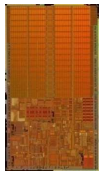
Architettura dei Calcolatori

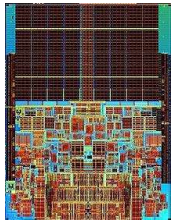
4-bit CPU (10 μm)
12 mm^2
Intel-4004 (11/1971)
0.0023Mtr

Willamette Core (0.18μm)
217 mm^2
Intel-Pentium-4 (11/2000)
42Mtr

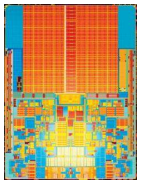
Northwood Core (0.13μm)
145 mm^2
Intel-Pentium-4 (01/2002)
55Mtr



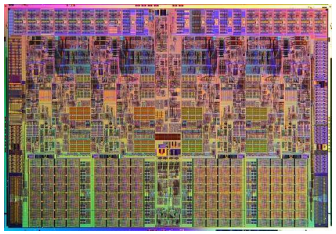
Dothan Core (0.09μm)
84 mm^2
Intel-Pentium-M (05/2004)
140Mtr



Conroe Core (0.065μm)
143 mm^2
Intel-Core2-Duo (07/2006)
291Mtr

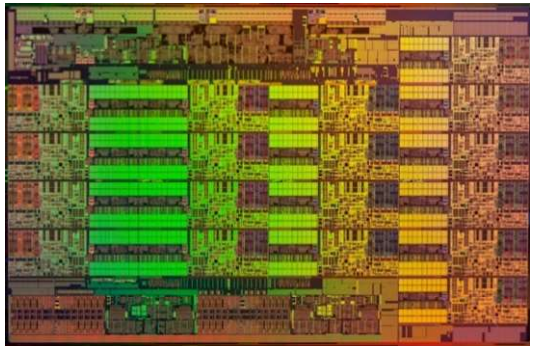


Penryn Core (0.045μm)
107 mm^2
Intel-Core2-Duo (01/2008)
410Mtr

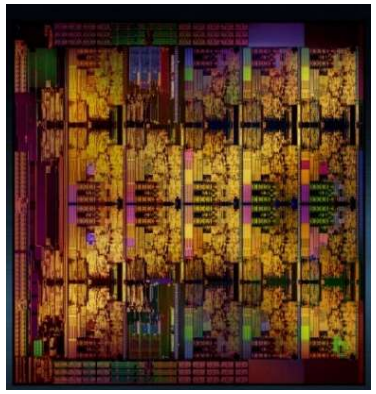


Bloomfield, 4 Cores (0.045μm)
263 mm^2
Intel-Core-i7 (11/2008)
Nehalem microarchitecture
731Mtr

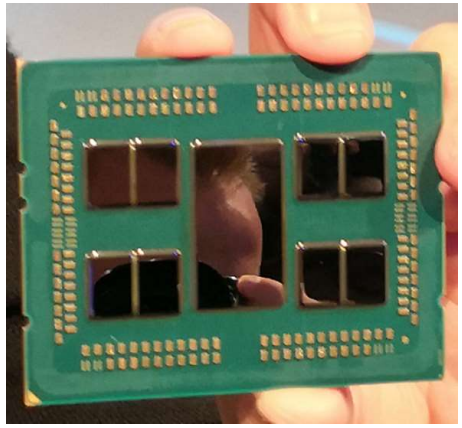
Ivy Bridge 4 Cores (0.022μm) tri-gate
160 mm^2
Intel-Core-i73770 (04/2012)
Ivy-Bridge microarchitecture
1'400Mtr



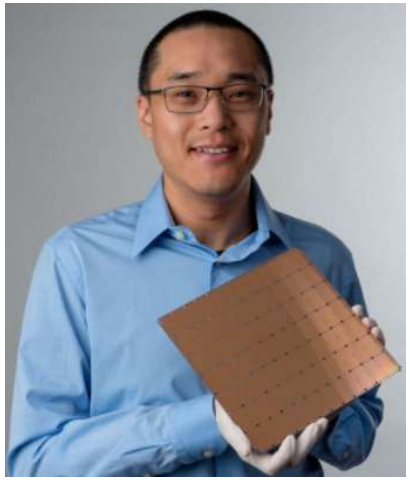
Haswell-EP, 18 Cores/2SMT (0.022 μm)
661 mm^2
Xeon E5-2600 v3 (06/2015 →7000$)
5'560Mtr



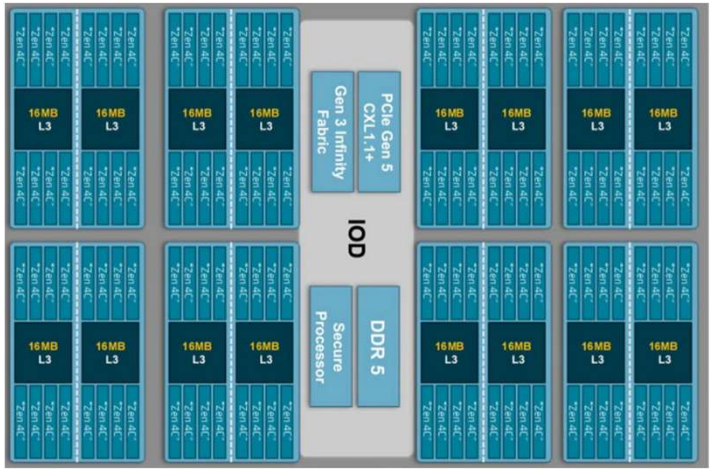
Skylake-X, 18 Cores/2SMT (0.014 μm)
473 mm^2
Intel Core i9-7980XE (09/2017 → 1900$)
??10'000Mtr



AMD Epyc Rome, 64 Cores/2SMT (0.007 μm) → 9 chiplets x86, 1 chip I/O
1'008 mm^2→ CHIPLETS
AMD EPYC-7742 (08/2019 → 7000$)
39'540Mtr



CERABRAS, 400'000 Cores (0.016 μm)
46'225 mm^2 →WAFER-SCALE
CEREBRAS CS-1 (09/2019 → 2.5 M$)
1'200'000Mtr

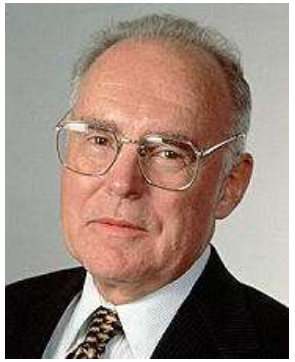


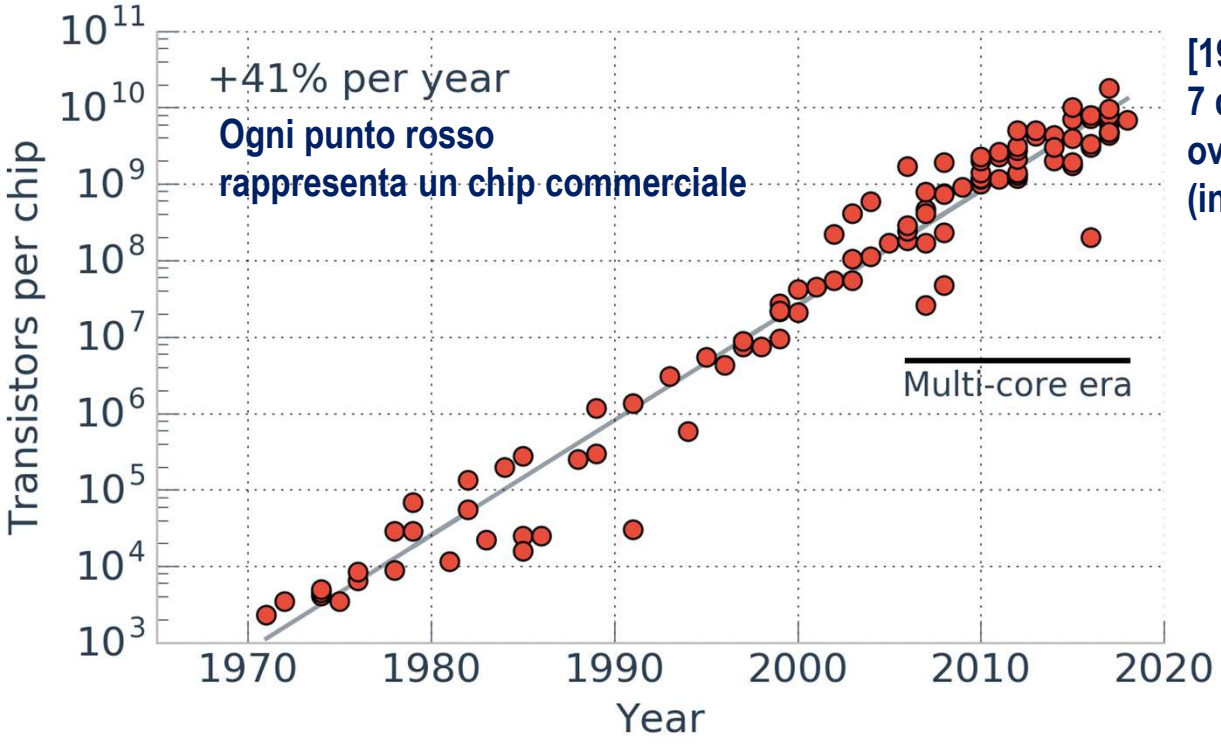
AMD Epyc Bergamo, 128 Cores/2SMT (0.005 μm) → 9 chiplets x86, 1 chip I/O
700 mm^2→ CHIPLETS
AMD EPYC-9754 (06/2023 → 12'000$) "Zen4c"
82'000Mtr

Vedi anche: Hennessy, J.L. and Patterson, D.A. 2019. **A new golden age for computer architecture**. Communications of the ACM. 62, 2 (Jan. 2019), 48–60. DOI: https://doi.org/10.1145/3282307

Legge di Moore

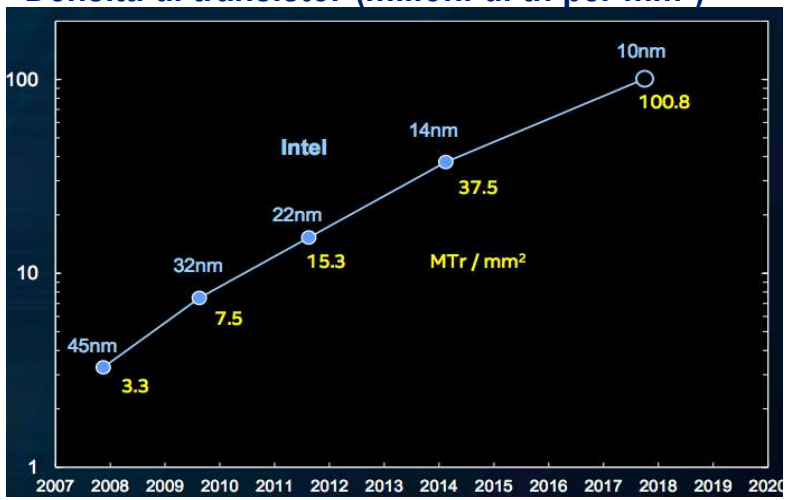
- **Il numero di transistor RADDOPPIA ogni 18 mesi** successivamente (modificato in "24 mesi")
- Consentito sia da una maggiore densità che da chip di maggiori dimensioni





[1971-2016] → 46anni*12mesi=552mesi
7 decuplicazioni circa → decuplica ogni ~79 mesi
ovvero raddoppia ogni $79/\log_2(10)$ =~ 24 mesi
(incremento annuo 10^(7/46) =~ 1.42 → 42% annuo)

Densità di transistor (milioni di tr. per mm^2)



Source:Intel

Obiettivi del Corso e Come Raggiungerli

1) Capire l'architettura dei moderni calcolatori

Analisi dell'organizzazione interna dei principali elementi:
processore, memoria, I/O, elementi di progettazione logica

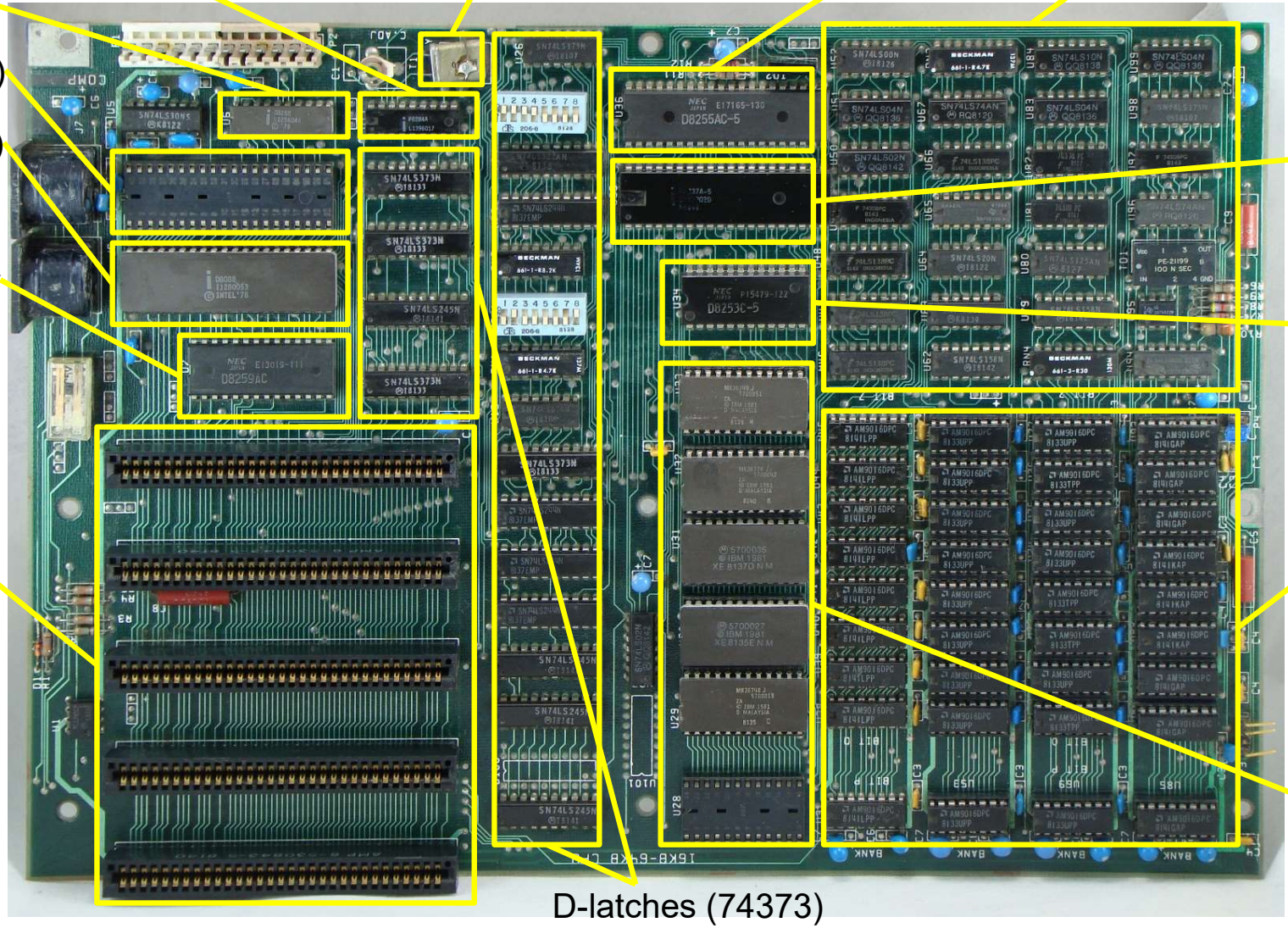
2) Saper scegliere un calcolatore esaminando i parametri che ne influenzano le prestazioni

Analisi quantitativa dei fattori che influenzano le prestazioni
e discussione su come le architetture incidono su tali fattori

3) Essere in grado di valutare l'efficacia dei meccanismi architetturali atti a migliorare le potenzialità dei calcolatori

Analisi di soluzioni architetturali non necessarie ma ormai
universalmente presenti quali la memoria cache

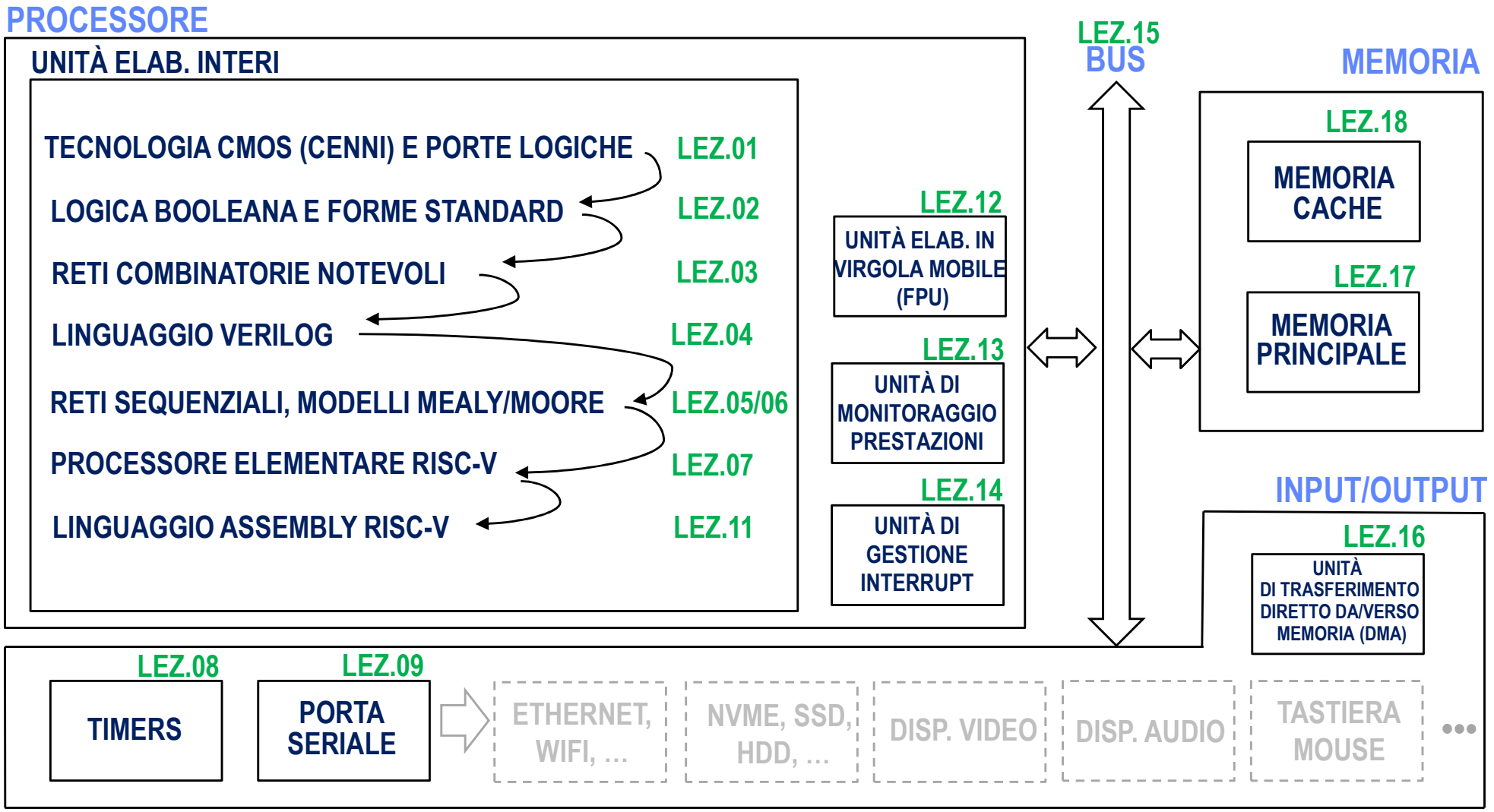
Il Personal Computer IBM-5150 (a.k.a. "PC" - Agosto 1981)



