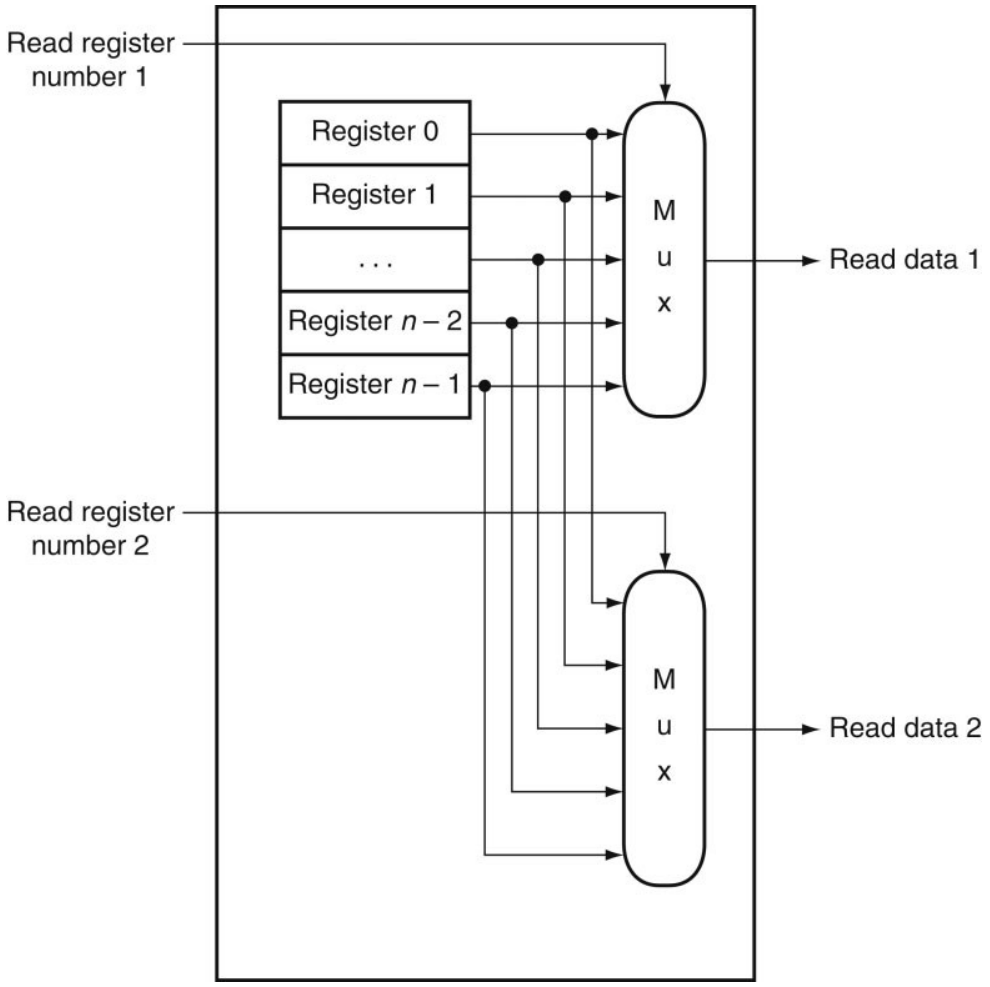
MIPS Register File - lettura

- Read reg. #1 (5 bit):
  - numero del primo registro da leggere
- Read reg. #2 (5 bit):
  - numero del secondo registro da leggere
- Read data #1 (32 bit):
  - valore del primo registro, selezionato sulla base di Read reg. #1
- Read data #2 (32 bit):
  - valore del secondo registro, selezionato sulla base di Read reg. #2



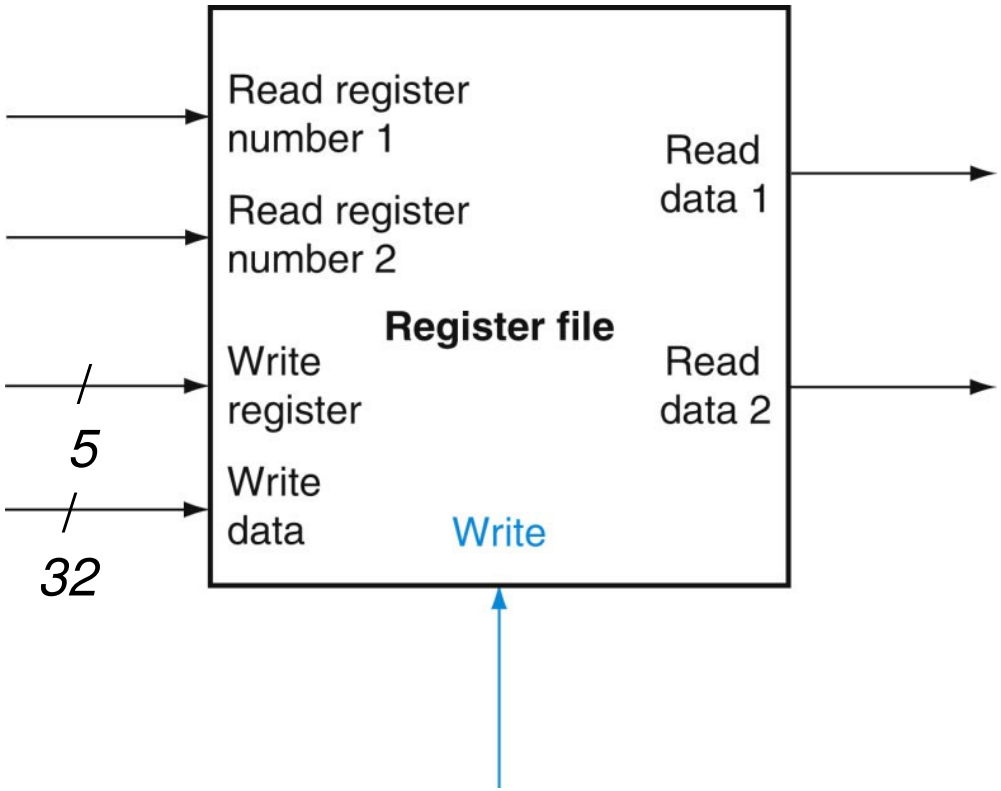
MIPS Register File - lettura

- MUX a 32 bit per selezionare il registro
- Le operazioni richiedono solitamente 2 operandi.
- Ogni input indica il numero del registro da leggere.
- Ogni output contiene il valore del registro letto.
- Il Register File fornisce sempre in output dei valori, semplicemente vengono ignorati se non si è in una operazione di lettura.



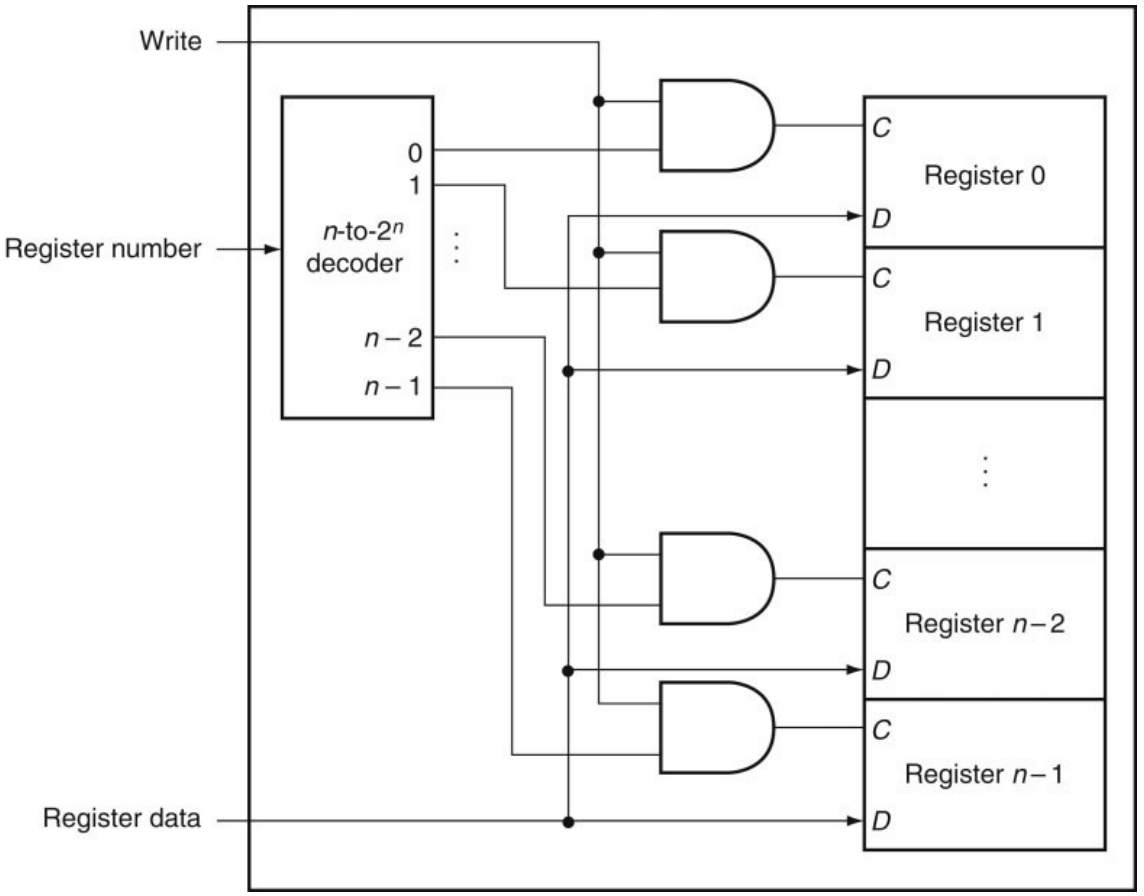
MIPS Register File - scrittura

- Write register (5 bit):
  - numero del registro da scrivere
- Write data (32 bit):
  - valore da scrivere nel registro

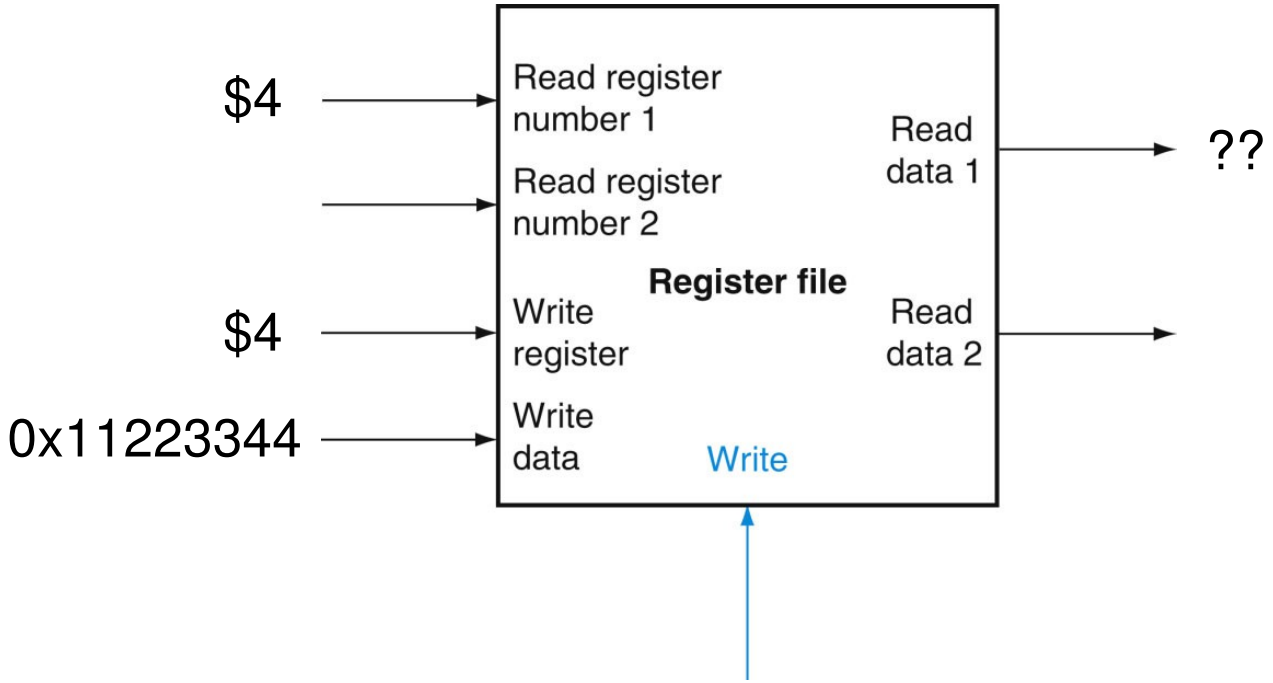


MIPS Register File - scrittura

- Viene scritto un solo registro alla volta.
- Il decoder trasforma il segnale a 5 bit nel numero di registro scelto (da `$0` a `$31`).
- Il segnale Write è in AND con l'output del decoder (e il clock, implicito) per abilitare la scrittura per il registro selezionato.



MIPS Register File – Lettura/Scrittura



- Cosa succede se abilito un registro sia in lettura che scrittura?
  - Ad esempio per `add $4,$0,$4` pongo Read Register Number #1 = 4 e Write Register = 4
- Leggo il valore precedente del registro, non quello che sta per essere scritto
  - Operazioni sincronizzate da clock

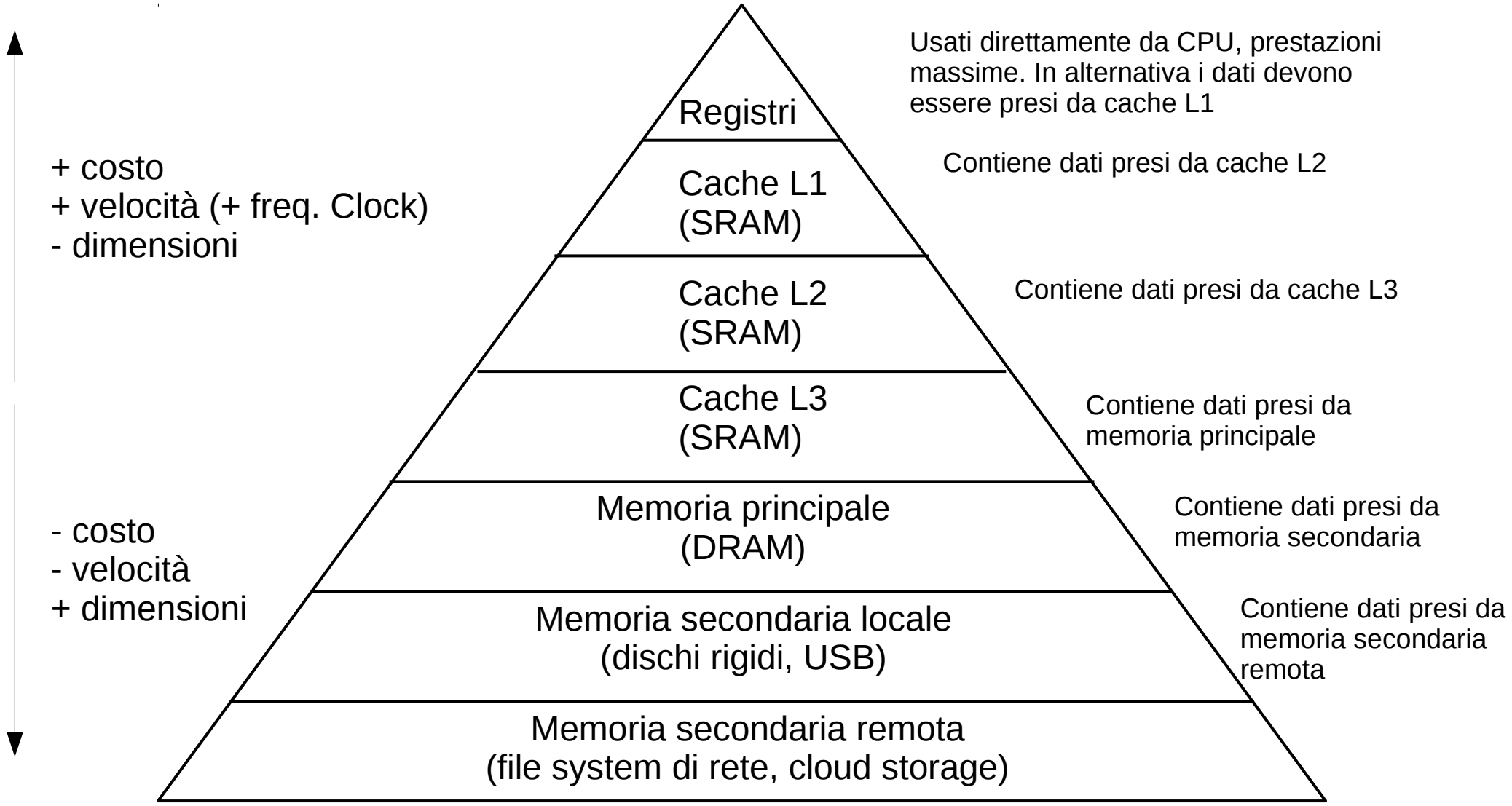
Altri tipi di memoria

- Non è economicamente sostenibile avere memorie implementate come il Register File ma di dimensioni molto superiori:
  - multiplexer e decoder enormi
  - troppi transistor per bit
- Esistono tecnologie alternative che permettono di avere densità maggiori di bit:
  - costi inferiori
  - prestazioni inferiori
  - Es. SRAM, DRAM, SDRAM, DDR