Source: https://el.m.wikipedia.org/wiki/%CE%91%CF%81%CF%87%CE%B5%CE%AF%CE%BF:IBM_PC_Motherboard_(1981).jpg

Analisi dell'architettura di un moderno calcolatore

- 4 elementi principali: processore, bus, memoria, input/output
  - Sempre gli stessi dal 1945: noti come «architettura di von Neumann»

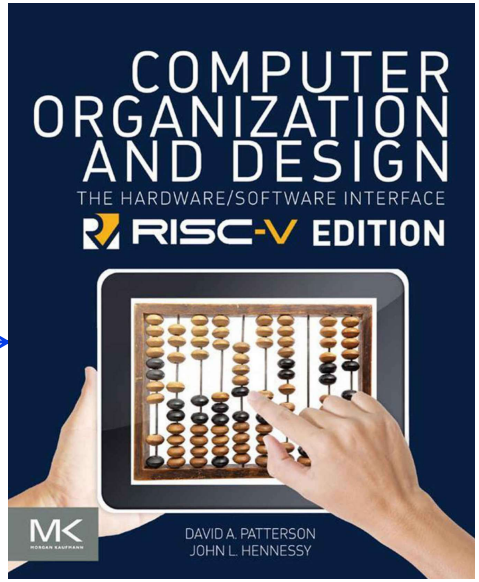


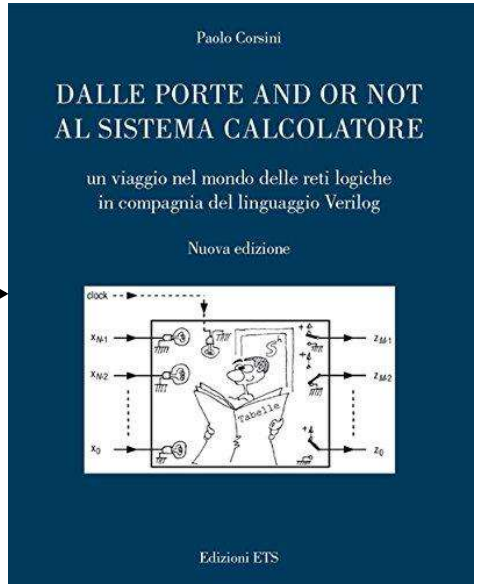
Conoscenze di base dai corsi precedenti

- Saper formalizzare un problema descritto in linguaggio naturale in modo che possa essere seguito dal calcolatore
- Saper leggere e saper scrivere semplici programmi *C*
- Struttura base della macchina
  - Modello di Esecuzione
    ovvero principi di una macchina astratta in grado di eseguire programmi (ciclo fetch-execute, struttura di Von Neumann)
- Capire i passi che vengono compiuti per generare un programma:
  - compilazione, link, caricamento e esecuzione

Amministrazione del Corso

- Docente: Roberto Giorgi (giorgi@unisi.it → NON ABUSARE)
- Telefono: 0577-23-5880
- Ricevimento: Lunedi 16:30/19:00
- Sito del corso: https://arcal.diism.unisi.it
- Testi di Riferimento:
  - *Patterson and Hennessy, Computer Organization and Design RISC-V Edition: The Hardware/Software Interface, Morgan Kauffman, 2017*
  - Edizione in Italiano (basata sulla precedente ed. in inglese): Patterson e Hennessy, *Struttura e Progetto dei Calcolatori – Progettare con RISC-V*, Zanichelli, 2019
  - P. Corsini, "Dalle porte AND OR NOT al sistema calcolatore", Edizioni ETS, 2015 (o 2020)





Materiale Didattico (v. https://arcal.diism.unisi.it)

- Sono disponibili tutte le slide presentate durante le lezioni

  N.B. LE SLIDE COSTITUISCONO UNA TRACCIA DELLA LEZIONE
  CHE DEVE ESSERE INTEGRATA CON LA PARTECIPAZIONE ATTIVA ALLE LEZIONI,
  CON LE ESERCITAZIONI E CON LO STUDIO PERSONALE

- Sono disponibili tutte le videolezioni dell'A.A. 2020-21
  - Il numero della lezione può variare rispetto all'A.A. attuale
- Per ogni lezione è indicato
  in quale parte dei testi di riferimento studiare l'argomento
- Sono disponibili moltissimi testi di compiti precedenti
  - molti dei quali svolti o con una traccia dello svolgimento
    e con molti programmi in assembly e Verilog da poter provare
- Sono disponibili tool didattici quali simulatori ed altro
  - per poter sperimentare personalmente quanto appreso
- Le informazioni sono aggiornate costantemente sul sito del corso

Formato dell'insegnamento

- LEZIONI+ ESERCITAZIONI (totale 60 ore)
  - Circa 30 ore di teoria, 30 ore di esercitazioni/prove_in_itinere
- Il programma segue molto da vicino i più elevati standard internazionali
  - https://www.computer.org/cms/Computer.org/professional-education/curricula/ComputerEngineeringCurricula2016.pdf
  - CE-CAO Computer Architecture and Design (circa 40 ore)
  - CE-DIG Digital Design (circa 20 ore)
- Useremo un approccio moderno per lo studio dei "Sistemi Digitali" con ampio riferimento al linguaggio Verilog
- Argomenti di Progettazione Architetturale e Logica saranno intercalati durante lo svolgimento delle lezioni
- In via SPERIMENTALE per il 2023-24 sono stati eliminati i seguenti argomenti (circa -20% rispetto dal 2022-23, -30% dal 2021-22):
  - Contatori
  - Memoria virtuale
  - Pipeline
  - Vari approfondimenti

Modalità di Esame – Syllabus Ufficiale

- **Prova scritta seguita da orale**
  - Sia la prova scritta che la prova orale consistono in domande aperte riguardanti gli argomenti svolti nelle lezioni frontali, documentati sia nei libri di testo, nelle diapositive illustrate e approfonditi nelle esercitazioni; la prova scritta viene svolta in laboratorio, ove ci si può avvalere del simulatore sia del linguaggio assembly che del linguaggio Verilog per una immediata verifica della correttezza dei corrispondenti esercizi; nella prova orale le domande sono del tutto analoghe a quelle dello scritto (e viceversa)
  - La prova scritta è superata con un voto non inferiore a 18/30. Il peso della prova scritta è il 50%, il peso della prova orale è il 50% del voto finale.
  - È caldamente consigliata la frequenza alle lezioni e esercitazioni; ove la frequenza non sia possibile, riferirsi al docente in caso di difficoltà
- **Prove in itinere (novità dal 2020-21)**
  - DUE prove in itinere
  - Il superamento delle prove scritte in itinere (media non inferiore a 18/30) permette l'esonero della prova scritta ai primi due appelli dell'anno accademico (la prova orale deve in ogni caso essere sostenuta)
- **Progetto facoltativo**
  - È possibile facoltativamente far valere un progetto di gruppo (max 3 persone) che contribuisce al voto complessivo fino a 5/30 in aggiunta al voto finale ottenuto con scritto e orale

Consigli per una miglior preparazione – Syllabus ufficiale

- Sia nelle prove scritte che orali (ma anche nei progetti), allo studente è principalmente richiesto di mostrare la conoscenza dettagliata dell'argomento, almeno al livello indicato dal docente durante la lezione. E' molto apprezzata la capacità di ragionamento sul problema, piuttosto che una meccanica (pedante) descrizione del tema

- Negli esercizi scritti, verrà principalmente considerata la correttezza della soluzione (in termini numerici) e una giustificazione molto breve del modo scelto per svolgere l'esercizio (lunghe formulazioni generali sono completamente inutili)

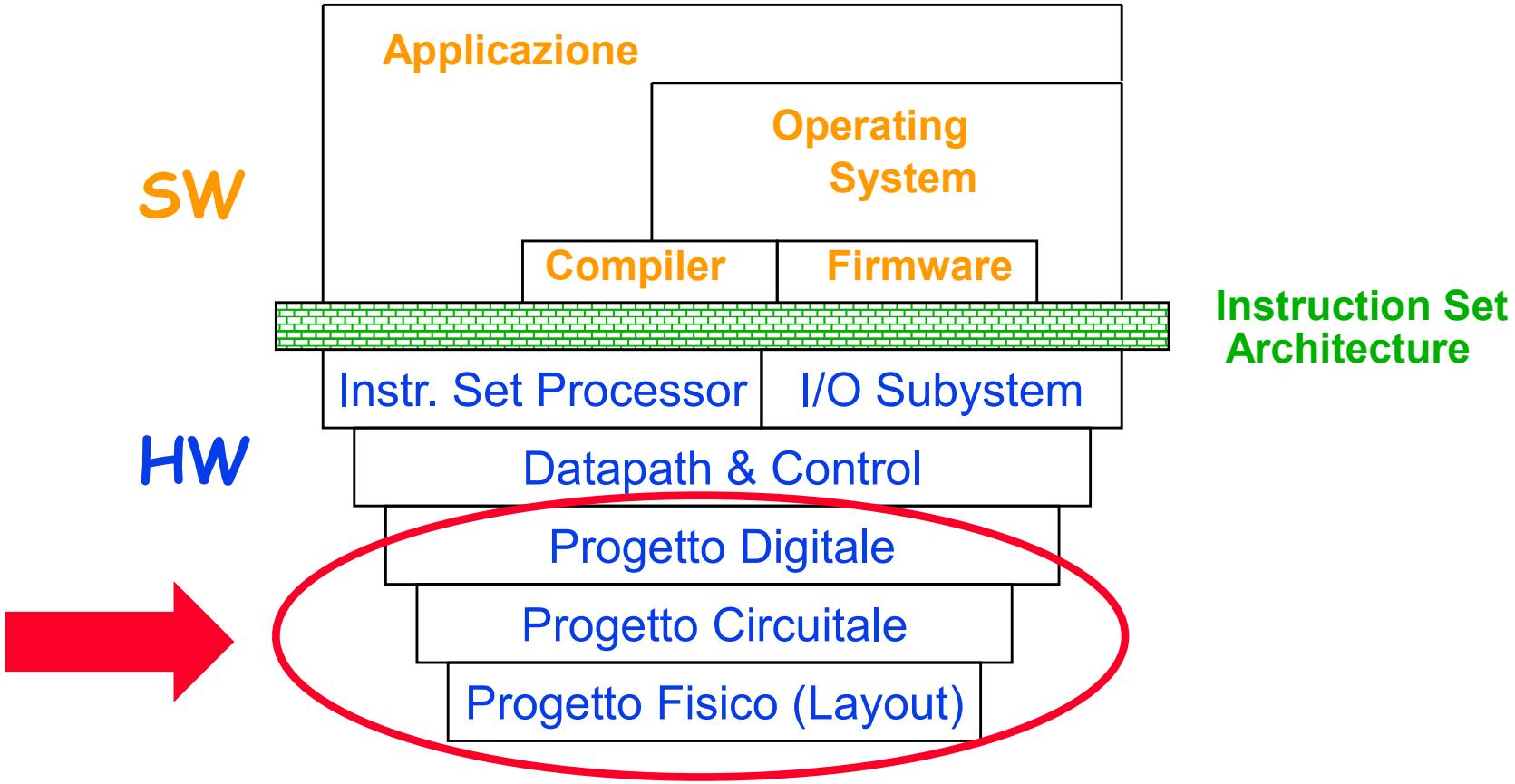
- Nell'interrogazione orale, l'argomento è in genere uno dei concetti illustrato durante la lezione. Elementi che sono richiesti sono, per esempio: la prova del concetto/teorema, schemi precisi del sistema, il comportamento e il funzionamento dettagliato, ragioni per le quali questa soluzione viene utilizzata nel mondo reale

Lezione 1
Dal transistor MOS alle Porte Logiche

https://arcal.diism.unisi.it

Dal livello fisico ai dispositivi logici

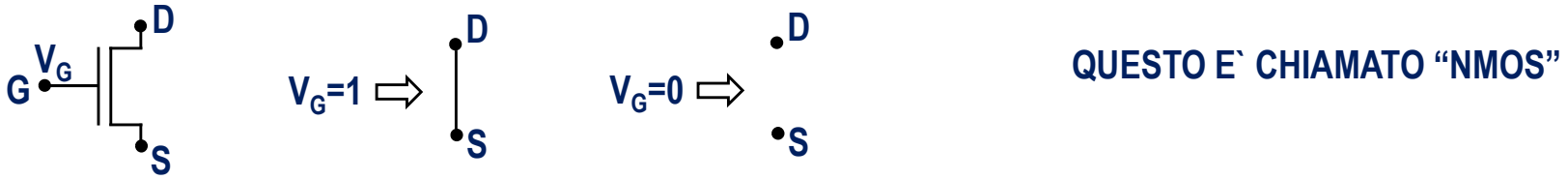
- Livelli di Astrazione di un Calcolatore



- Come sono correlati «progetto digitale» e «layout» ?
- Quali possibilità offre l'attuale tecnologia ?

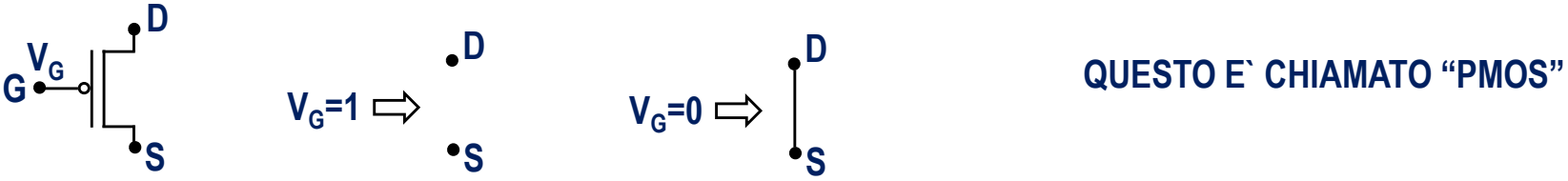
Il transistor MOS (Metal+Oxide+Silicon)

- Dettagli transistor MOS → insegnamento di Elettronica
  - Esistono di due tipi: NMOS e PMOS
- Possiamo assimilare il transistor MOS ad un interruttore
  - Quando la tensione di ingresso è alta l' NMOS conduce
  - Quando la tensione di ingresso è bassa l' NMOS è interdetto



QUESTO E` CHIAMATO "NMOS"

- Esiste anche il duale, chiamato PMOS
  - Quando la tensione di ingresso è alta il PMOS è interdetto
  - Quando la tensione di ingresso è bassa il PMOS conduce



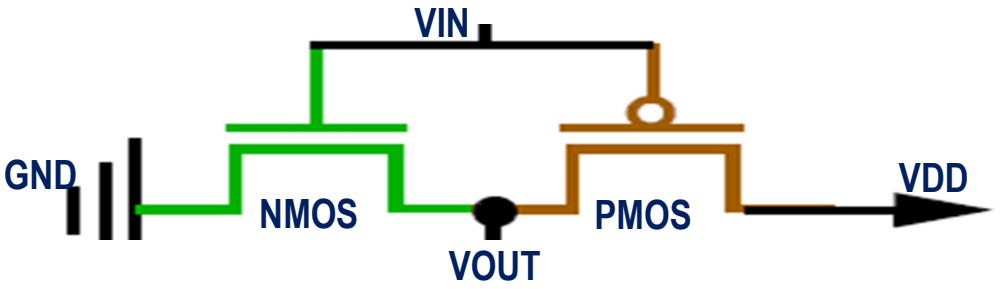
QUESTO E` CHIAMATO "PMOS"

- Usando contemporaneamente transistor NMOS e dualmente PMOS si ottiene un circuito CMOS (Complementary MOS)

L'inverter CMOS

- Il circuito CMOS più semplice è l'inverter
  - Quando la tensione di ingresso è alta l'uscita è bassa
  - Quando la tensione di ingresso è bassa l'uscita è alta

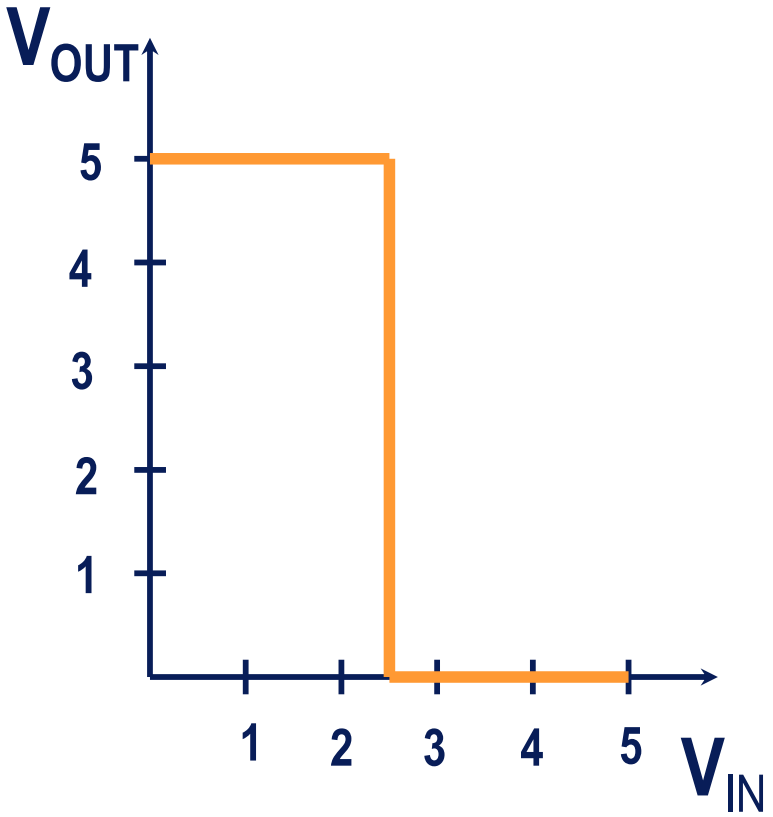
- Circuito elettrico:



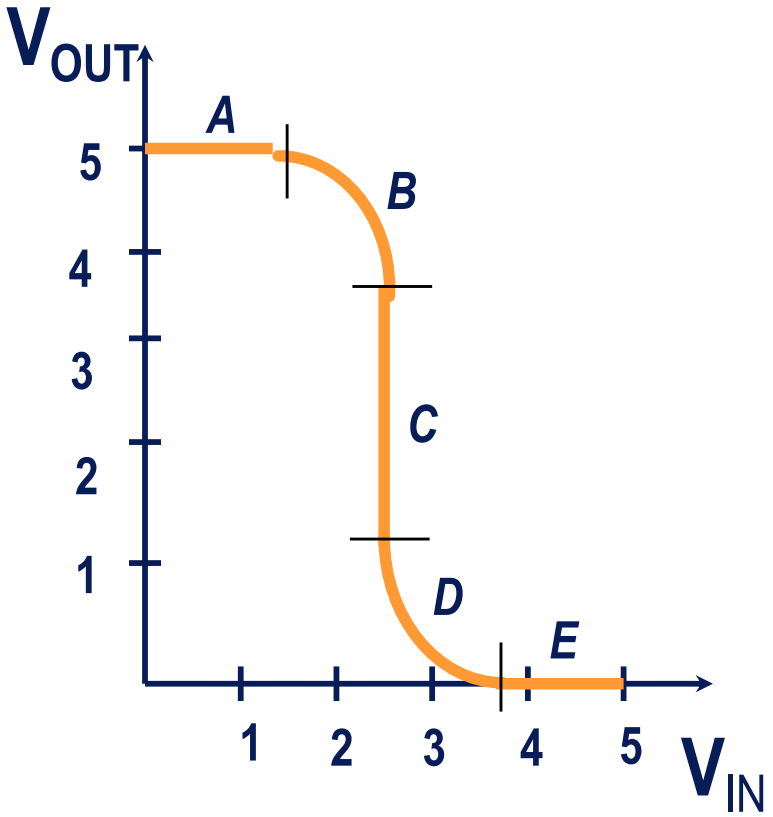
e logico:

CARATTERISTICA DI TRASFERIMENTO dell'INVERTER

Caratteristica IDEALE



Caratteristica REALE



Tratto di curva	Transistor NMOS	Transistor PMOS
A	Interdetto (VGS = Vin → bassa ovvero <Vth,n)	Conduce (ID ∝ Vin) → Vout collegata a VDD *
B	Saturo (VDS>VGS-Vth,n →ID ≅ costante)	Conduce (ID ∝ Vin) → Vout collegata a VDD *
C	Saturo (VDS>VGS-Vth,n →ID ≅ costante)	Saturo (VDS<VGS-|Vth,p| →ID ≅ costante) **
D	Conduce (ID ∝ Vin *) → Vout collegata a massa	Saturo (VDS<VGS-|Vth,p| →ID ≅ costante) **
E	Conduce (ID ∝ Vin *) → Vout collegata a massa	Interdetto (VGS = Vin-VDD → bassa ovvero >-|Vth,p|)

* Comportamento "resistivo": corrente proporzionale alla tensione

** Nota: La tensione di soglia Vth,n è positiva ma Vth,p è negativa

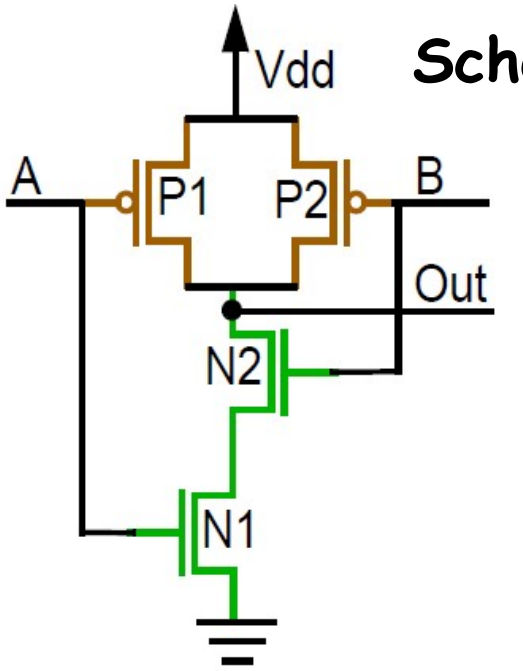
Osservazioni

- La tecnologia CMOS consente di realizzare la funzione logica più semplice (inverter o porta NOT) in modo tale che la sua caratteristica di trasferimento sia vicina alla caratteristica ideale
  - Altre tecnologie (es. NMOS, BJT) hanno alcune limitazioni (NMOS: livelli logici non pieni, BJT: alto consumo) ma possono offire alcuni vantaggi (NMOS: maggiore compattezza, BJT: maggiore velocità)
  - Attualmente la tecnologia CMOS è la tecnologia di gran lunga più utilizzata per realizzare sistemi digitali soprattutto per la sua efficienza energetica

- Per realizzare funzioni logiche più complesse ci viene in aiuto l'algebra booleana (v. lezioni successive)
  - In teoria oltre all'inverter ci servirebbero le funzioni elementari AND e OR
  - In pratica, le porte più semplici da realizzare a partire dall'inverter sono le porte NAND e NOR (v. slide successive)
  - Fortunatamente si può facilmente dimostrare che è possibile costruire qualsiasi funzione logica a partire anche dalle sole porte NAND (o NOR)
  - È immediato altresì vedere che NAND seguito da NOT equivale a AND e NOR seguito da NOT quivale a OR

Porta NAND CMOS

Schema elettrico



Schema logico

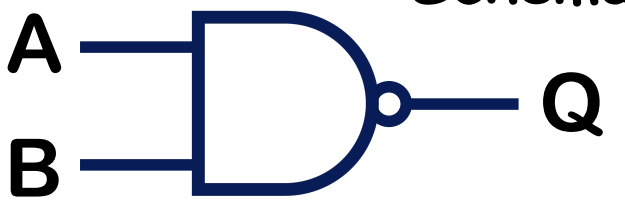


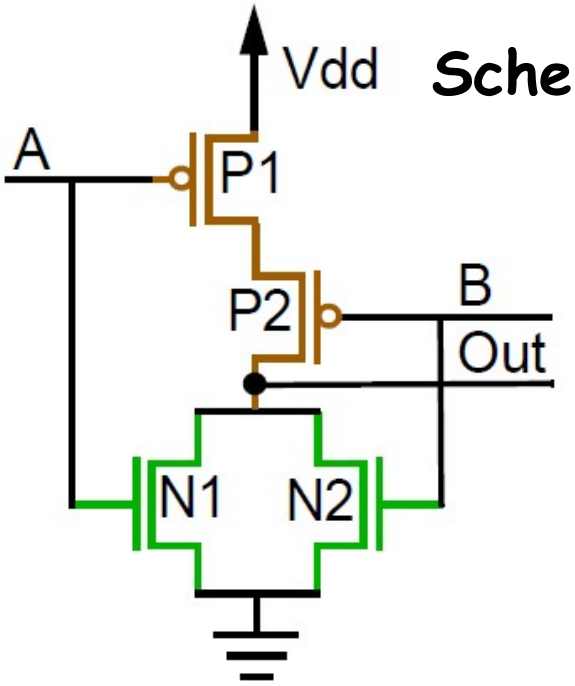
Tabella di Verità

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

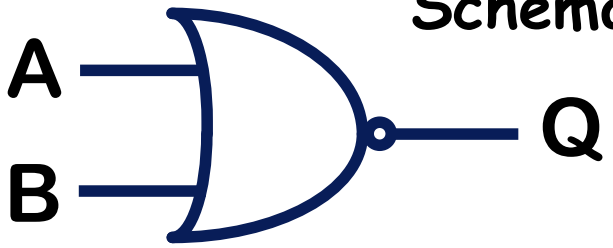
$Q=\sim(A \cdot B)$ Equazione booleana

Q=~(A & B) Verilog

Porta NOR CMOS



Schema elettrico



Schema logico

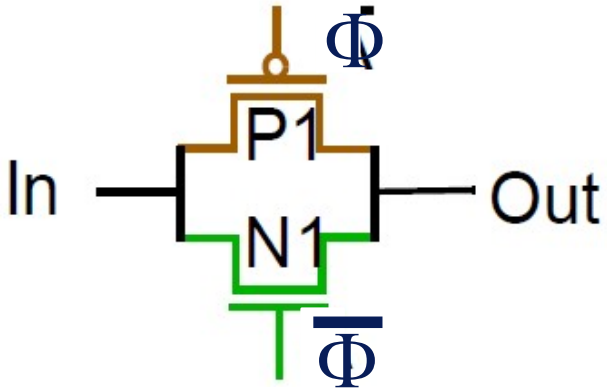
Tabella di Verità

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

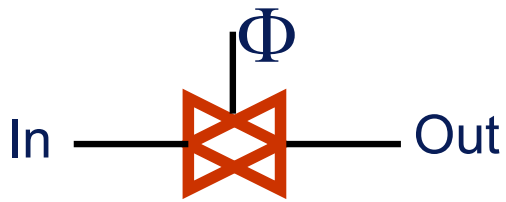
Q=~(A + B) Equazione booleana

Q=~(A | B) Verilog

Porta di passo/transito CMOS (transimission gate)



Schema elettrico



Simbolo

Nota: $\overline{\Phi}$ è sottinteso

Nota1: Φ e $\overline{\Phi}$ rappresentano le due fasi di un «clock a due fasi», ovvero di un'onda quadra Φ e della sua negata $\overline{\Phi}$ in cui però i due segnali non sono mai attivi contemporaneamente

Tabella di Verità

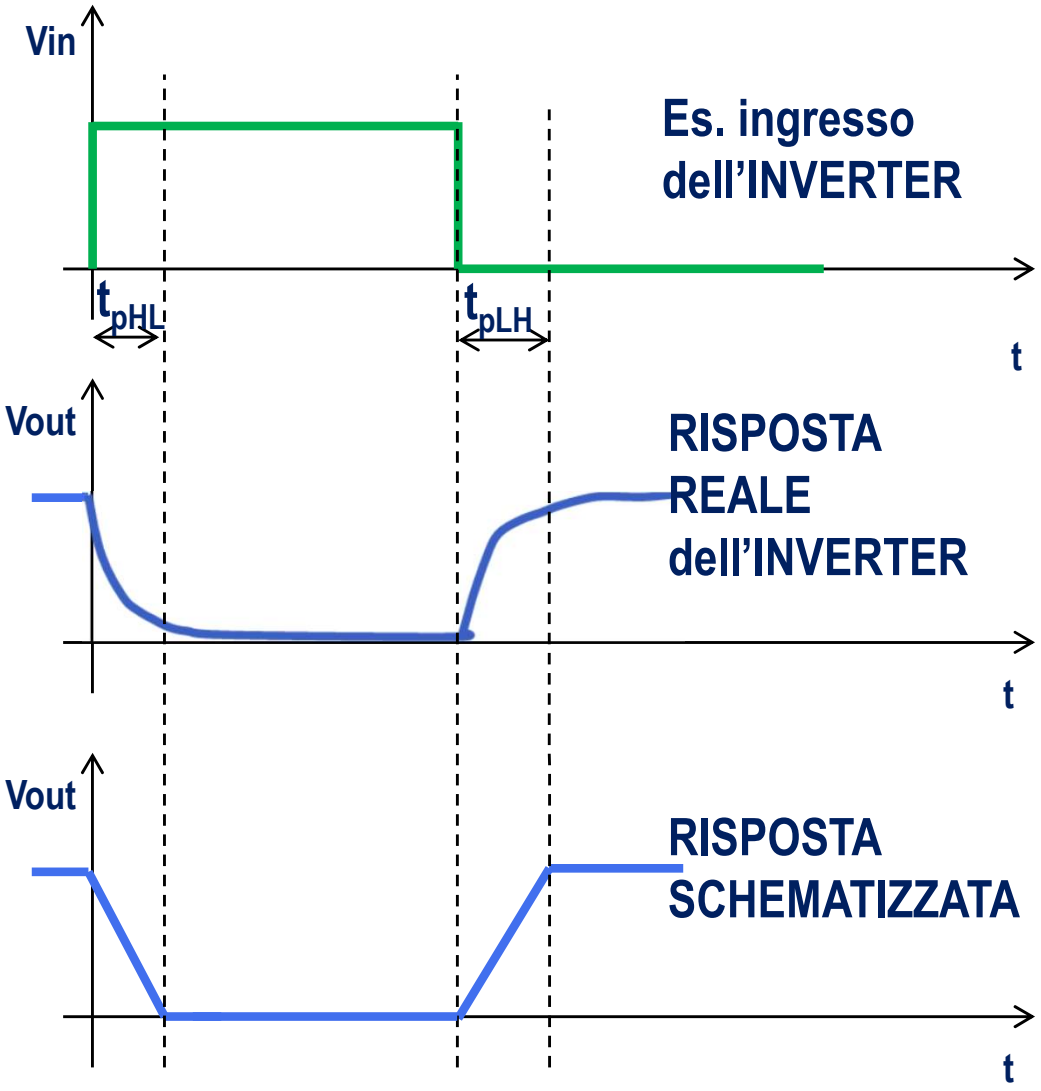
Φ	In	Out
0	X	Z
1	0	0
1	1	1

Nota2: non è un "tri-state" buffer in quanto quest'ultimo rigenera anche il segnale, mentre in questo caso il segnale si degrada un poco nel passaggio.

Nota2: viceversa si realizza un buffer mettendo due inverter in cascata

Ritardo di propagazione dell'inverter CMOS

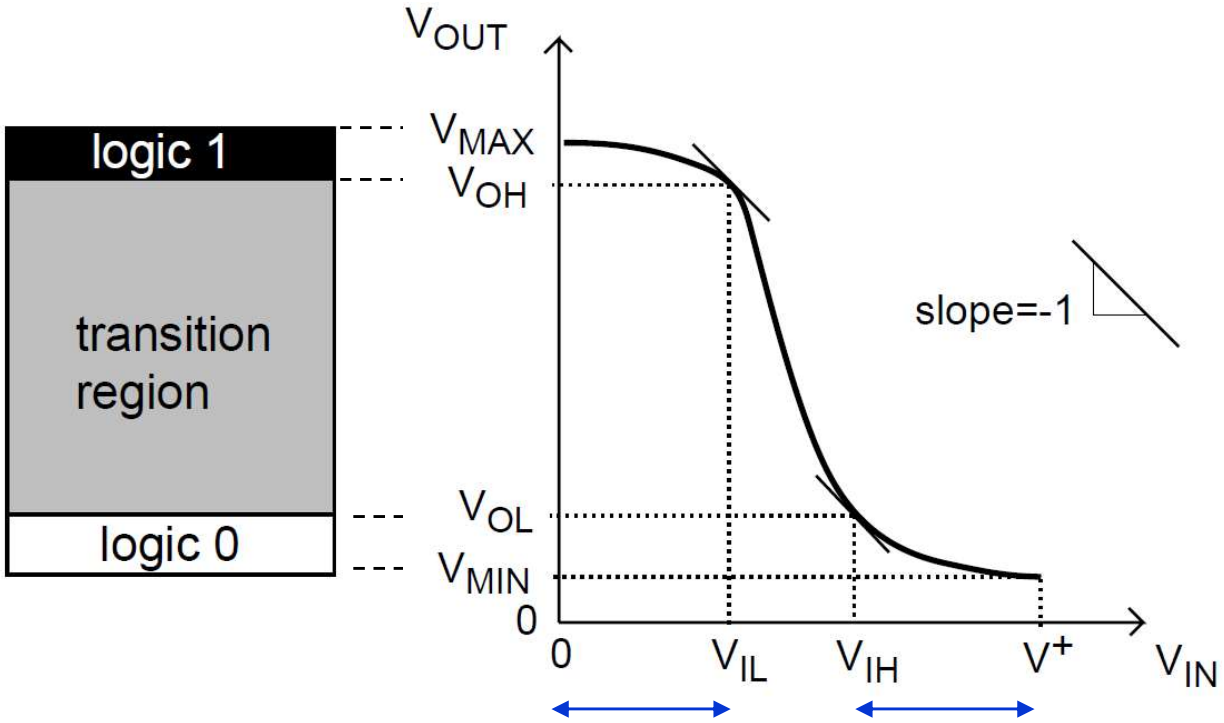
- Le porte fondamentali e di trasmissione presentano un certo ritardo di propagazione t_p (t_{pHL} quando l'uscita va da High a Low e t_{pLH} viceversa)



Applicando un segnale a gradino all'ingresso dell'inverter la forma d'onda di uscita raggiunge un livello accettabile (es. 90%) con un ritardo t_p
Il ritardo t_{PHL} necessario per la transizione dal livello alto a livello basso può essere diverso da quello t_{pLH} della transizione opposta

Approfondimenti ulteriori possono essere fatti nei corsi di Elettronica.
In questo contesto schematizzeremo il ritardo con una transizione lineare con durata finita

Definizione dei livelli logici 1 e 0



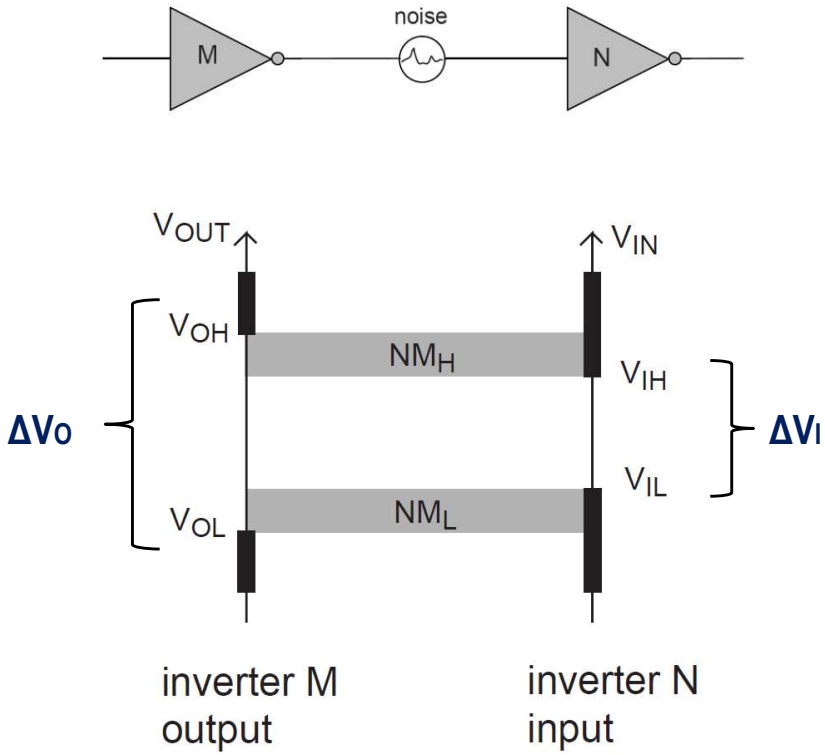
Intervallo di valori di ingresso che produce un valido 1 logico

Intervallo di valori di ingresso che produce un valido 0 logico

- 0 logico: intervallo di valori di uscita compreso fra V_{MIN} e V_{OL}
  - V_{MIN} = tensione di uscita per la quale VIN=V+
  - V_{OL} = la più piccola tensione di uscita per la quale la pendenza è -1
- 1 logico: intervallo di valori di uscita compreso fra V_{OH} e V_{MAX}
  - V_{OH} = la più grande tensione di uscita per la quale la pendenza =-1
  - V_{MAX} = tensione di uscita per la quale VIN=0

Margine di rumore

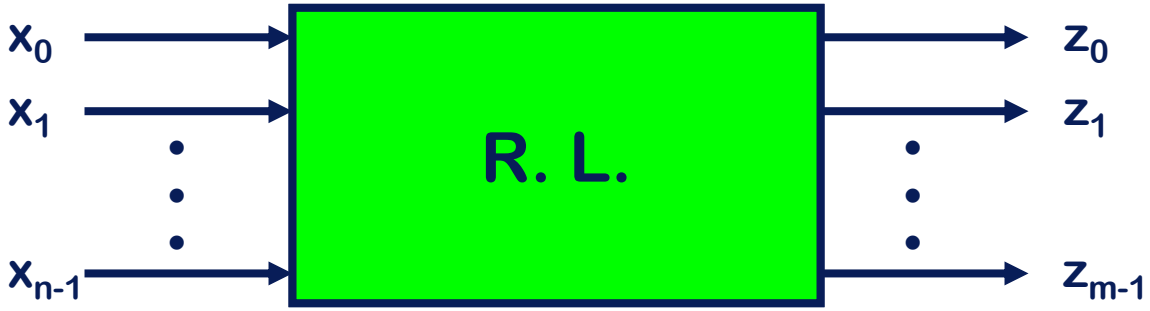
- Se (VOH-VOL) > (VIH-VIL) l'inverter ha una certa immunità al rumore



- NMH = (VOH-VIH) margine di rumore sul livello alto
- NML = (VIL-VOL) margine di rumore sul livello basso

Lezione 2: Dalle Porte Elementari alle Reti Logiche

- **Una Rete Logica è tipicamente un sistema elettronico:**
  - **Unidirezionale, cioè in cui i segnali viaggiano dagli ingressi verso le uscite e non viceversa (ovvero il sistema non è reversibile)**
  - **I cui segnali di ingresso e di uscita sono digitali, cioè ogni filo assume valori zero, uno, indeterminato o alta-impedenza**
  - **Descrivibile con equazioni dell'algebra booleana, se si assume una situazione ideale (in cui i valori dei segnali sono solo zero o uno)**



Reti Combinatorie e Reti Sequenziali

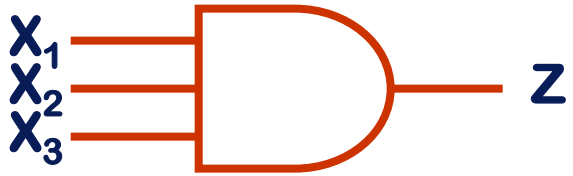
- Reti COMBINATORIE (RC o CN==Combinatorial Network)
  - In qualunque istante
    le uscite sono funzione del valore che gli ingressi hanno in quell'istante
  - Il comportamento (uscite in funzione degli ingressi)
    è descritto da una tabella
  - NON CONTENGONO ANELLI DI RICHIUSURA

- Reti SEQUENZIALI
  - In un determinato istante
    le uscite sono funzione del valore che gli ingressi hanno in quell'istante
    e dei valori che gli ingressi hanno assunto precedentemente
  - Presentano Stati Interni (sono quindi reti dotate di *MEMORIA*)
    → La descrizione è più complessa di una semplice tabella
  - PRESENTANO ANELLI DI RICHIUSURA

Porte Logiche Base

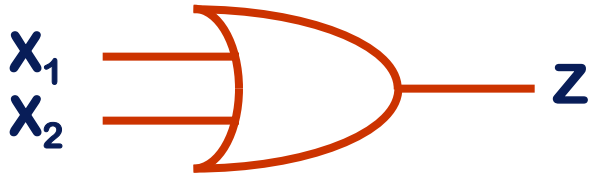
- Rappresentazione circuitale delle funzioni logiche
  - AND o prodotto

$$Z = X_1 \cdot X_2 \cdot X_3$$



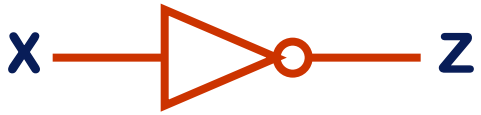
  - OR o somma

$$Z = X_1 + X_2$$



  - NOT o negazione

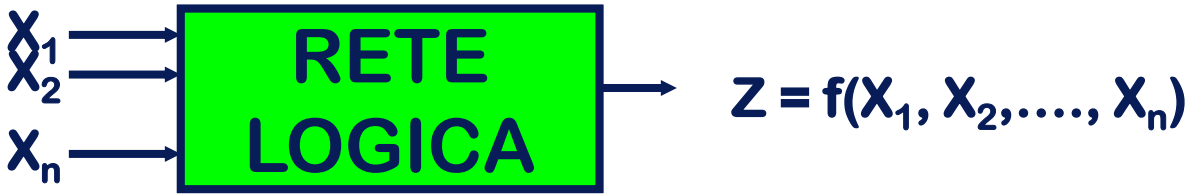
$$Z = \overline{X}$$



- Con le sole porte AND, OR, NOT si può realizzare una Rete Logica qualsiasi
  - Verrà dimostrato nel seguito riducendo una rete logica arbitraria a forme «standard» (cf. "somma di prodotti" o "prodotto di somme")

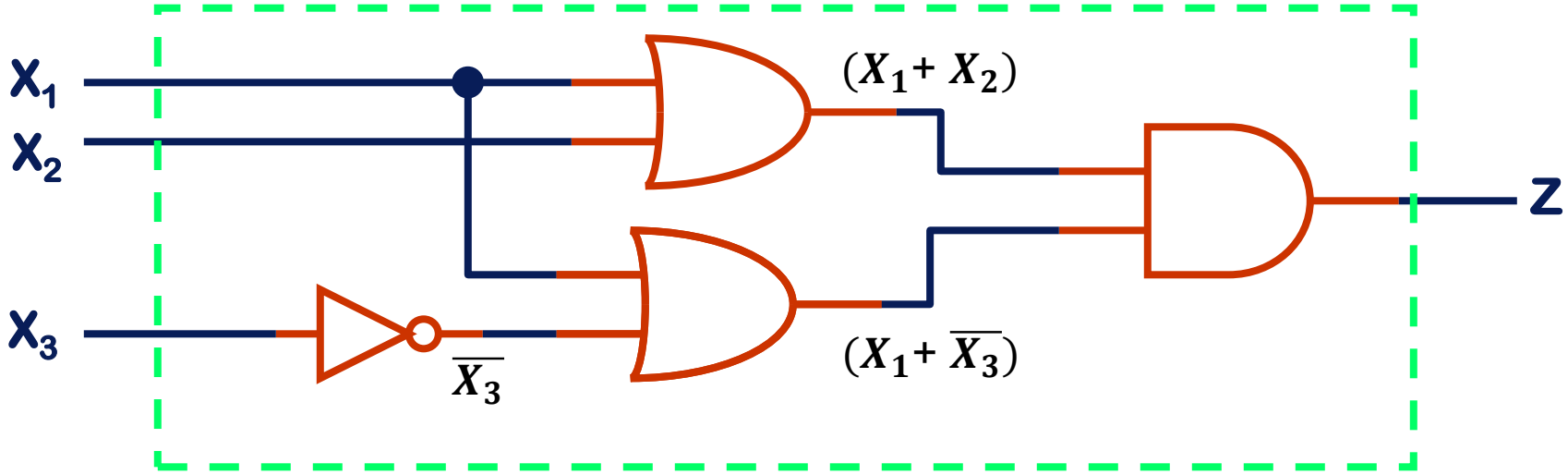
Esempio di rete logica

- Data una RETE LOGICA, possiamo descrivere la relazione fra ingresso X e uscita Z tramite una funzione 'f'



- Ad esempio, potremmo avere:

$$Z = f(X_1, X_2, \ldots, X_n) = (X_1 + X_2) \cdot (X_1 + \overline{X_3})$$

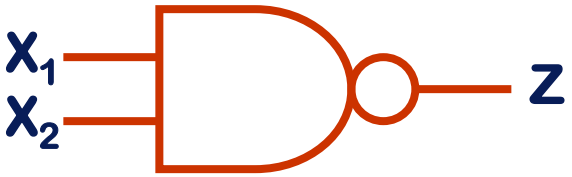


Porta NAND e NOR

- NAND

X_1	X_2	Z
0	0	1
0	1	1
1	0	1
1	1	0

$$Z = \overline{X_1 \cdot X_2}$$



- NOR

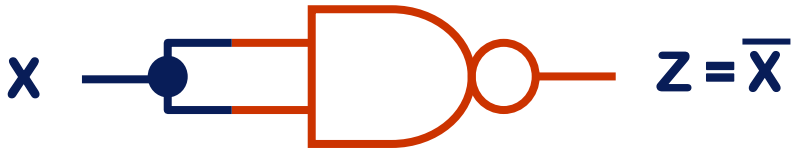
X_1	X_2	Z
0	0	1
0	1	0
1	0	0
1	1	0

$$Z = \overline{X_1 + X_2}$$

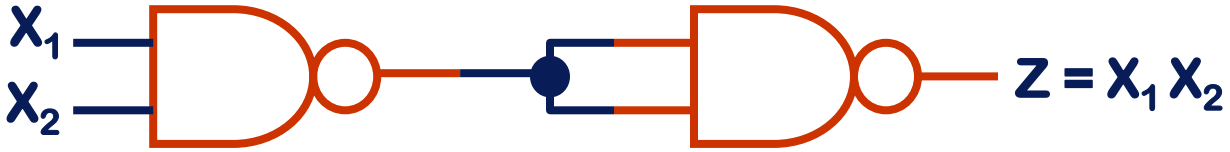
Proprietà della porta NAND (NOR)

- Utilizzando solamente porte solo NAND (o solo NOR) è possibile realizzare qualunque rete logica; infatti

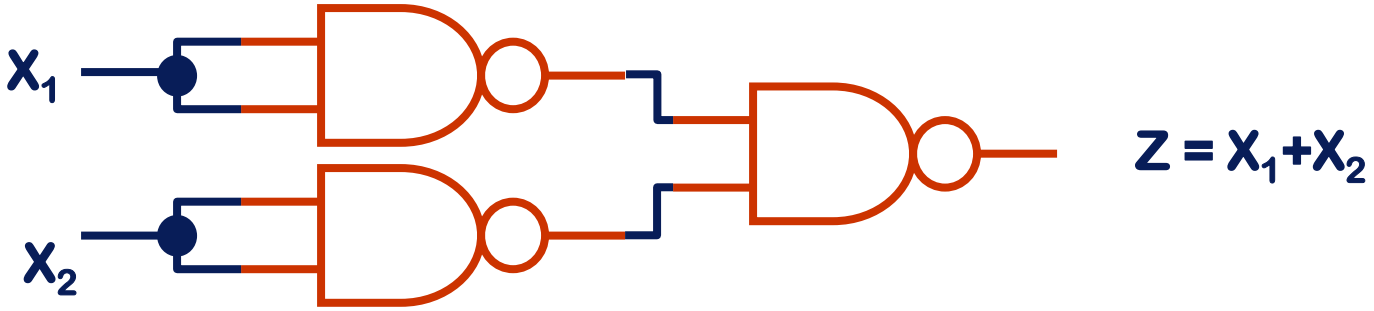
- NOT



- AND



- OR

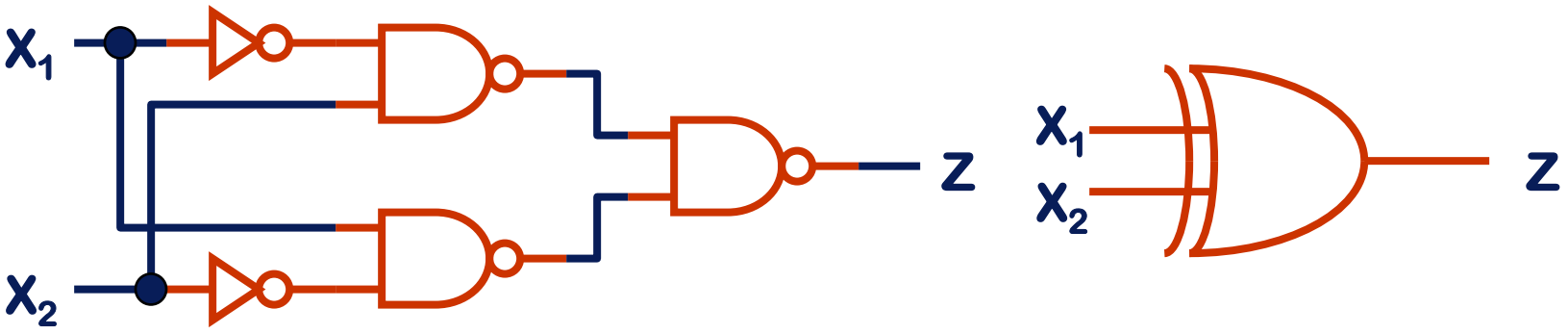


OR Esclusivo (XOR)

- **Realizzazione dell'OR Esclusivo**

X_1	X_2	Z
0	0	0
0	1	1
1	0	1
1	1	0

$$Z = X_1 \oplus X_2 = \overline{X_1}X_2 + X_1\overline{X_2} = \overline{(X_1 + \overline{X_2})(\overline{X_1} + X_2)} = \overline{(\overline{\overline{X_1}}X_2)(X_1\overline{\overline{X_2}})}$$



Algebra della Logica

- George Boole
  - Matematico inglese (1815 – 1864)
- Algebra della Logica, Algebra di Boole, Algebra Booleana
  - Sistema matematico formale che descrive funzioni logiche
- Funzioni e variabili che possono assumere solo due valori
  - Falso — Vero
  - 0 (logico) — 1 (logico)
  - Livello logico Basso — Livello logico Alto
  - 0 V — 5 V
- Sistema matematico formale
  - insieme di elementi
  - insieme di operazioni
  - insieme di postulati
  - TEOREMI
- Perché è importante:
  - Nel 1938, Shannon dimostrò che l'Algebra booleana può descrivere le operazioni di circuiti elettrici che commutano fra due valori

Tabella di Verità

- Una funzione logica può sempre essere espressa da una tabella detta:

  TABELLA DI VERITÀ (TRUTH TABLE)

- Osservazioni
  - Una funzione logica di "n" variabili ammette 2^n possibili valori
  - Tale funzione è completamente descritta da una tabella che ha sulla sinistra le 2^n possibili configurazioni degli ingressi e a destra i valori (0 o 1) a secondo del valore della funzione. Tale tabella è detta tabella di verità

- Esempio (funzione di tre variabili) $z = f(x_1, x_2, x_3)$

x_1	x_2	x_3	z
0	0	0	$f(0,0,0)$
0	0	1	$f(0,0,1)$
0	1	0	$f(0,1,0)$
0	1	1	$f(0,1,1)$
1	0	0	$f(1,0,0)$
1	0	1	$f(1,0,1)$
1	1	0	$f(1,1,0)$
1	1	1	$f(1,1,1)$

Postulati di HUNTINGTON

Postulato	Espressione primale (rispetto alla somma)	Espressione duale (rispetto al prodotto)
Esistenza di almeno due elementi distinti	$\exists\, x, y \in B : x \neq y$	
Esistenza degli elementi identità	$x + 0 = x$	$x \bullet 1 = x$
Proprietà commutativa	$x + y = y + x$	$x \bullet y = y \bullet x$
Proprietà distributiva	$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$	$x + (y \bullet z) = (x + y) \bullet (x + z)$
Esistenza del complemento	$x + \overline{x} = 1$	$x \bullet \overline{x} = 0$
Chiusura	$\forall x, y \in B, (x + y) \in B$	$\forall x, y \in B, (x \bullet y) \in B$

Osservazioni

- Alcune proprietà dell'algebra booleana sono vere anche nell'algebra normalmente usata:
  - Proprietà commutativa
  - Proprietà distributiva del prodotto logico
- Altre proprietà non sono vere nell'algebra tradizionale:
  - Proprietà distributiva della somma logica
- L'operazione complemento logico esiste solo nell'algebra booleana
- La sottrazione e la divisione non esistono nell'algebra booleana

- Vale un PRINCIPIO DI DUALITÀ
  - I postulati duali si ottengono da quelli primali operando le seguenti sostituzioni:

Scambiando i due operatori binari fra loro, (+) con (·) e (·) con (+)

E

Scambiando fra loro i due elementi identità, 1 con 0 e 0 con 1

TEOREMI FONDAMENTALI

- **Tecniche di dimostrazione dei teoremi**
  - Impiego dei postulati fondamentali
  - Uso di teoremi precedentemente dimostrati
  - Dimostrazione per assurdo
    - (si ipotizza verificata l'ipotesi opposta a quella desiderata e si conclude che non è possibile che sia vera)
  - Dimostrazione per induzione
    - (se una ipotesi è vera per n=0 (nel caso booleano sempre) e assumendo vera l'ipotesi con k variabili risulta vera anche per k+1 variabili allora è vera per qualunque n)

- **Dimostrazione per induzione perfetta**
  - La tabella di verità consente di provare la veridicità di una relazione logica, poiché verifica se la relazione è vera per TUTTE le possibili combinazioni dei valori delle variabili
  - Tale metodo prende il nome di *Metodo dell'INDUZIONE PERFETTA*

TEOREMI

TEOREMA	Espressione primale (rispetto alla somma)	Espressione duale (rispetto al prodotto)
Annullamento	$x + 1 = 1$	$x \bullet 0 = 0$
Involuzione	$\overline{(\overline{x})} = x$	
Idempotenza	$x + x = x$	$x \bullet x = x$
Assorbimento	$x + xy = x$	$x(x + y) = x$
Somma con il complemento	$x + \overline{x}y = x + y$	$x(\overline{x} + y) = xy$
Associatività	$x + (y + z) = (x + y) + z$ $= x + y + z$	$x(yz) = (xy)z$ $= xyz$
Teorema del consenso	$xy + \overline{x}z + yz$ $= xy + \overline{x}z$	$(x + y)(\overline{x} + z)(y + z)$ $= (x + y)(\overline{x} + z)$
Teorema di De Morgan	$\overline{(x + y)} = \overline{x} \cdot \overline{y}$	$\overline{(x \cdot y)} = \overline{x} + \overline{y}$

Esempio di dimostrazione

- **Teorema di De Morgan**

$$\overline{(x + y)} = \overline{x} \cdot \overline{y}$$

x	y	$\overline{x}$	$\overline{y}$	$x+y$	$\overline{(x+y)}$	$\overline{x} \cdot \overline{y}$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

c.v.d.