Memoria principale

- Memoria principale
  - meno veloce della memoria dei registri, ma molto più capiente
  - è composta da più livelli (gerarchie di memoria)
- RAM - Random Access Memory - i tempi di accesso sono indipendenti dall'indirizzo della cella di memoria acceduta
  - Solitamente è un tipo di memoria volatile (perde i dati se non alimentata)
- Direttamente indirizzabile dalla CPU
  - In genere un indirizzo nelle istruzioni MIPS si riferisce alla memoria principale
  - La memoria effettiva può essere mappata solo in uno specifico intervallo
    - ad es. se ho meno di 4 GB di memoria, il massimo indirizzabile a 32 bit, avrò delle aree di memoria "vuote"
- Altri tipi di memoria, come i dischi rigidi, non sono indirizzabili direttamente ma si accedono tramite delle apposite periferiche (controller SATA, schede di rete, etc)

Gerarchia di memoria



Memoria SRAM

- Memorie ad accesso casuale statiche
  - Possono mantenere i dati per lungo tempo mantenendo bassi i consumi
- A differenza del register file, non permettono lettura e scrittura contemporanee
- Memorizzazione stato effettuata con dei **latch**, da 4 a 8 transistor per bit
- Tempi di accesso ~ 1 nanosecondi → molto veloci
- Memorie solitamente **asincrone** (no clock)
  - Esistono varianti sincrone chiamate SSRAM (Synchronous SRAM)
- Dimensioni raramente oltre qualche MB per motivi economici
- Usate per:
  - Cache L1/L2/L3 in processori per PC/server/Workstation (es. x86)
  - Memoria principale in sistemi embedded a basso consumo (System on Chip)

Memorie DRAM

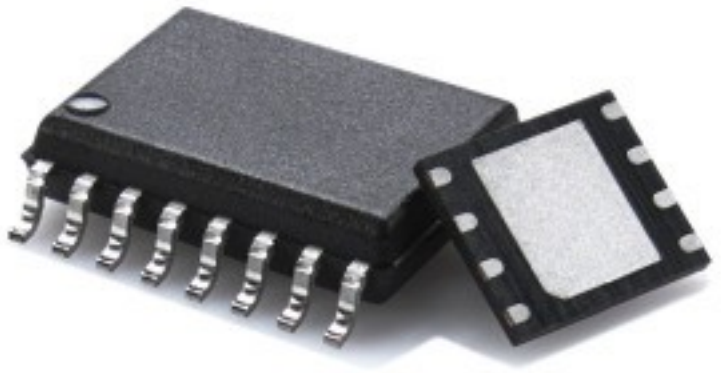
- Memorie ad accesso casuale dinamiche
  - Necessitano di un continuo segnale di rinfresco per mantenere memorizzati i dati (~ ogni ms)
- No lettura/scrittura contemporanea
- Memorizzazione effettuata tramite un condensatore
  - Piccola “batteria” che si può caricare e scaricare molto velocemente, per memorizzare rispettivamente 1 o 0
  - É sufficiente 1 transistor per bit
  - Richiesto controller complesso
- Tempi di accesso ~ 50-100 nanosecondi → più lente di SRAM

Memorie DRAM

- Dimensioni anche di vari GB grazie al basso costo per bit
- Usate per:
  - Memoria principale su PC/server/Workstation, comunemente chiamata “RAM”
- Spesso l’accesso alla memoria è asincrono
  - Interfacciamento complesso con CPU, può introdurre ritardi
- SDRAM → Sychronous DRAM
  - Interfacciamento CPU più semplice grazie all’uso di un clock
- DDR – Double Data Rate SDRAM
  - Utilizzo più efficiente del clock, può trasferire dati sia sui fronti di salita che di discesa

Memorie Flash

- Uso consolidato in sistemi embedded come memoria di “grandi” dimensioni e a lunga ritenzione
  - ~ 10 – 100 MB, recentemente anche diversi GB
- In crescente uso anche su PC come memoria di massa (SSD, Solid State Drive)
- Non si cancella in mancanza di tensione di alimentazione



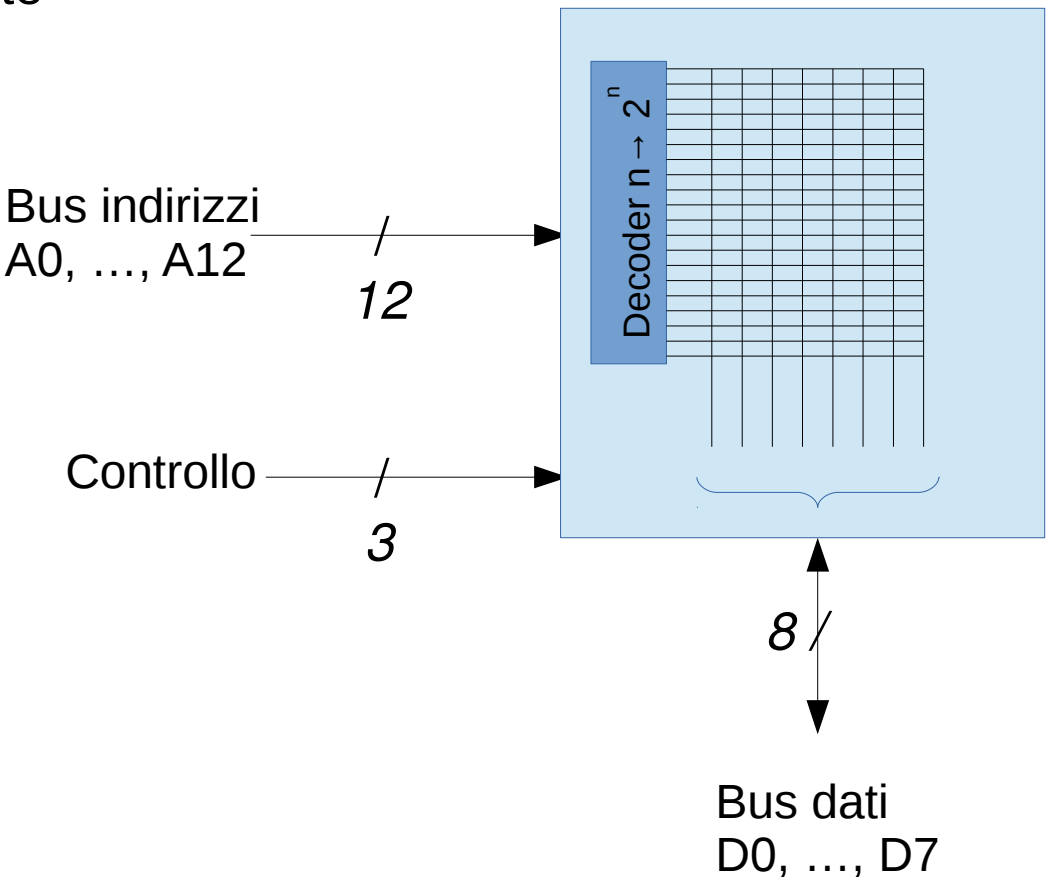
- Organizzata in blocchi, che sono la minima unità scrivibile
- Memoria ad accesso quasi casuale in lettura, tempi di accesso ~ 10 us
- Operazioni di scrittura solitamente a blocchi, richiede esplicitamente cancellazione
- Cicli di scrittura limitati, su SSD si implementano algoritmi specifici per allocare i blocchi al fine di limitarne l’usura
  - A volte implementati a livello di file-system