Teorema del Consenso

$$xy + /xz + yz = xy + /xz$$

x	y	z	/x	xy	/xz	yz	xy + /x z + yz	xy+/xz
0	0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	1	1
0	1	0	1	0	0	0	0	0
0	1	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
1	1	0	0	1	0	0	1	1
1	1	1	0	1	0	1	1	1

c.v.d.

Dalle tabelle di verità alla sintesi della rete logica

- Una funzione booleana può sempre essere definita tramite tabella di verità
  - Esempio:

a	b	c	z
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

- È possibile derivare una espressione algebrica a partire dalla tabella di verità in (almeno) due modi:
  - Identificando tutte le righe che producono un 1 e sommando (OR) i corrispondenti termini (prodotti ovvero AND di variabili) che producono tale 1. Tali «termini prodotto» prendono il nome di mintermini. L'espressione risultante è detta somma di prodotti. Esempio:

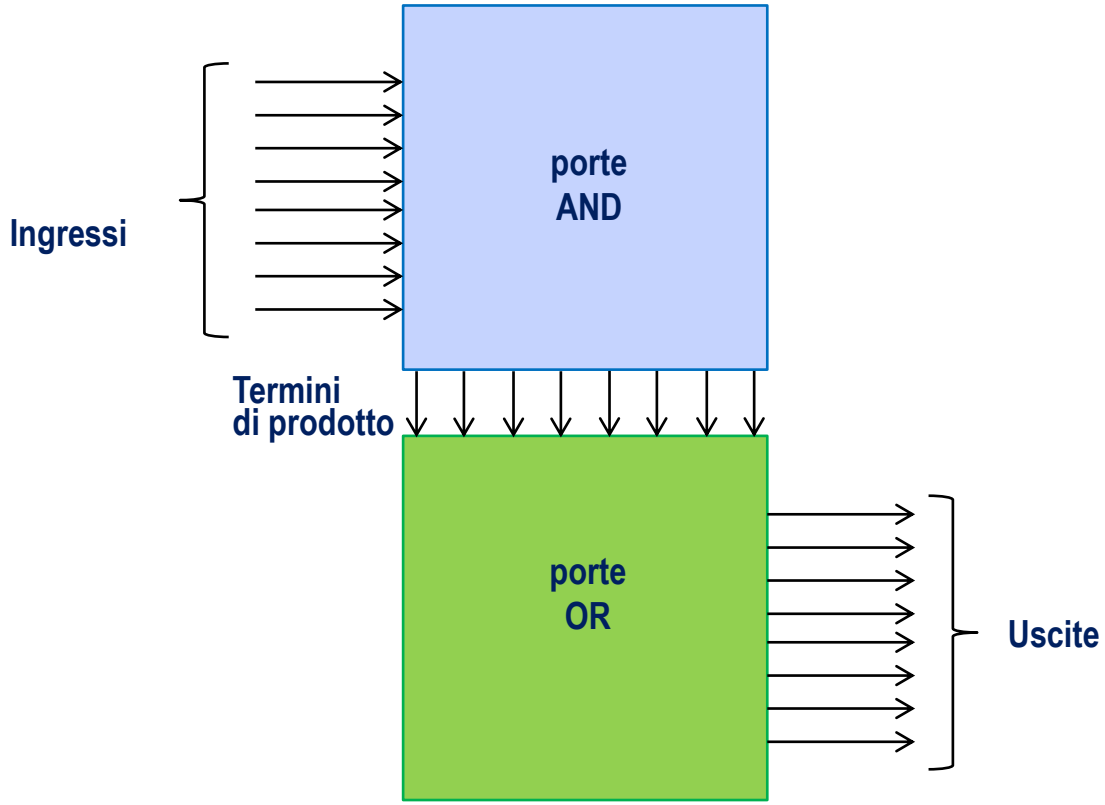
$$z = \bar{a} \cdot \bar{b} \cdot \bar{c} + \bar{a} \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot c + a \cdot b \cdot \bar{c}$$

  - Identificando tutte le righe che producono uno 0 e facendo il prodotto (AND) dei corrispondenti termini (somme ovvero OR di variabili) che producono tale 0. Tali «termini somma» prendono il nome di maxtermini. L'espressione risultante è detta prodotto di somme. Esempio:

$$z = \left(a + b + \bar{c}\right) \cdot \left(a + \bar{b} + \bar{c}\right) \cdot \left(\bar{a} + \bar{b} + \bar{c}\right)$$

Programmable Logic Array - PLA

- Dato che si può sempre rappresentare una funzione logica come tabella di verità, si può sempre rappresentare anche come somma di prodotti (o prodotto di somme)
- Questo conduce ad una tecnica di implementazione universale nota come PLA

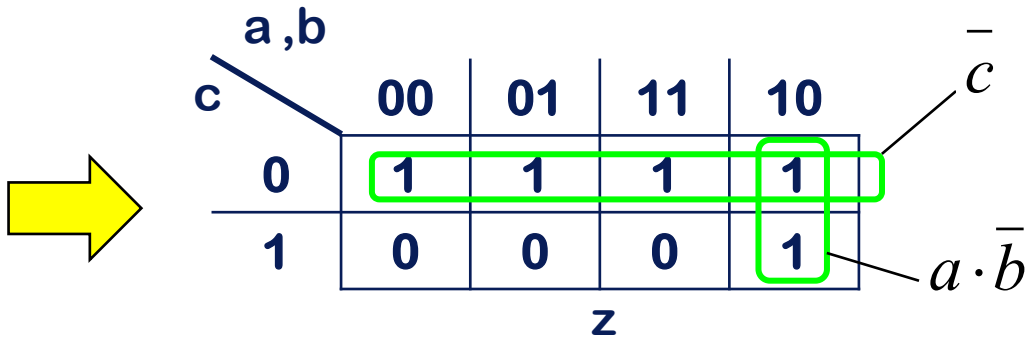


Le due matrici AND e OR possono essere anche rese "programmabili" (vedere lezioni successive)

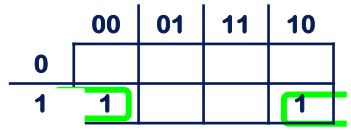
Mappa di Karnaugh

- Può essere comodo semplificare l'espressione booleana
  - Per ottenere espressioni booleane più semplici e facili da verificare
  - Per sintetizzare una rete logica con un minor numero di porte (minor area)
- La Mappa di Karnaugh è una rappresentazione alternativa della tabella di verità che, sfruttando la proprietà di assorbimento*, permette rapidamente di identificare termini più semplici
- La Mappa di Karnaugh si ottiene:
  - disponendo gli "indici" generati dalle variabili di ingresso in modo che due indici adiacenti differiscano per una sola cifra. Esempio:

a	b	c	z
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



Nota: la mappa si svolge in realtà su una superfice toroidale → I sottocubi possono continuare su celle adiacienti la parte opposta



- Localizzando i gruppi adiacenti di 1 con un numero di 1 pari a una potenza di 2 (detti sottocubi di ordine k se contengono 2^k uni)
- Un termine è identificato dagli indici che rimangono "fissi" nel sottocubo. Esempio:

$$z = \bar{c} + a \cdot \bar{b}$$

* $xy+\bar{x}y=y$

Mappe di Karnaugh a 2, 3, 4, 5 variabili

	0	1
0	1	1
1		

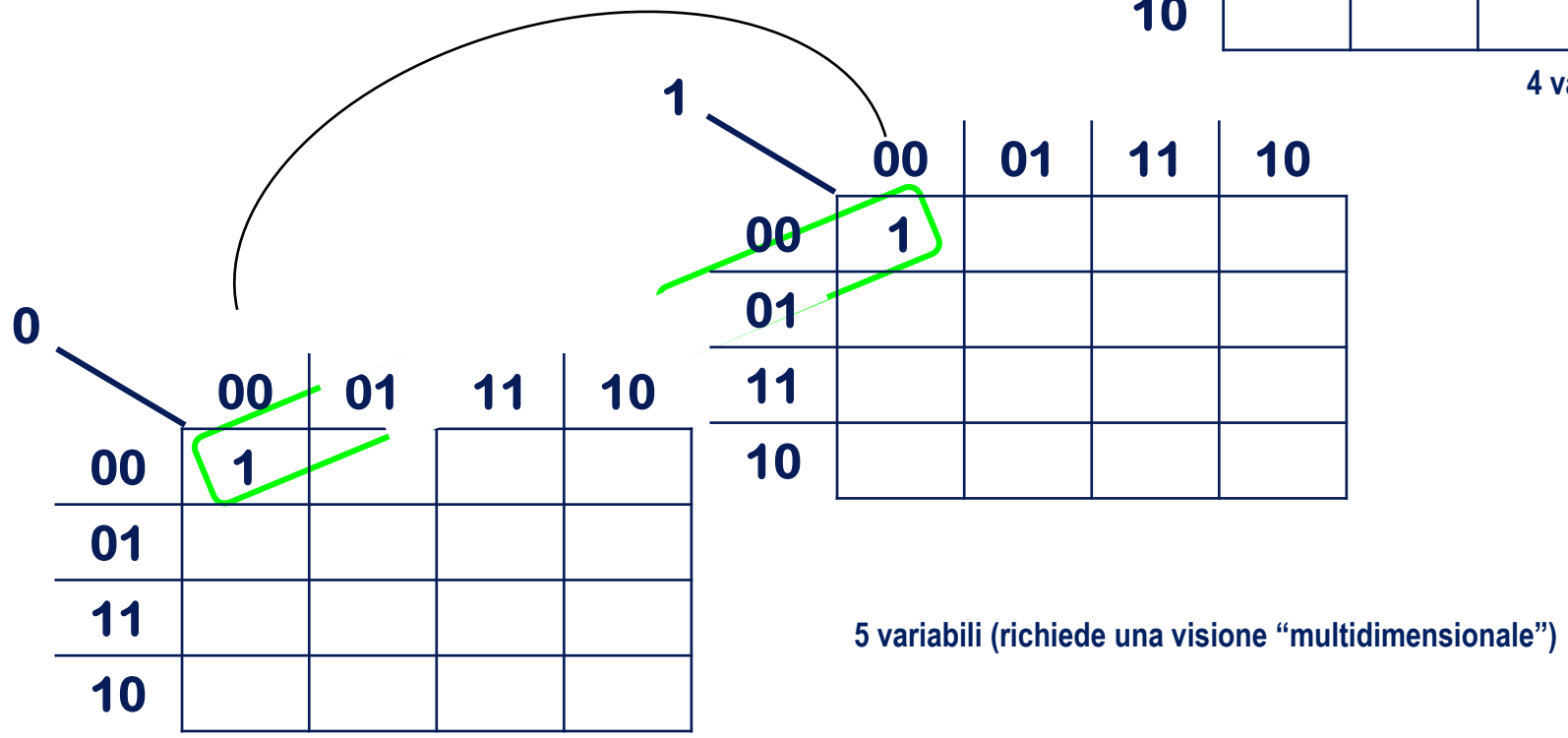
2 variabili

	00	01	11	10
0				
1		1	1	

3 variabili

	00	01	11	10
00				
01		1	1	
11		1	1	
10				

4 variabili



5 variabili (richiede una visione “multidimensionale”)

Note sulle Mappe di Karnaugh

- Se la mappa contiene dei non-specificato (X o -)
  - Si può interpretare come è più conveniente per ottenere sottocubi più grandi
- Se ci sono più possibilità con sottocubi più piccoli e più grandi
  - Si preferiscono sempre coperture con sottocubi più grandi (hanno meno ingressi)
- Si prendono solo i termini strettamente necessari per ottenere una lista di copertura "a costo minimo"
  - Esiste molta letteratura in merito alla ottimizzazione delle liste a costo minimo (ma non fa parte di questo corso)
- Esercizio:
  - Scaricare la app per Android: "Karnaugh Kmap Solver" (simili esistono anche per iPhone)
  - Provare a verificare la sintesi con alcune funzioni logiche usando tale app

Karnaugh, M. "A Map method for Synthesis of Combinational Logic", Transactions of AIEE, Communication and Electronics, 72, part I, Nov. 1953, pp. 593-99

Lezione 3
Elementi Architetturali Principali: Reti Combinatorie Notevoli

(v. PHRV1, appendice A.2,A.3,A.5,A.12)

Decoder

Encoder

Encoder con priorità

Multiplexer

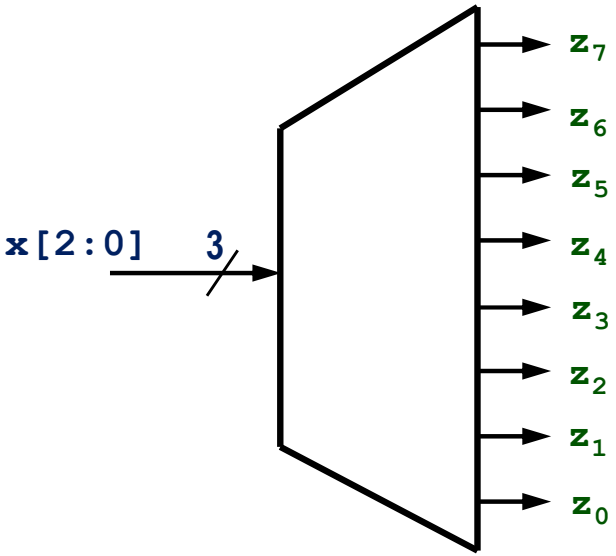
Demultiplexer

Look-Up-Table (LUT)

Arithmetic-Logic Unit (ALU)

(v. PHRV1, appendice A.3)

Decoder

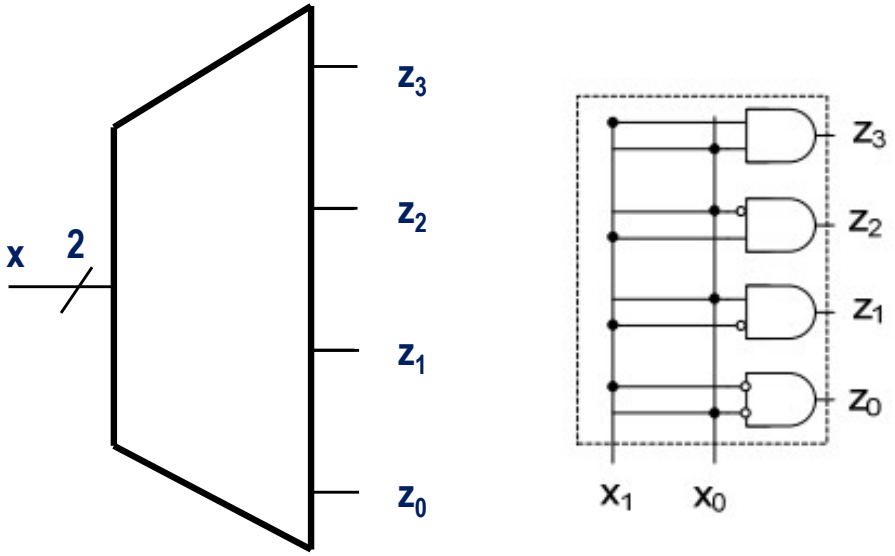


INGRESSI			USCITE							
x_2	x_1	x_0	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

- Esempio di decoder a 3 bit o "da-3-a-8"
- Un solo filo di uscita vale 1 (gli altri valgono tutti 0): quello di indice pari all'intero "posto sui fili" di ingresso
- In generale, un decoder da-n-a-2n ha n ingressi e 2n uscite

Realizzazione del decoder

- Si può sintetizzare in termini di porte logiche con 2n porte AND ognuna delle quali riconosce uno dei 2n mintermini



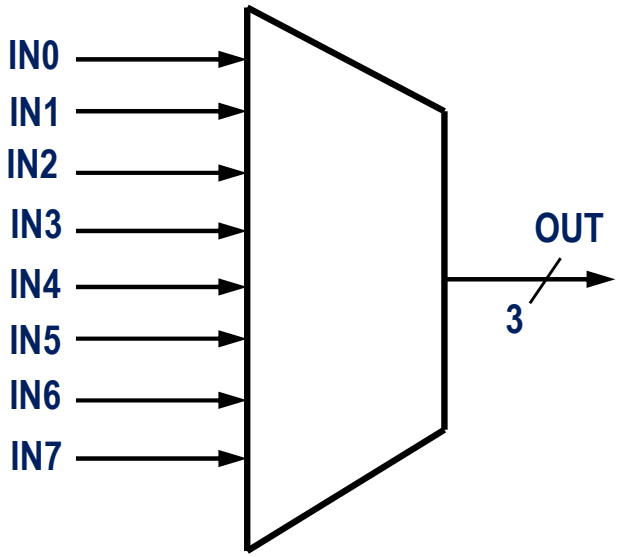
Es. decoder da-2-a-4

Nota: in realtà possono esistere implementazioni fisiche CMOS diverse è più efficienti dell'elemento architetturale "decoder"

Nota2: può anche avere un segnale di abilitazione; si realizza mettendo ogni uscita in AND con tale segnale di abilitazione

Encoder

- L'encoder effettua l'operazione inversa del decoder
- Posto un 1 sul k-esimo dei 2n ingressi (gli altri sono posti a 0): l'uscita presenta i bit corrispondenti alla rappresentazione di k in base due



IN7	IN6	IN5	IN4	IN3	IN2	IN1	IN0	OUT2	OUT1	OUT0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Nota: questa NON è una tabella della verità; la si può interpretare come tabella di verità osservando che nelle mancanti 244 righe le uscite sono indeterminate (X o “don’t care”)

Realizzazione dell'encoder

- OUT0=IN1+IN3+IN5+IN7
- OUT1=IN2+IN3+IN6+IN7
- OUT2=IN4+IN5+IN6+IN7

- Dunque tale encoder può essere realizzato con 3 porte OR a 4 ingressi

Esercizio per casa: è possibile ricavare queste equazioni dalle mappe di Karnaugh ?