Memorie su disco fisso

- Capacità molto superiore a SRAM/DRAM/Flash (Terabyte, 10^{12} byte)
- Memorizzazione attraverso un segnale magnetico su disco rotante (es. 7200 giri/minuto)
- Non è una memoria ad accesso casuale
  - Motivi meccanici, la lettura deve essere sincronizzata con la rotazione del disco
  - A 7200 rpm ho 120 giri/secondo, posso dover aspettare fino a 1/120 =~ 8 ms per avere un certo dato
  - In compenso, ho letture sequenziali abbastanza veloci
- Memoria molto lenta rispetto a CPU (~ ms)
  - Normalmente si usa una cache per velocizzare l'accesso, pone però problemi di integrità dei dati in caso di interruzione alimentazione

Struttura memorie SRAM e DRAM

- Una memoria RAM è sostanzialmente una matrice di elementi, o **blocco**
  - BUS → insieme di linee di comunicazione
    - Bus dati
    - Bus indirizzi
  - Segnali di controllo
    - CS – Chip Select
    - OE – Output enable
    - WE – Write enable
- Organizzate come matrici **R** x **C** di celle a 1 bit
- Posso selezionare una riga (**R**) alla volta attraverso un bus indirizzi
  - L'indirizzo indica il numero di riga da accedere

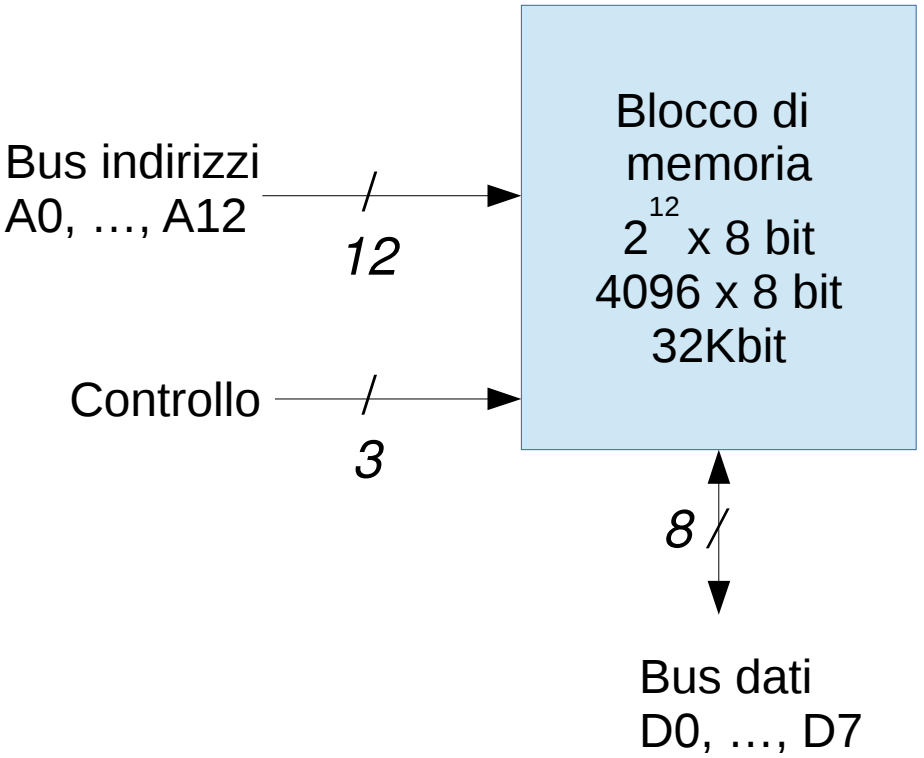


Struttura memorie SRAM e DRAM

- Sul bus dati posso leggere/scrivere l'intera riga selezionata
- Esempi:
  - 4096 x 8 bit
    - → 4096 righe, 8 colonne
    - → 32768 bit = 32 Kbit
  - 512k x 16 bit
    - → 524228 righe, 16 colonne
    - → 8388608 bit = 8 Mbit