Esercizio: Encoder da 4 a 2

- L'encoder effettua l'operazione inversa del decoder
- Posto un 1 sul k-esimo dei 2^n ingressi (gli altri sono posti a 0): le n uscite producono i bit corrispondenti alla rappresentazione in base due di k

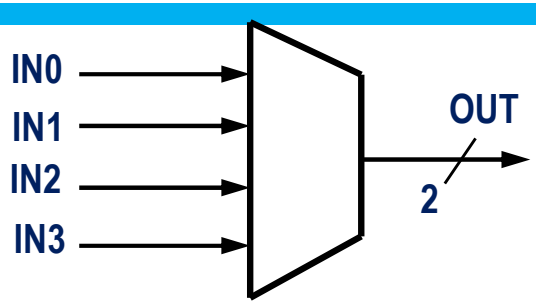


TABELLA DI VERITÀ (COMPLETA):

IN3	IN2	IN1	IN0		OUT1	OUT0
0	0	0	1		0	0
0	0	1	0		0	1
0	1	0	0		1	0
1	0	0	0		1	1
0	0	0	0		X	X
0	0	1	1		X	X
0	1	0	1		X	X
0	1	1	0		X	X
0	1	1	1		X	X
1	0	0	1		X	X
1	0	1	0		X	X
1	0	1	1		X	X
1	1	0	0		X	X
1	1	0	1		X	X
1	1	1	0		X	X
1	1	1	1		X	X

IN3,IN2 \ IN1,IN0	00	01	11	10
00	X	0	X	1
01	0	X	X	X
11	X	X	X	X
10	1	X	X	X

OUT0

IN3,IN2 \ IN1,IN0	00	01	11	10
00	X	0	X	0
01	1	X	X	X
11	X	X	X	X
10	1	X	X	X

OUT1

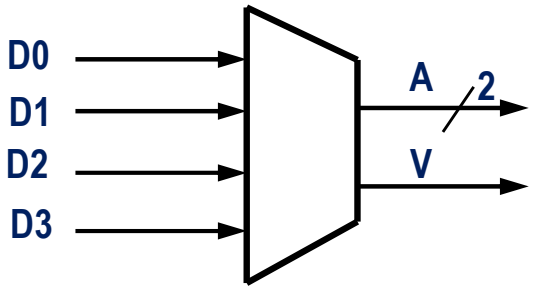
OUT0=IN1+IN3 OUT1=IN2+IN3

Priority Encoder

- Questa è una rete diversa dall'encoder semplice

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

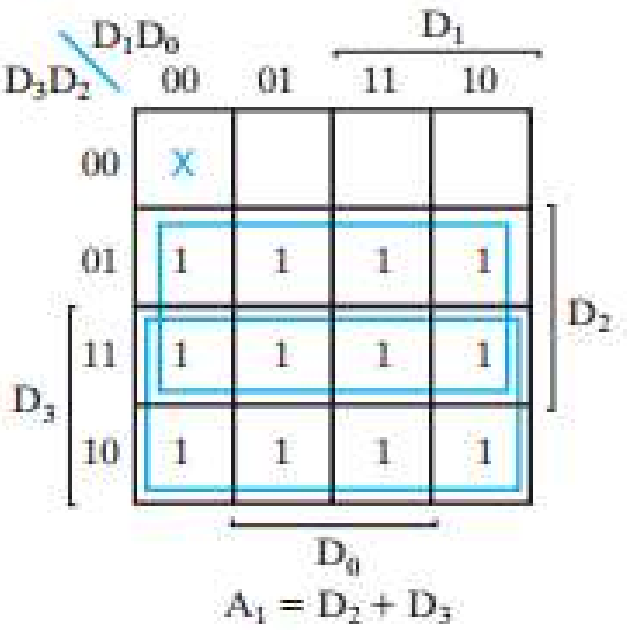
Tabella di verità completa



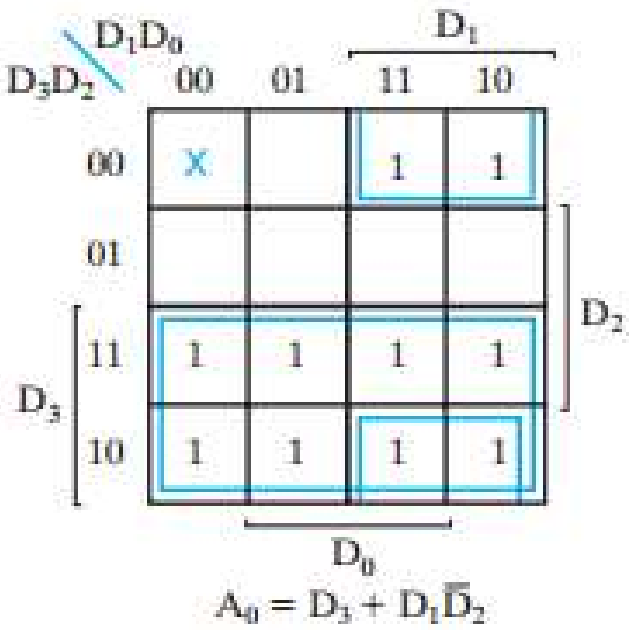
- Quando D3 vale 1 tutti gli altri bit non contano e l'uscita indica la massima priorità $(11)_2$ ovvero 3
- Quando D3 vale 0 ma D2 vale 1 gli altri bit a destra non contano e l'uscita indica priorità 2 e così via…
- Il bit V è necessario per indicare se l'uscita è valida ovvero se almeno un 1 è stato posto in ingresso

Nota: stavolta gli 'X' (non-specificato) sugli INPUTS assumono il significato di "sia nel caso 0 che nel caso 1".

Realizzazione del Priority Encoder



$A_1 = D_2 + D_3$

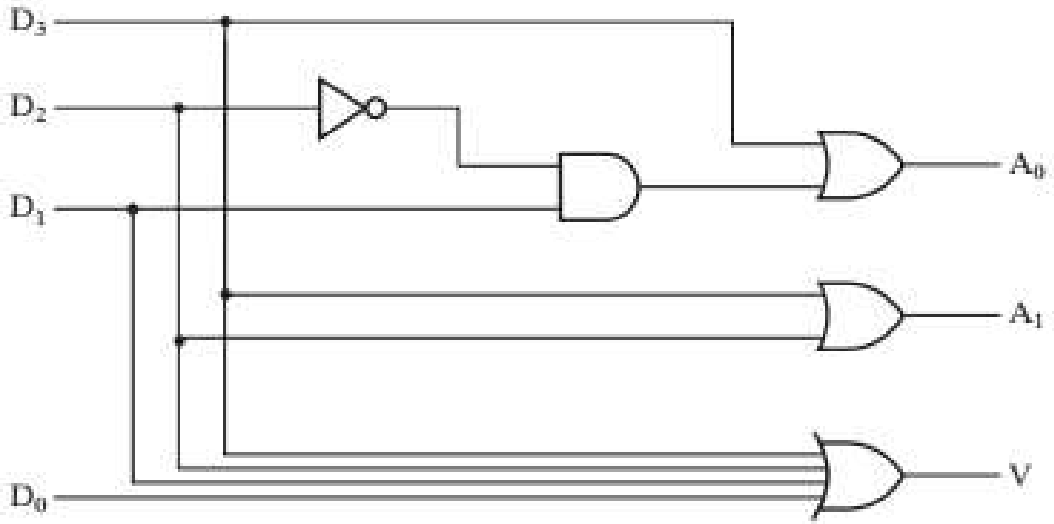


$A_0 = D_3 + D_1\overline{D}_2$

$A_0 = D_3 + D_1\overline{D}_2$

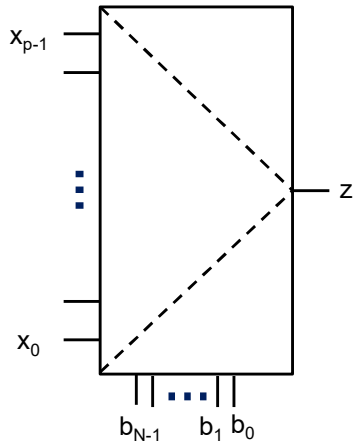
$A_1 = D_2 + D_3$

$V = D_0 + D_1 + D_2 + D_3$



Multiplexer

- Il multiplexer serve per selezionare un'informazione



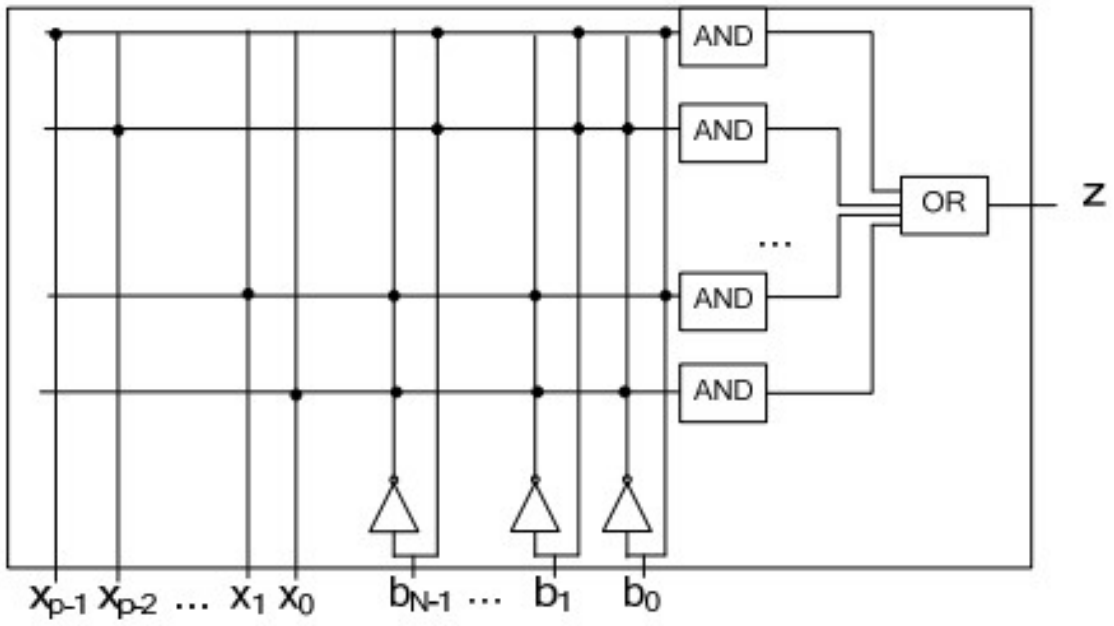
Multiplexer con 2^N ingressi

- Ha 2^N+N ingressi e 1 uscita
- Gli N ingressi b sono detti variabili di comando
- Specificando B=i viene selezionato uno (quello con indice i) dei $2^N=p$ ingressi x

$$z = x_i \Leftrightarrow (b_{N-1} \ldots b_1 b_0)_2 \equiv i$$

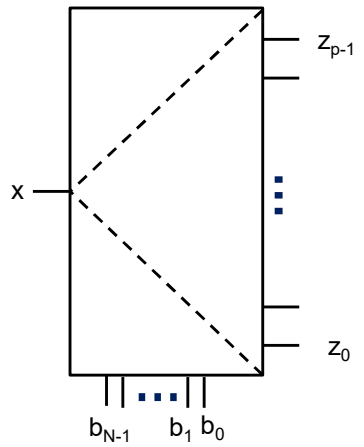
$$z = x_0 \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot \ldots \cdot \overline{b_1} \cdot \overline{b_0} +$$
$$x_1 \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot \ldots \cdot \overline{b_1} \cdot b_0 +$$
$$\ldots$$
$$x_{p-2} \cdot b_{N-1} \cdot b_{N-2} \cdot \ldots \cdot b_1 \cdot \overline{b_0} +$$
$$x_{p-1} \cdot b_{N-1} \cdot b_{N-2} \cdot \ldots \cdot b_1 \cdot b_0$$

AND con N+1 ingressi



Demultiplexer

- Realizza la funzione inversa del multiplexer, ovvero data un'informazione x, la distribuisce su una di 2^N uscite possibili



Demultiplexer con 2^N uscite

- Ha 1+N ingressi e 2^N uscite
- Gli N ingressi b sono detti variabili di comando
- Specificando B=i viene inviato l'ingresso x su una (quella con indice i) delle 2^N=p uscite z
- Le altre uscite valgono 0

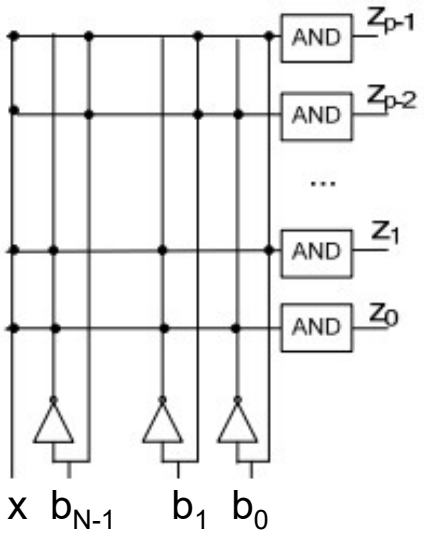
$$z_0 = x \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot \ldots \cdot \overline{b_1} \cdot \overline{b_0}$$

$$z_1 = x \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot \ldots \cdot \overline{b_1} \cdot b_0$$

...

$$z_{p-2} = x \cdot b_{N-1} \cdot b_{N-2} \cdot \ldots \cdot b_1 \cdot \overline{b_0}$$

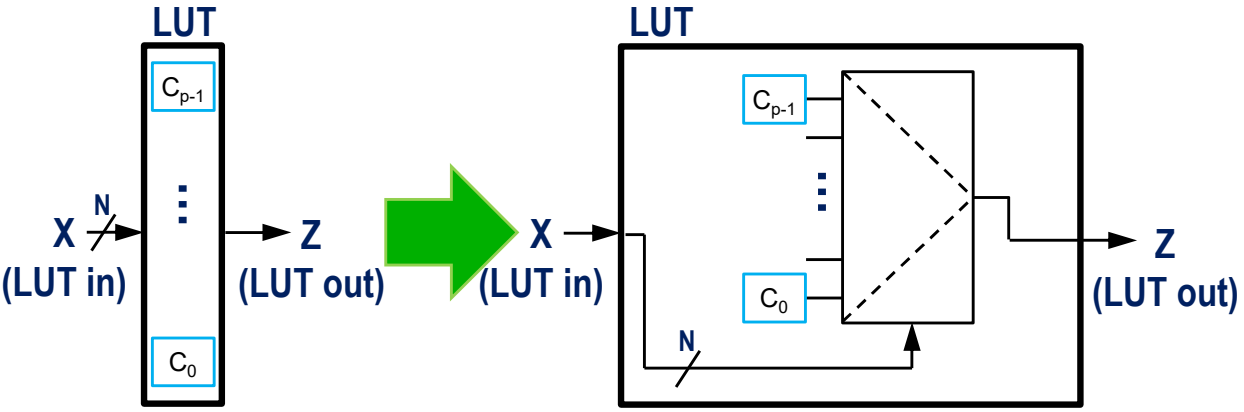
$$z_{p-1} = x \cdot b_{N-1} \cdot b_{N-2} \cdot \ldots \cdot b_1 \cdot b_0$$



Può essere visto come un decoder con abilitazione

Look Up Table (LUT)

- è un Multiplexer in cui gli ingressi sono costanti



- Il multiplexer ha (con $p=2^N$) 2^N+N ingressi e 1 uscita
- Gli N ingressi X fungono da variabili di comando
- Specificando X=i viene selezionato uno (quello con indice i) dei $2^N=p$ ingressi (FISSI ovvero COSTANTI)

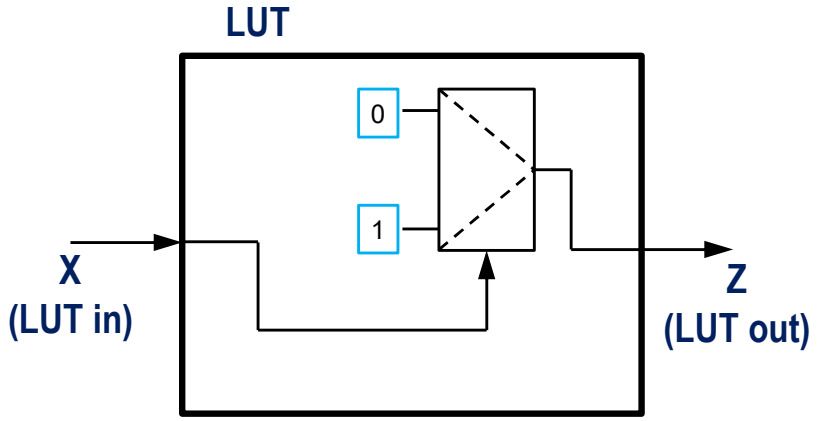
- è utile per realizzare una funzione booleana generica
- Esempio realizzazione della funzione "inverter" tramite LUT

X	Z
0	1
1	0

Profondità (depth)