1 Kbit = 2^{10} bit = 1024 bit

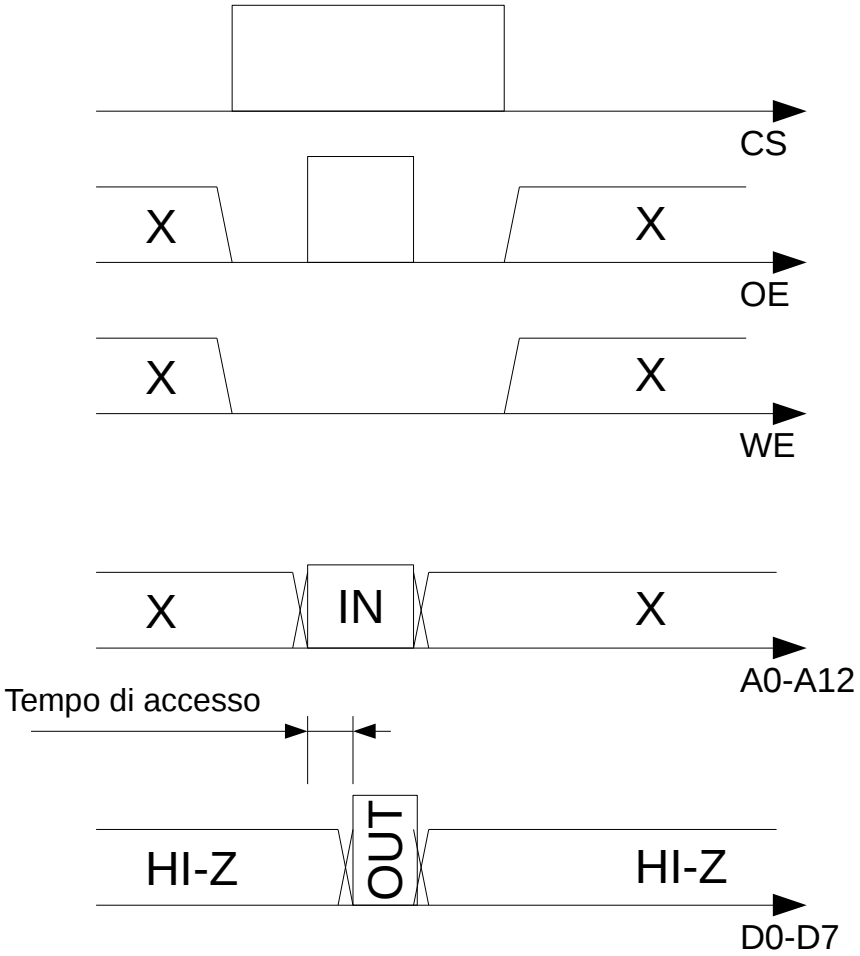
1 Mbit = 2^{20} bit = 1048576 bit

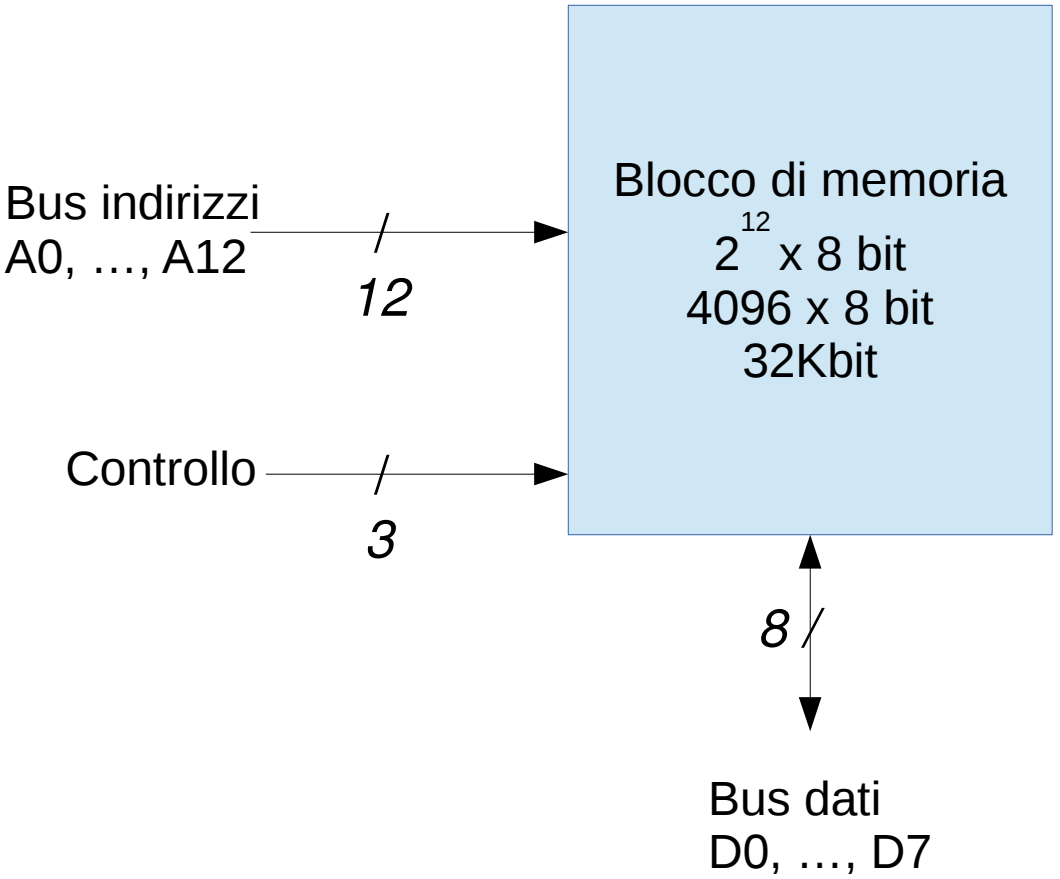


Interfacciamento con memorie RAM
Segnali di controllo

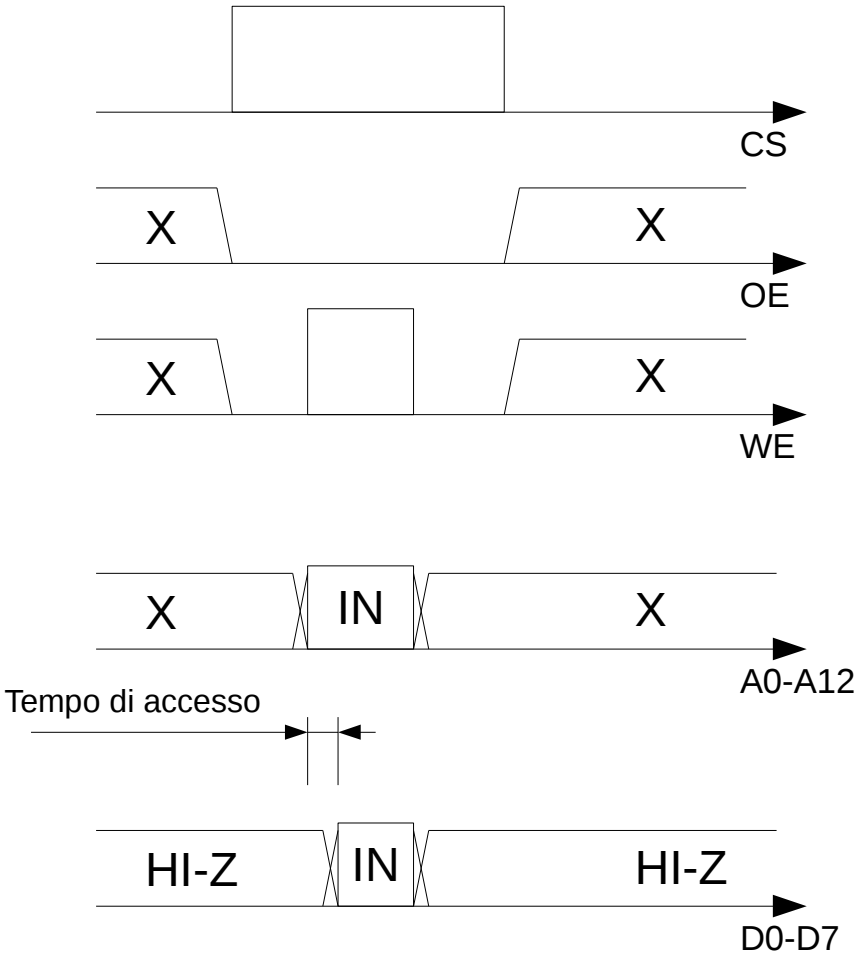
- Segnali di controllo
  - **CS – Chip Select** – abilita la comunicazione con il blocco di memoria
  - **OE – Output enable**, abilita la lettura
  - **WE – Write enable**, abilita la scrittura
- Quando un blocco non è selezionato (CS inattivo) le linee dati vengono solitamente messe nello stato di **alta impedenza (HI-Z)**
  - Non forzano né 1 né 0
  - Stato equivalente a scollegare il blocco (impedenza equivalente molto alta)
  - Permette a diverse uscite di essere collegate insieme, a patto che una e una sola venga selezionata in ogni momento

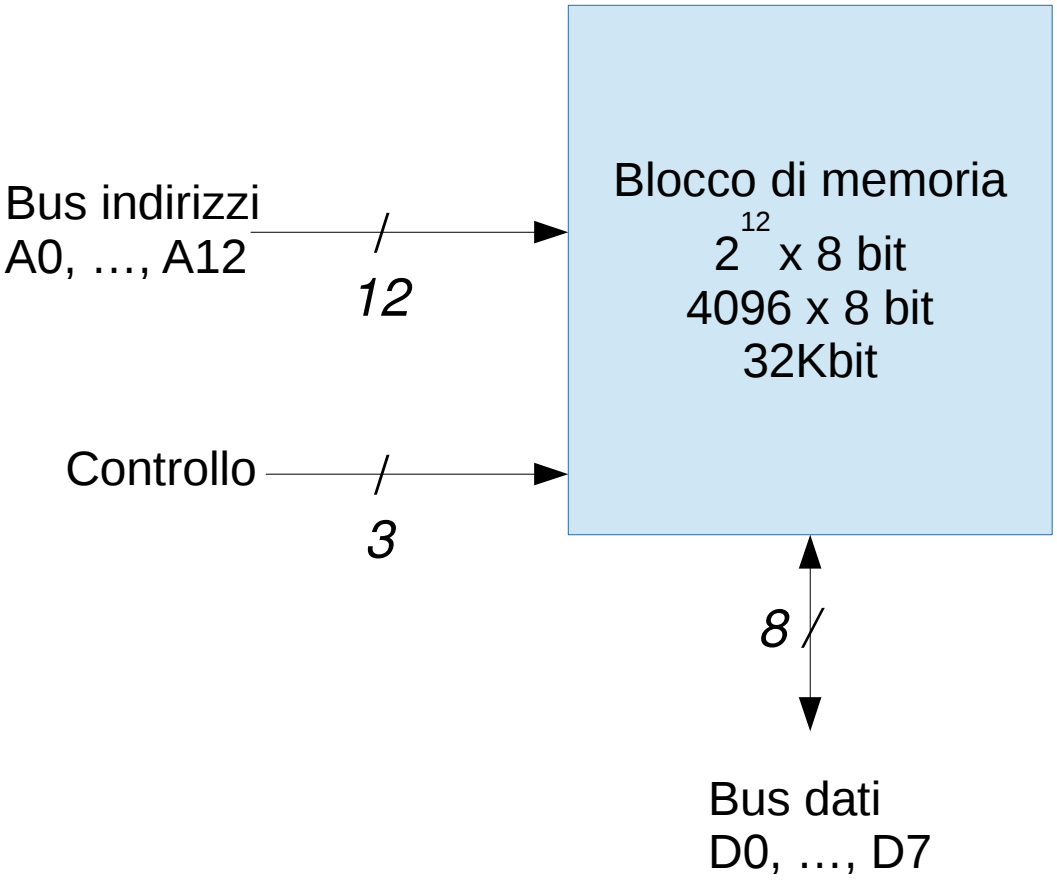
Interfacciamento con memorie RAM
Lettura