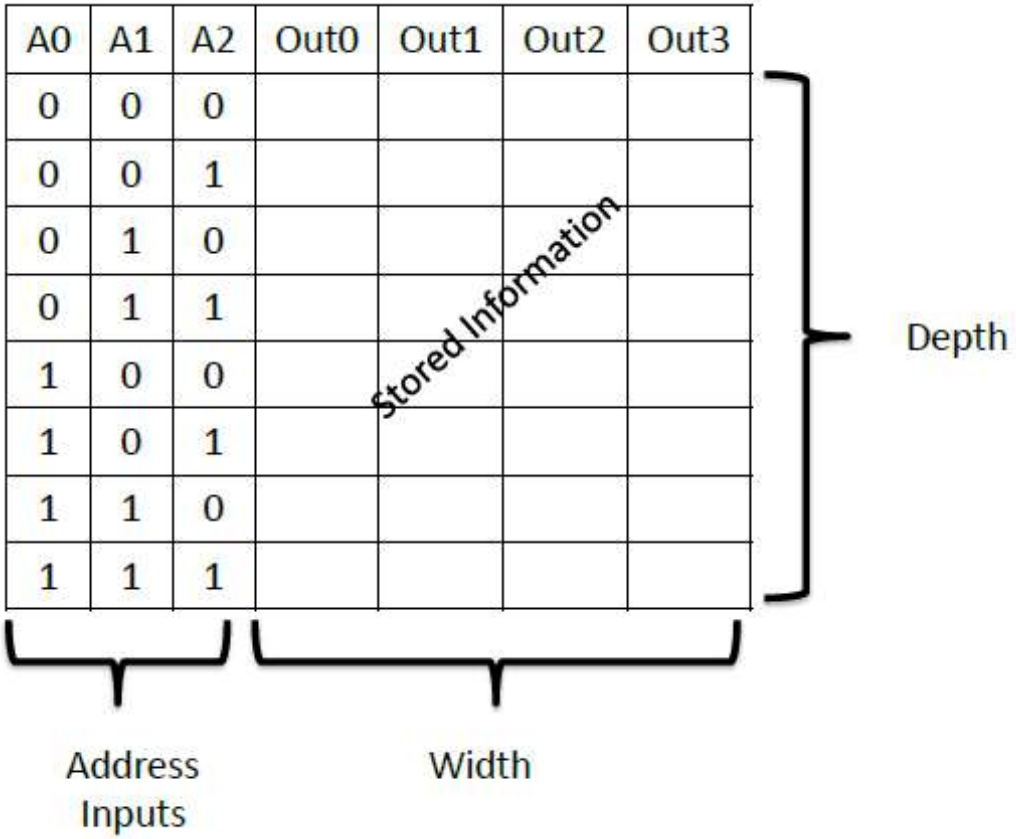
Ingressi

Larghezza (width)



LUT larga 4 e profonda 8

- I segnali di controllo del MUX sono usati per selezionare le possibili uscite della funzione booleana

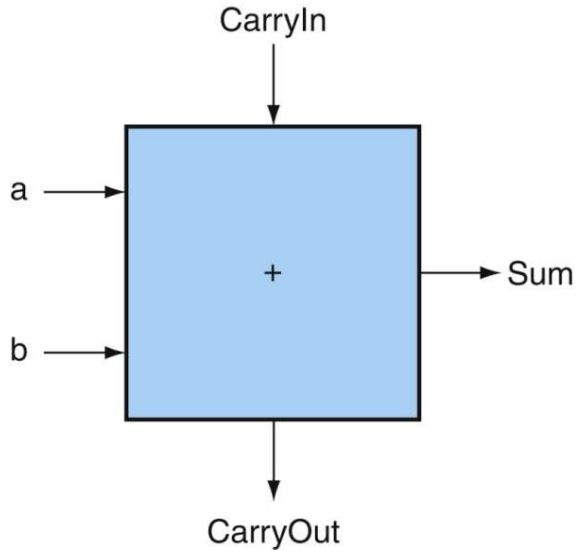


- Vedremo successivamente come rendere tali bit di ingresso "programmabili" (usando» celle di memoria»)
- Le FPGA forniscono un determinato numero di LUTs per realizzare funzioni più complesse (anche soft-processors !)

COSTRUZIONE DI UNA ARITHMETIC LOGIC UNIT (ALU)

(V. PHRV1, APPENDICE A.5)

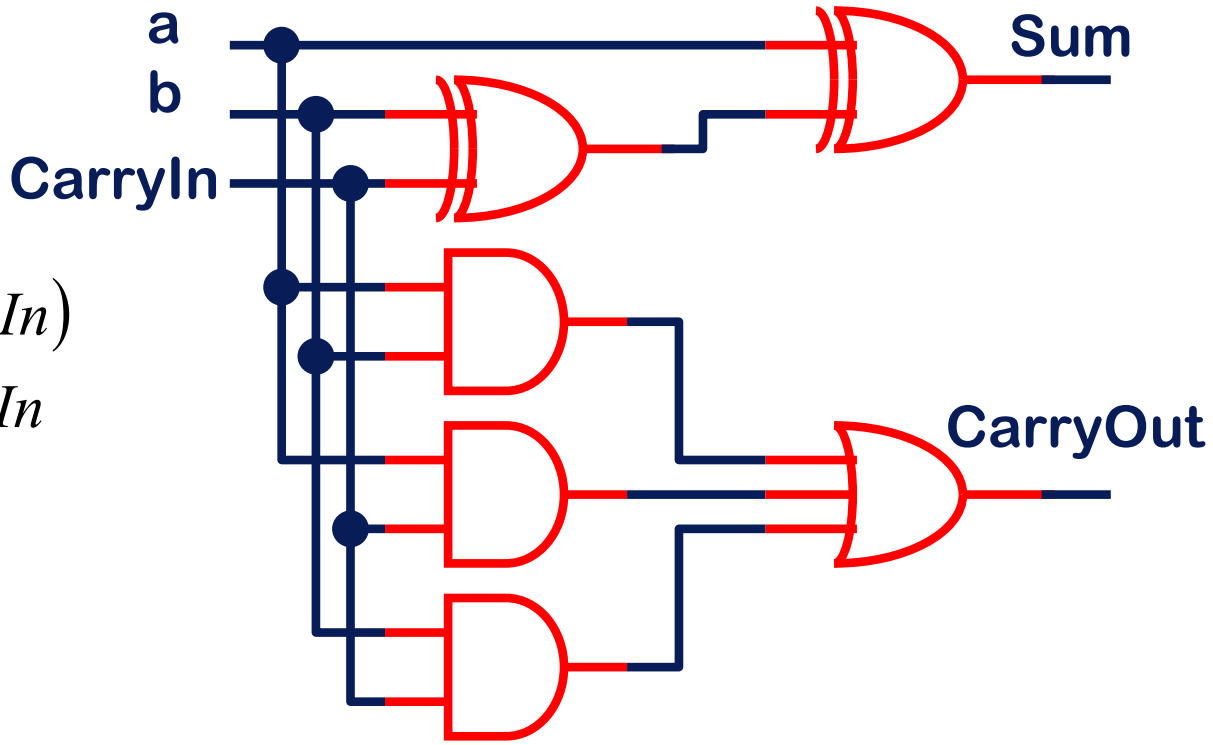
Full Adder a 1 bit



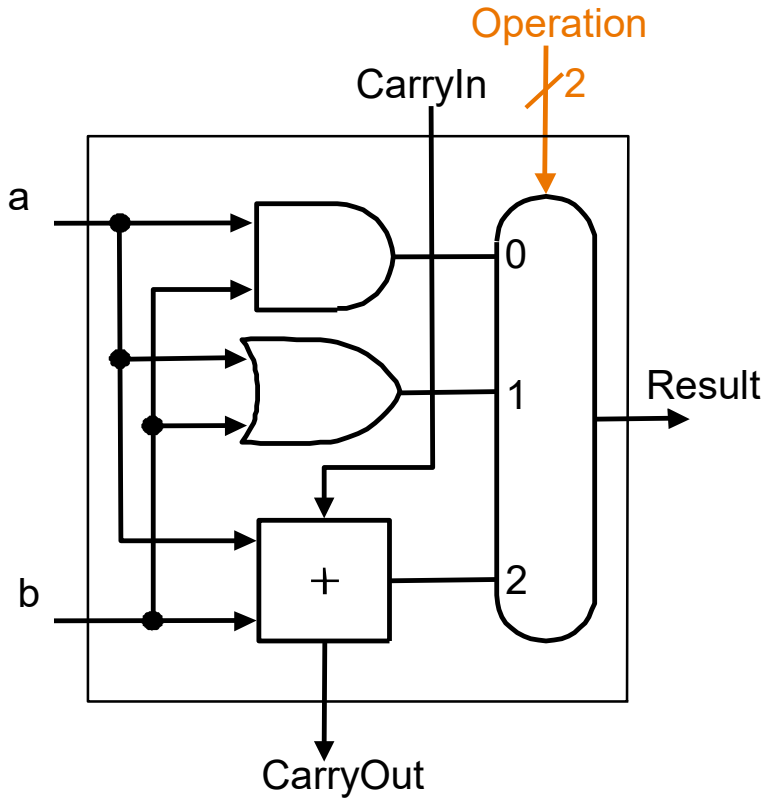
Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

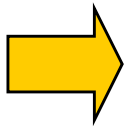
$$sum = a \oplus b \oplus CarryIn = a \oplus (b \oplus CarryIn)$$

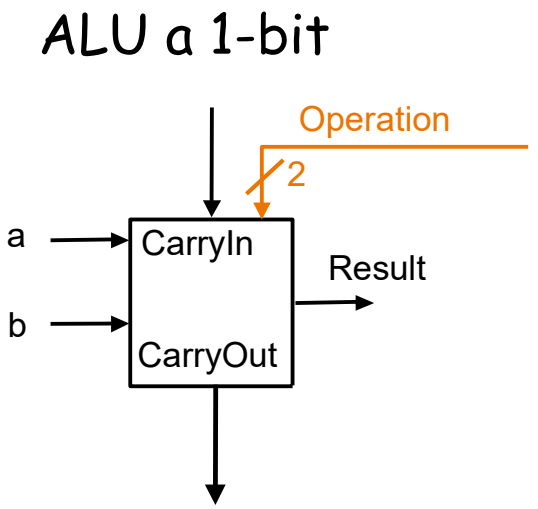
$$CarryOut = a \cdot b + a \cdot CarryIn + b \cdot CarryIn$$



Semplice ALU a 1-bit

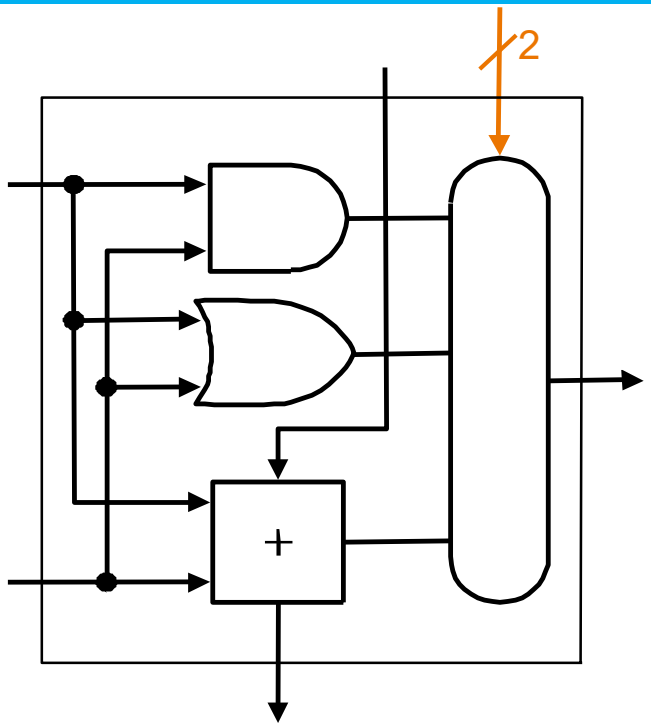




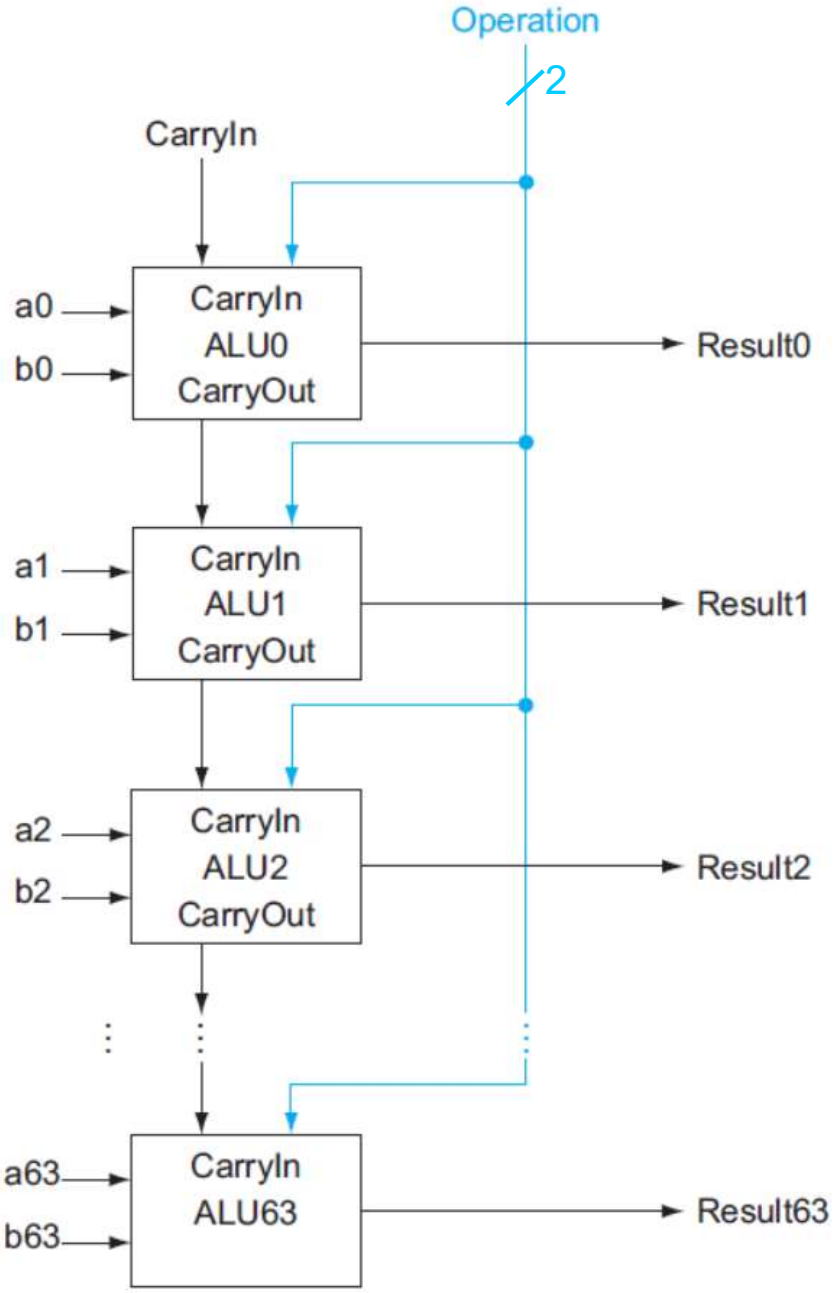


Operation	Function
0	AND
1	OR
2	ADD

Costruire una ALU a 64 bit



da 1 a 64 bit



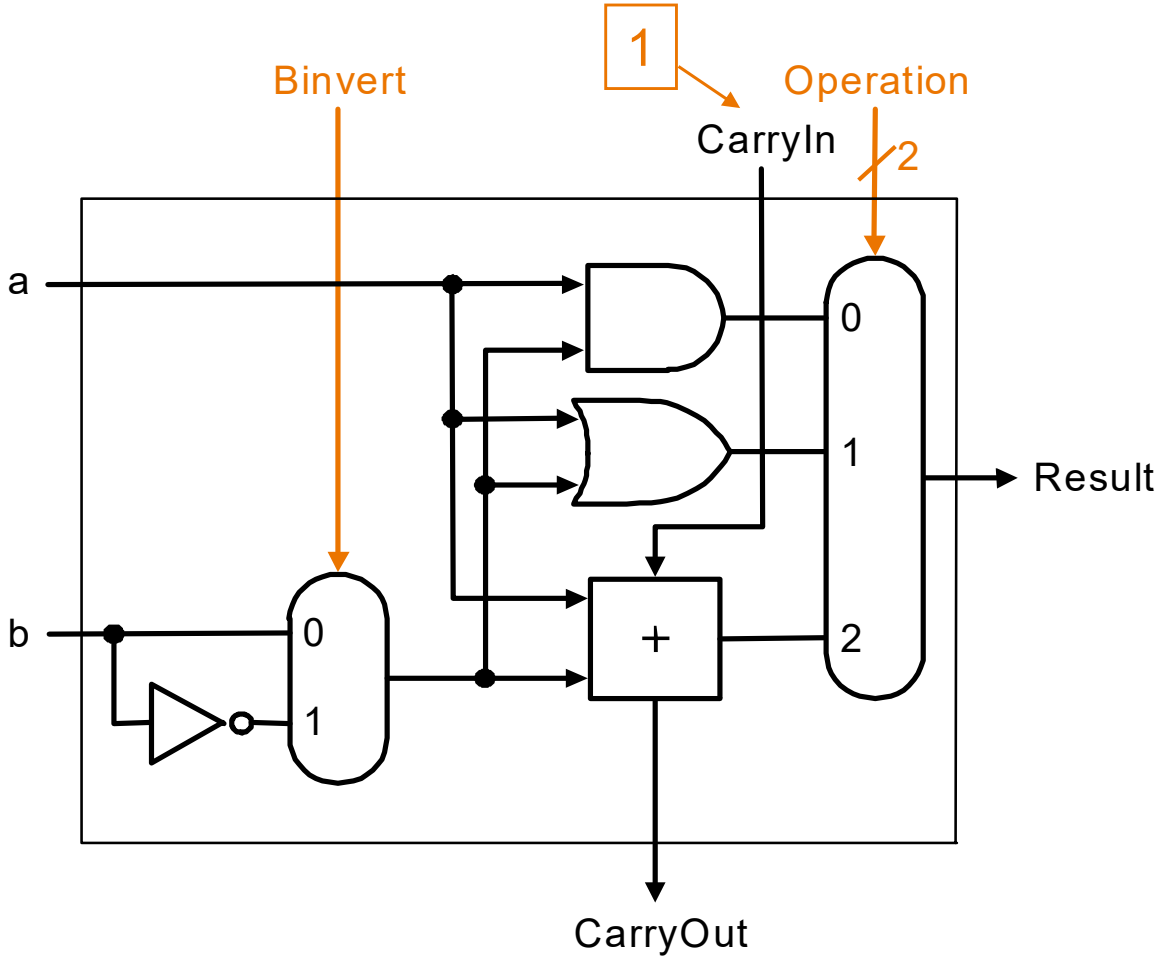
Un sommatore costruito in questo modo è chiamato "sommatore con riporto seriale"

Problema:
il risultato è disponibile solo dopo che ALU63 ha ricevuto il carry

Altre implementazioni "con riporto parallelo" o "Carry Look Ahead" superano questo problema (v. Lezione-06 e PHRV1 appendice A.6)

Supporto per la sottrazione (a – b)

- Approccio: fare a + (-b)
- Usare la rappresentazione in complemento a due semplifica la logica necessaria per fare la negazione: -b = ~b+1