Interfacciamento con memorie RAM
Scrittura

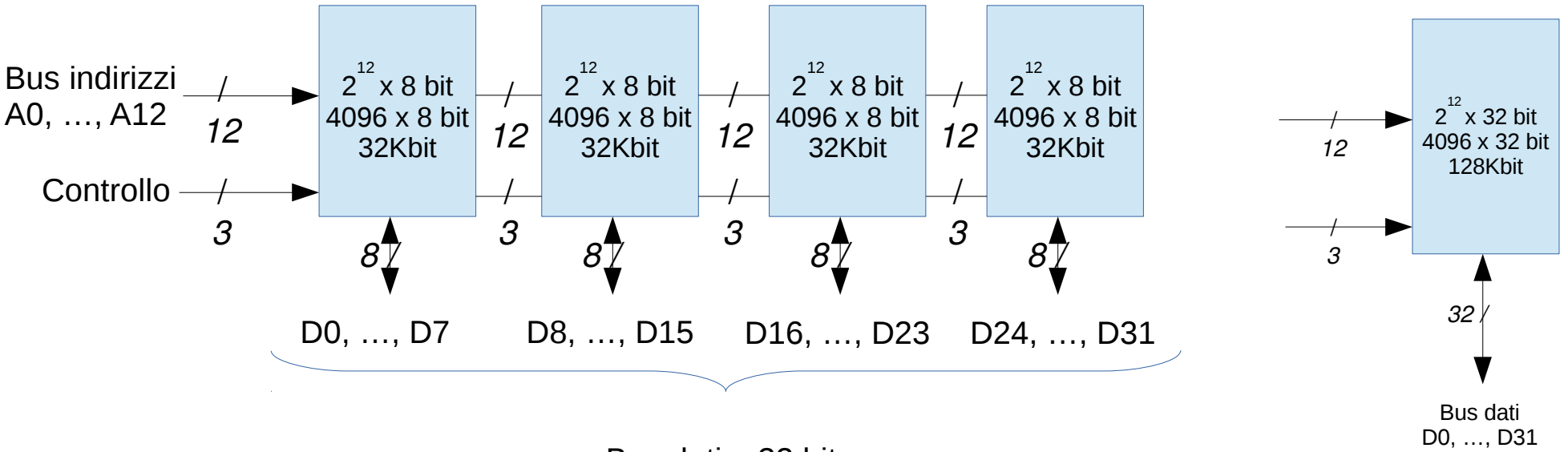




Organizzazione memoria

- Non è conveniente realizzare blocchi di dimensione troppo elevata
  - Decoder per selezione indirizzo di dimensione 2^n
    - Se avessi un blocco con bus indirizzi a 32 bit avrei un decoder con 2^{32} uscite → ~ 4 miliardi!
  - Posso ottenere memorie grandi usando blocchi di capacità piccola e diversi livelli di decoding
- Espansione bus dati
  - È conveniente usare un bus dati più ampio per aumentare prestazioni, ad esempio a 32 bit
  - Questo significa che ogni accesso in memoria trasferirà blocchi a 32 bit!

Organizzazione memoria



Totale modulo di memoria:
32 Kbit x 4 = 128 Kbit = 16 KByte
4K x 32 bit

Design memoria

- Le specifiche di una memoria sono:
  - **capacità** (solitamente espressa in byte)
  - **unità di indirizzamento** (quanti bit mi identificano un indirizzo)
  - **unità di lettura/scrittura** (quanti bit accedo con una singola lettura/scrittura, solitamente pari ad una parola dell'architettura)
  - **Indirizzo di mappatura** (dove si trova la memoria nella mappatura fisica della CPU)
- Si procede in 4 passi:

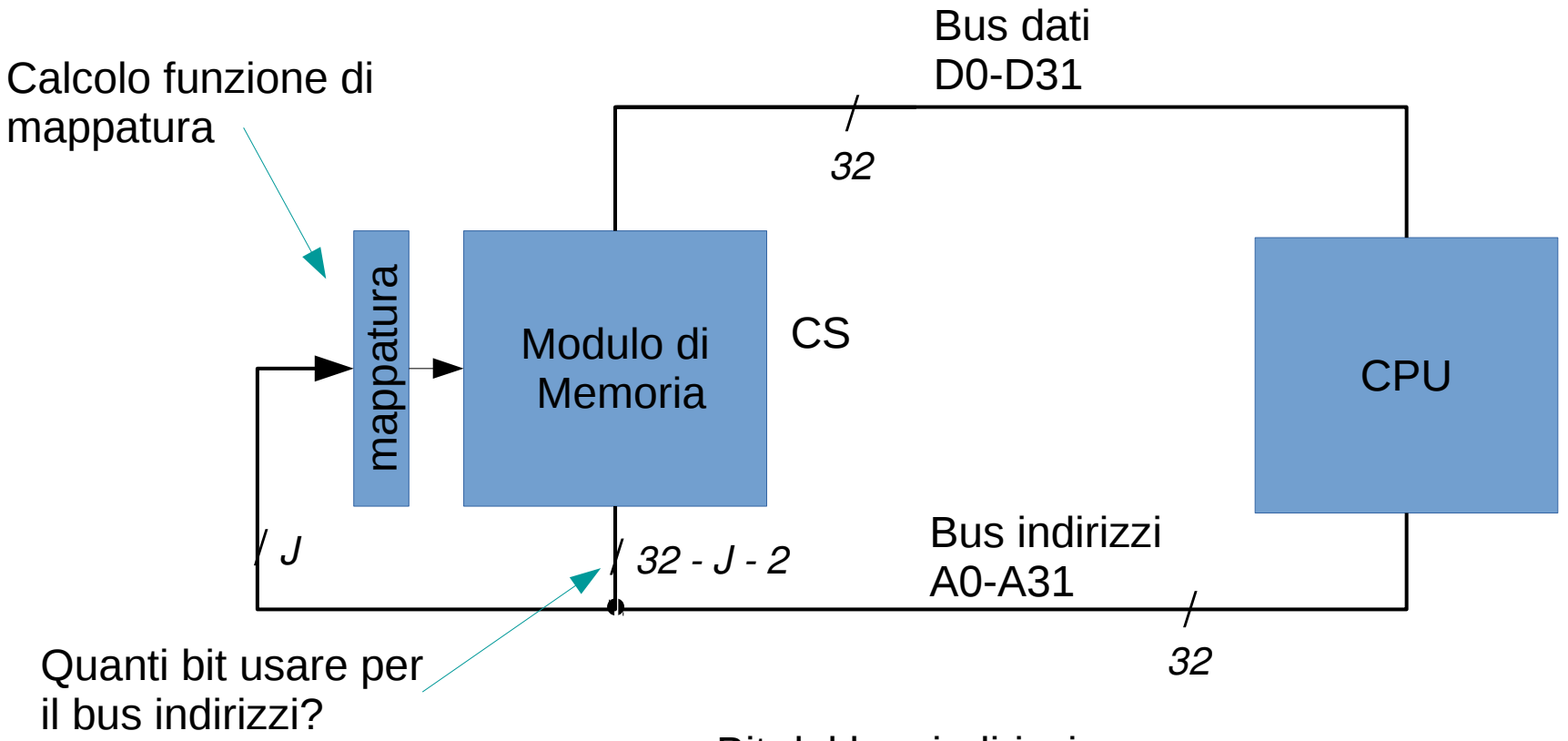
  1 - Calcolo **numero chip** necessari - Dipende dal tipo di chip e dalla memoria totale richiesta

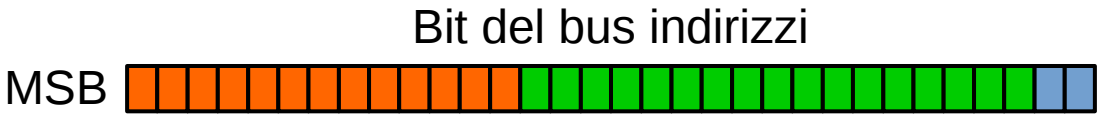
  2 - Calcolo **numero set** - Dipende dalla dimensione del bus dati e include calcolo del numero di chip per set

  3 - Calcolo **dimensione bus indirizzi** - Dipende dalla memoria totale richiesta e dalla dimensione del bus dati

  4 - Calcolo **mappatura** - Calcolo il valore dei bit di maggior peso per "abilitare" la memoria
    - Solitamente implementato nel controllore lato CPU e non nel modulo (es. DDR, DIMM)

Collegamento CPU MIPS ↔ memoria dati



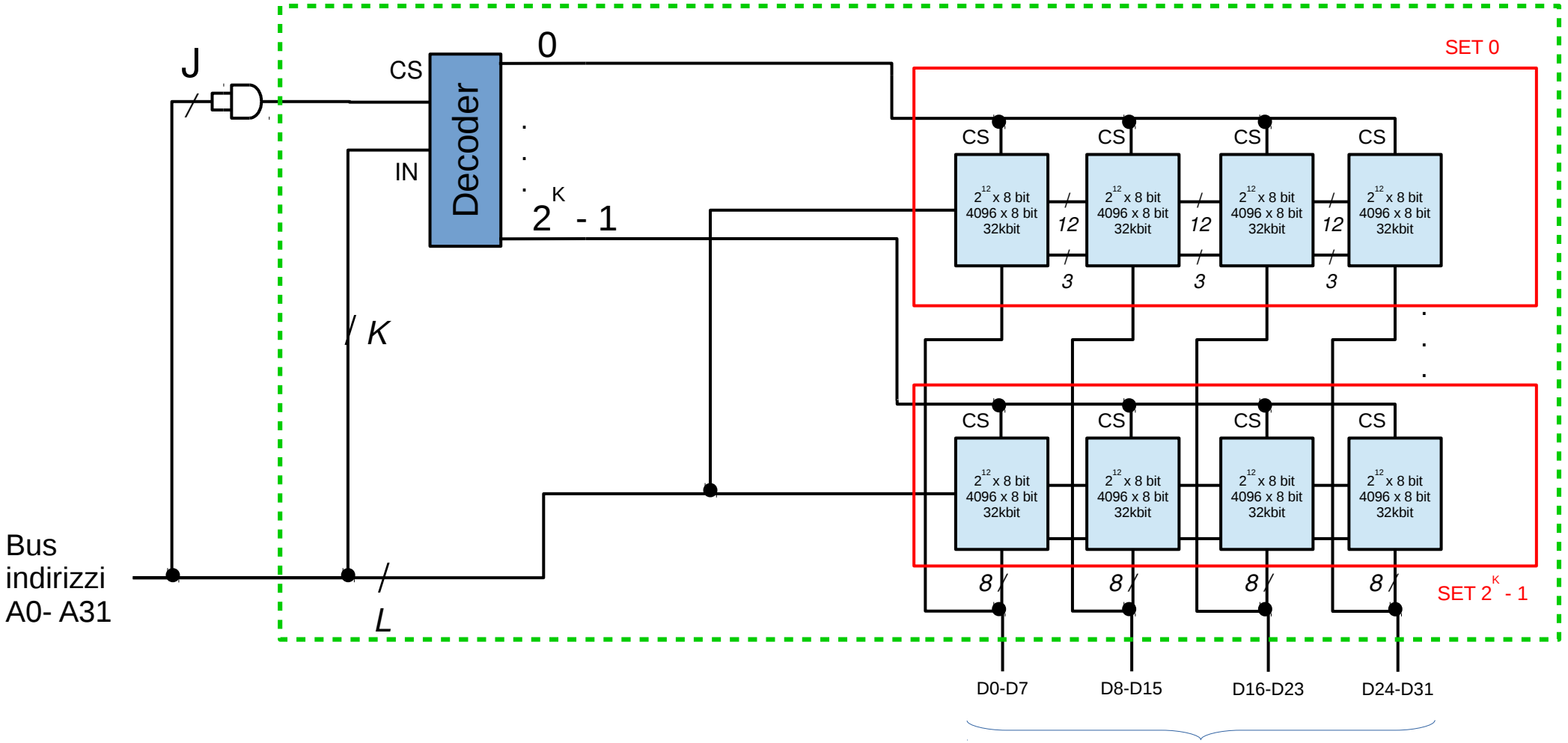


Dovuti a Mappatura in memoria

Dimensione Bus indirizzi per la memoria

Dovuti a Dimensione bus dati (2 per 32 bit)

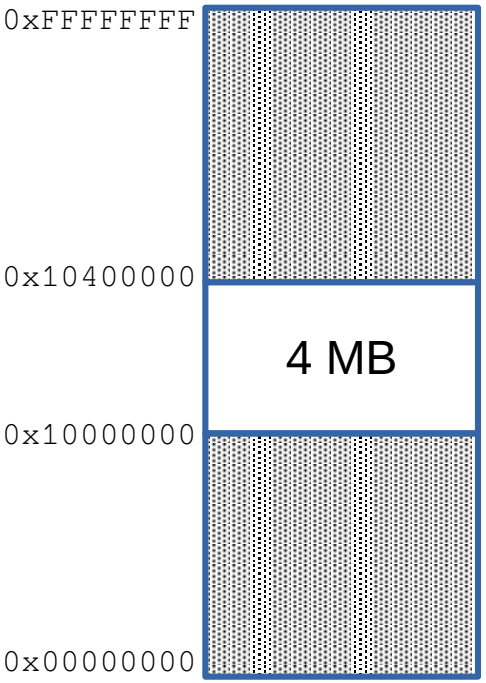
Schema generale organizzazione e mappatura memoria



J bit → mappatura in memoria
K bit → selezione set
L bit → selezione riga nel singolo set

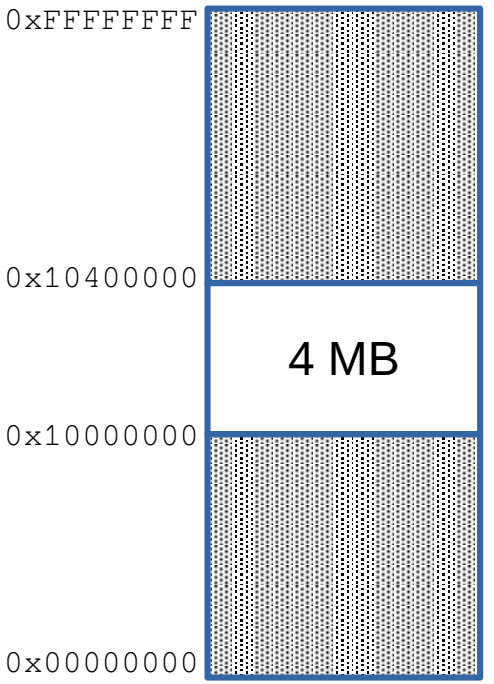
Design Memoria – Esempio 1

- Realizzare una memoria di 4 MB (2^{22} byte) con 32 bit di lettura/scrittura e unità di indirizzamento di 32 bit, utilizzando chip da 512K x 8 bit e mappando la memoria all'indirizzo 0x10000000.
- Quanti chip mi servono?
  - 4 MB → 32 Mb (bit totali della memoria)
  - 512K x 8 → 4 Mb (bit del singolo chip)
  - abbiamo bisogno di 32 / 4 = **8 chip**
- Quanti chip per set?
  - 32 bit di lettura/scrittura sul bus dati
  - ogni chip ha 8 bit di lettura/scrittura (512K x 8)
  - ogni set contiene 32 / 8 = **4 chip per set**
  - Avendo 8 chip totali abbiamo bisogno di 8 / 4 = 2 set



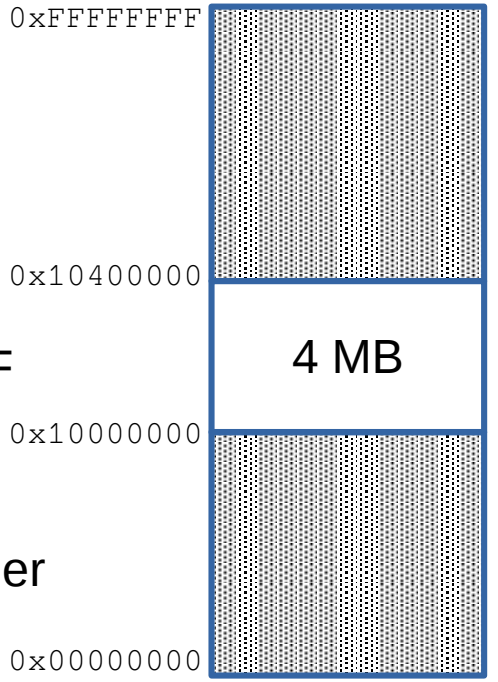
Design Memoria – Esempio 1

- Realizzare una memoria di 4 MB (2^{22} byte) con 32 bit di lettura/scrittura e unità di indirizzamento di 32 bit, utilizzando chip da 512K x 8 e mappando la memoria all'indirizzo 0x10000000.
- Da quanti bit è formato ogni indirizzo?
  - 4 MB → 32 Mb (bit totali della memoria)
  - 32 Mb / 32 b (unità di indirizzamento) = 1024K = 1M celle
  - $\log_2$ 1024K = 20
    → ogni indirizzo è composto da **20 bit**



Design Memoria – Esempio 1

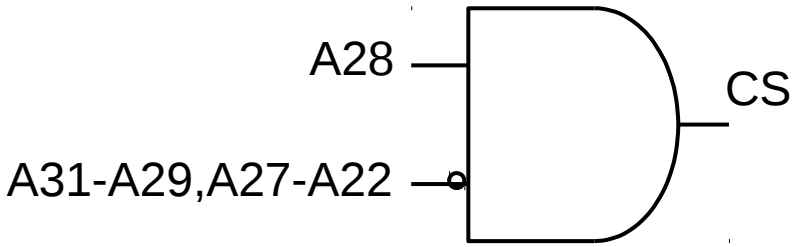
- Realizzare una memoria di 4 MB (2^{22} byte) con 32 bit di lettura/scrittura e unità di indirizzamento di 32 bit, utilizzando chip da 512K x 8 e mappando la memoria all'indirizzo 0x10000000.
- Calcolo mappatura
  - Avendo bus dati a 32 bit (2^2) forzo i 2 bit meno significativi a 0, per avere sempre indirizzi allineati
  - Ho $2^{32 - 20 - 2} = 2^{10} = 1024$ possibili mappature, ognuna con offset di 4MB = 0x00400000
  - L'offset di mappatura 0x10000000 sarà l'indirizzo iniziale della memoria, l'indirizzo finale sarà 0x103FFFFF
  - I 10 bit di maggior peso devono essere impostati a 0b0001000000xxxxxxxxxxx…
  - Sarà necessaria una rete logica che attivi il CS del decoder solo con questa combinazione

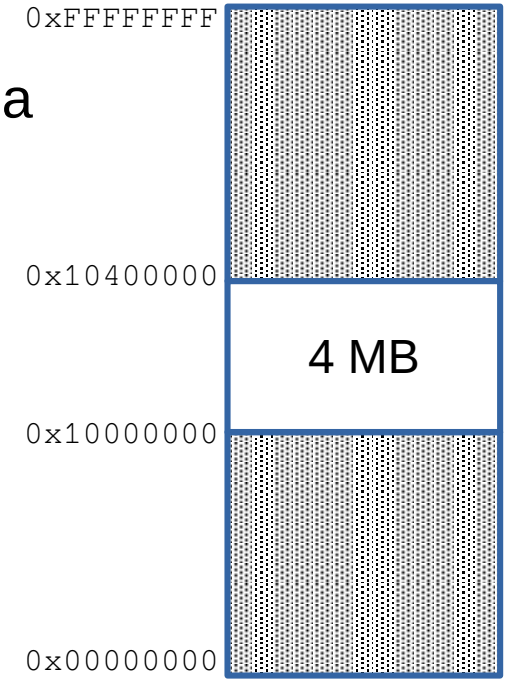


Design Memoria – Esempio 1

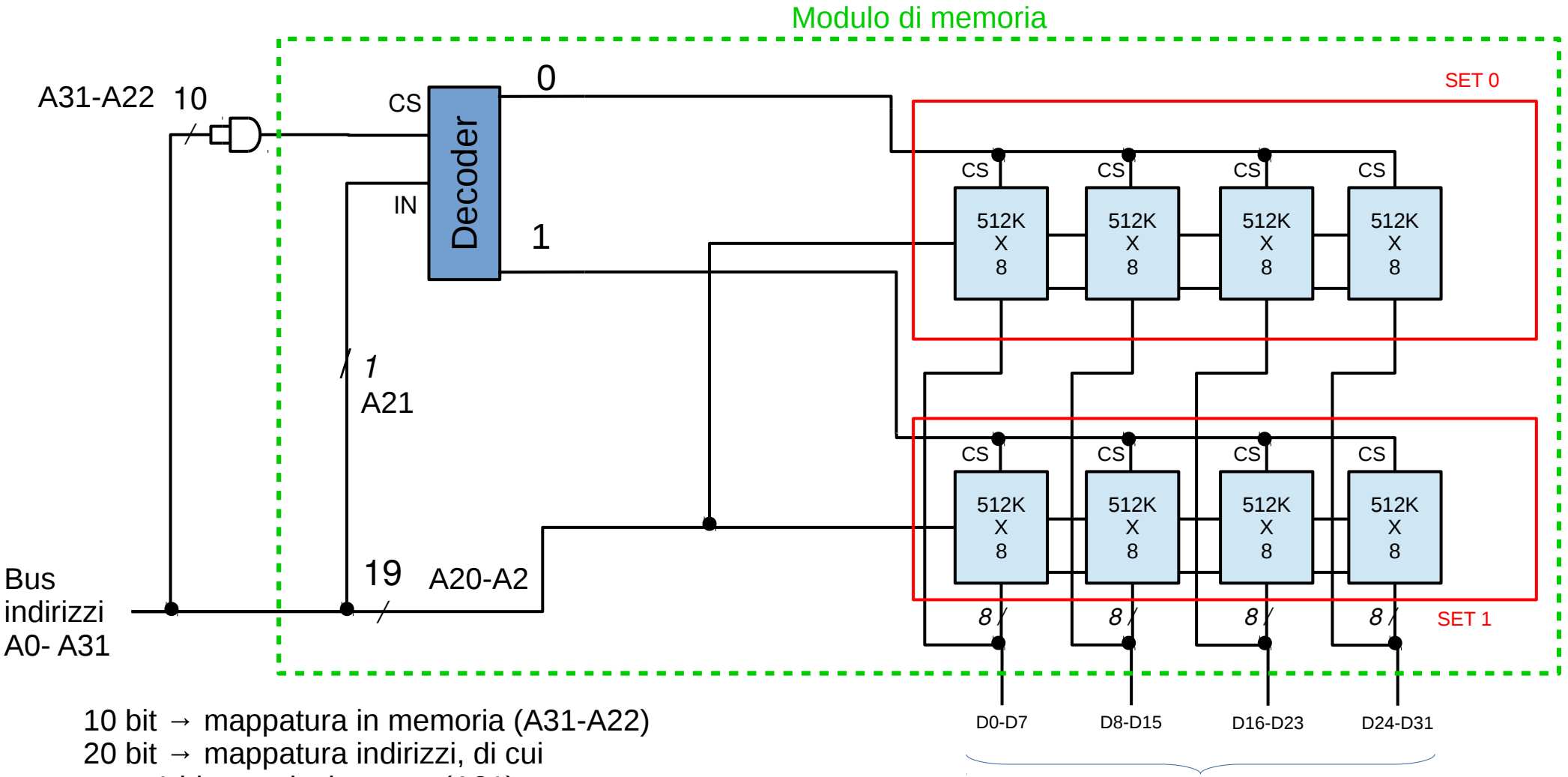
- Realizzare una memoria di 4 MB (2^{22} byte) con 32 bit di lettura/scrittura e unità di indirizzamento di 32 bit, utilizzando chip da 512K x 8 e mappando la memoria all'indirizzo 0x10000000.
- Calcolo mappatura (2)
  - I 10 bit di maggior peso devono essere impostati a 0b0001000000xxxxxxxxxx…

$$F_{map} = \overline{A}_{31} \cdot \overline{A}_{30} \cdot \overline{A}_{29} \cdot A_{28} \cdot \overline{A}_{27} \cdot \overline{A}_{26} \cdot \overline{A}_{25} \cdot \overline{A}_{24} \cdot \overline{A}_{23} \cdot \overline{A}_{22}$$





Esempio 1 - risultato



10 bit → mappatura in memoria (A31-A22)
20 bit → mappatura indirizzi, di cui
1 bit → selezione set (A21)
19 bit → selezione riga nel singolo set (A20-A2)
2 bit → non usati perché bus dati a 32 bit (sempre 0)

Design Memoria – DIMM ed ECC

- L’esempio appena descritto rappresenta il normale schema di collegamento di singoli chip di memoria per la realizzazione di un modulo DIMM (Dual In-line Memory Module).
- Esiste anche la variante ECC (Error-Correcting Code) dove si usa un bus dati più largo del richiesto per riuscire a rilevare e correggere alcuni errori
  - I bit in più sono calcolati attraverso un codice sulla base degli altri bit (es. codici di Hamming, Reed-Solomon)
- Ad esempio per realizzare un modulo DIMM di capacità 1 GB (organizzato 128M×64) possiamo usare 8 chip con capacità 1024 Mb (organizzazione 128M×8).
  - 8 bit x 8 = **64 bit**, caso **non ECC**
  - 8 bit x 9 = **72 bit** → 64 bit + 8 per controllo errori, caso **ECC**