Supporto per la `slt` e per il test di uguaglianza

- **Funzione set-on-less-than (`slt`)**
  - Se a < b produce 1, altrimenti 0
  - Idea: usare la sottrazione
    (a-b) < 0 implica a < b
  - Per vedere se (a-b) è negativo basta che verifico se il bit più significativo vale 1

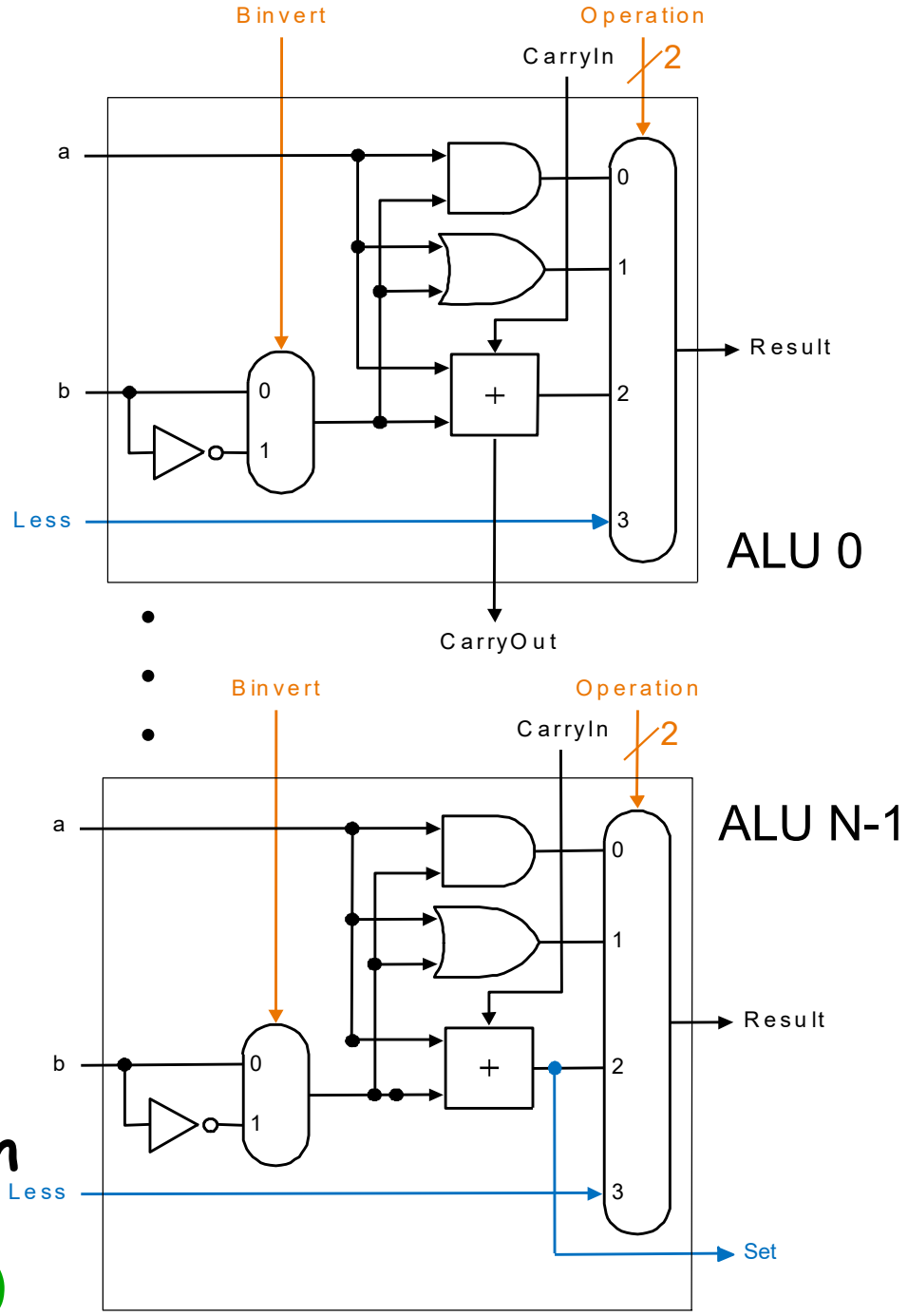
Supporto per il test di uguaglianza

- Ci servirà per supportare le condizioni degli statement if/while/for
  per decidere se devo eseguire quanto segue o «saltare» a un altro pezzo di codice
- Se a == b allora produco 1, altrimenti 0
- Idea: usare nuovamente la sottrazione
  (a - b) = 0 implica a = b

→ In entrambi i casi si può si può sfruttare la rete per fare la sottrazione

Supporto alla `slt` *nella ALU*

- Inserisco il caso "3" al multiplexer, corrispondente a `slt`
- Nelle ALU 0,1, ..., N-1 questo ingresso è collegato a un filo "Less" che costituisce un "bypass" ingresso-uscita
- Nella ALU N-1 inoltre (bit più significativo), controllo il valore di uscita 'set' che indica come è andato l'esito del confronto a<b
- Mentre per gli interi in complemento a due è semplice sapere quando il risultato è negativo (segnale 'set'), in generale per rivelare l'overflow ho bisogno di una rete logica un poco più complessa (v.slide finale)



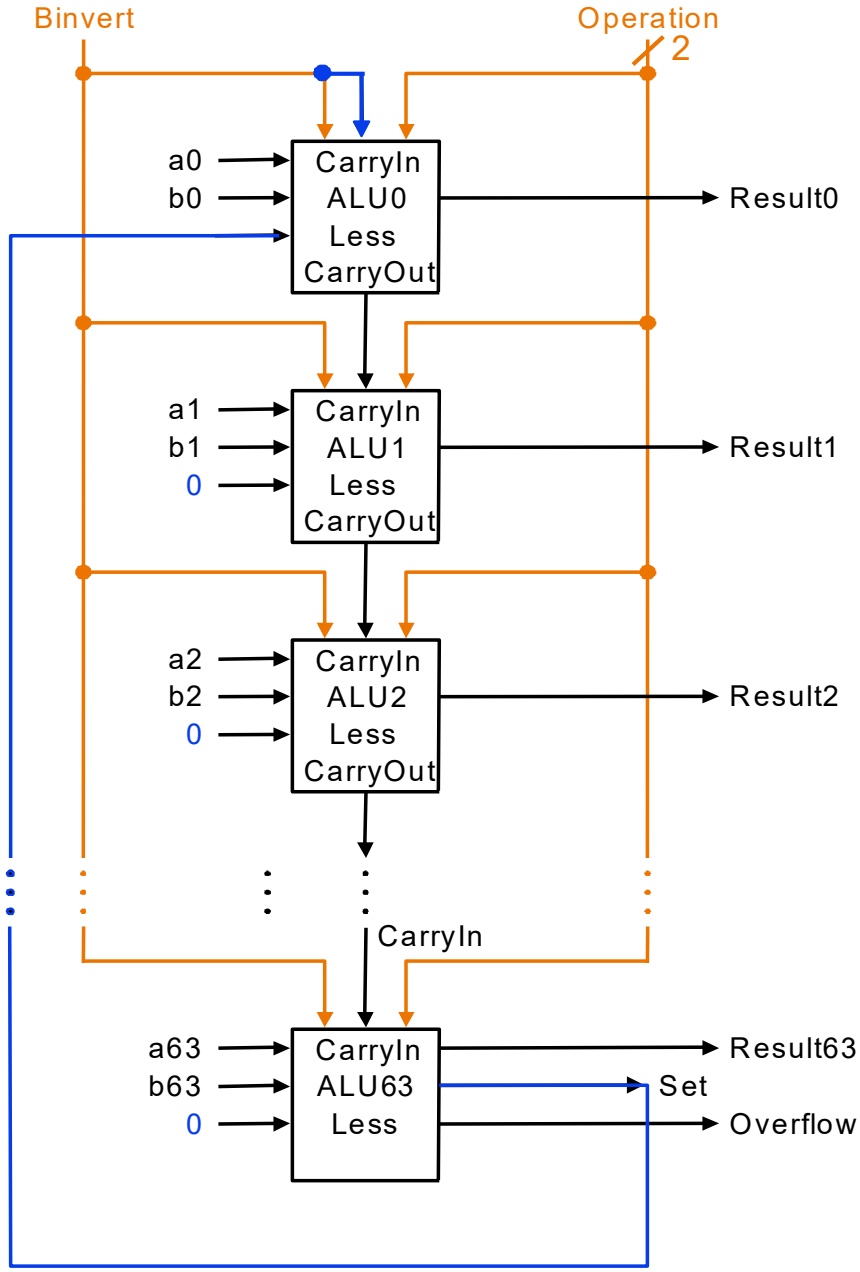
Supporto alla `slt` *fuori* dalla ALU

- Se il confronto ha esito positivo il filo less, manda sul bit meno significativo il risultato (1)

- Aggiustiamo il $CarryIn_0$:

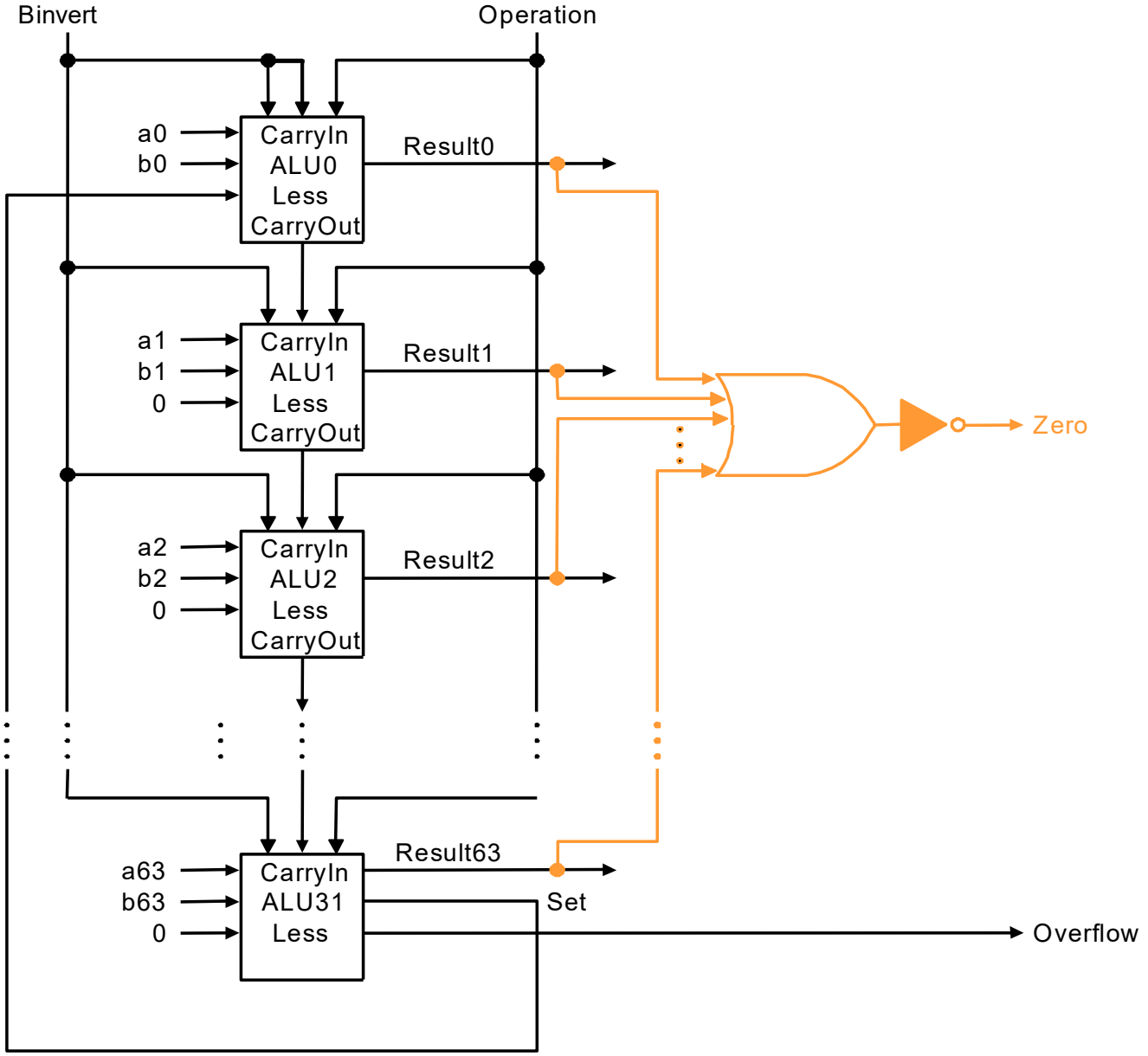
Operation	Function	CarryIn0	BInvert
0	AND	0	0
1	OR	0	0
2	ADD	0	0
2	SUB	1	1
3	SLT	1	1

→ $CarryIn_0$ = BInvert



Test per l'uguaglianza (eq)

- La rete fornisce il valore 1 quando tutti i bit del risultato sono a zero, in quanto: (a-b)==0 --> a==b
- Chiameremo tale segnale "Zero" e ci serve per decidere se l'esito del confronto è positivo o meno



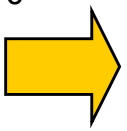
•Nota: "Zero" vale 1 quando il risultato è zero!

ALUcontrol

- Aggiungiamo quindi la funzione «EQ» che sfrutta la sottrazione e produce un ulteriore filo di uscita dalla ALU («Zero»)
- I fili di ingresso {Operation,Binvert} possono essere raggruppati nei fili chiamati «ALUcontrol» come mostrato in tabella

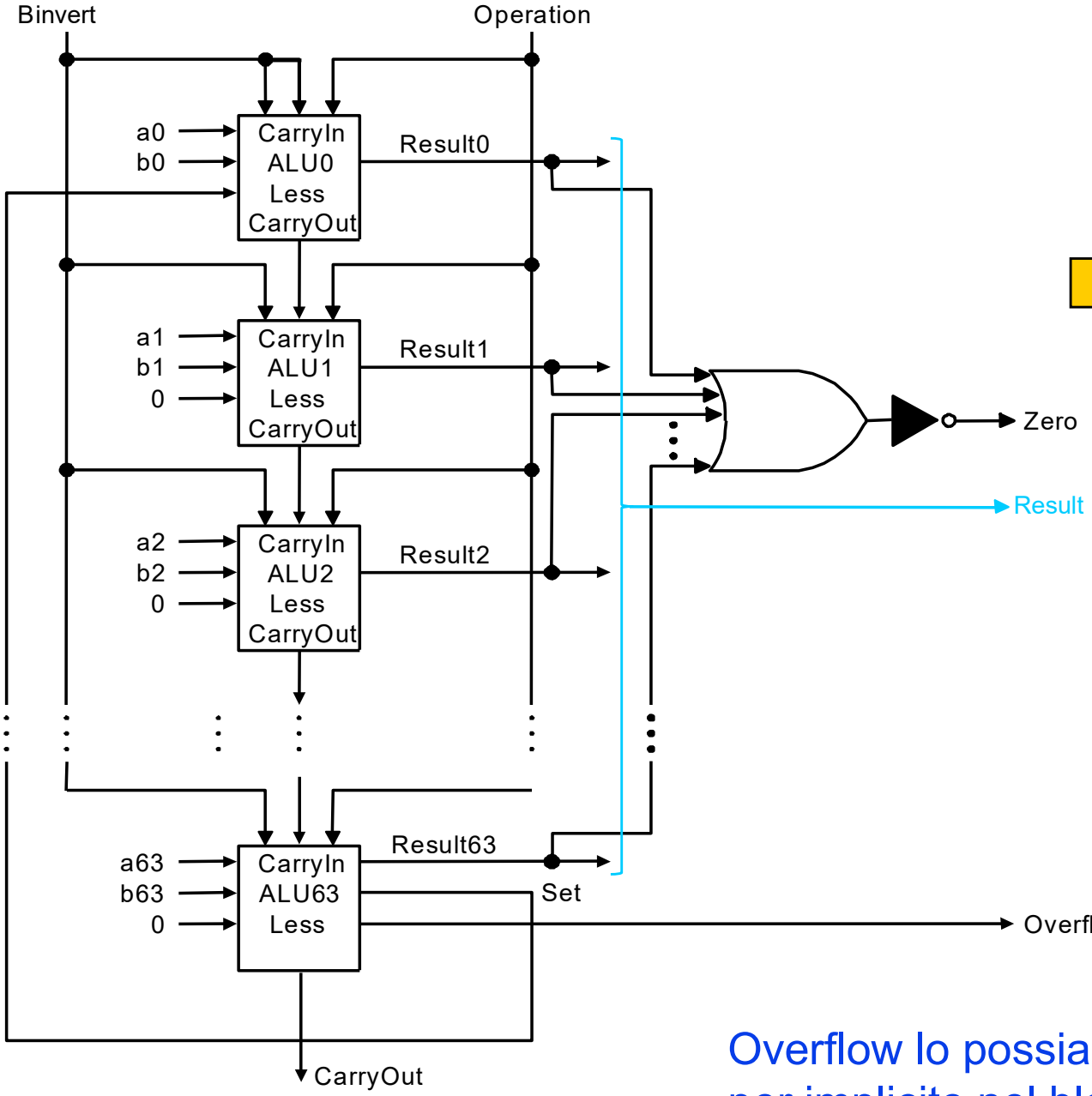
Operation	Function	BInvert
0	AND	0
1	OR	0
2	ADD	0
2	SUB	1
3	SLT	1
2	EQ	1

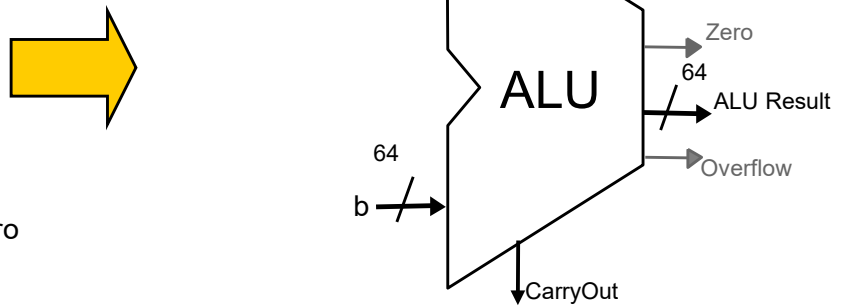
Binvert → $ALUcontrol_2$
$Operation_1$ → $ALUcontrol_1$
$Operation_0$ → $ALUcontrol_0$



ALUcontrol	Function
0 00	AND
0 01	OR
0 10	ADD
1 10	SUB
1 11	SLT
1 10	EQ

Blocco "ALU a 64-bit" – situazione finale



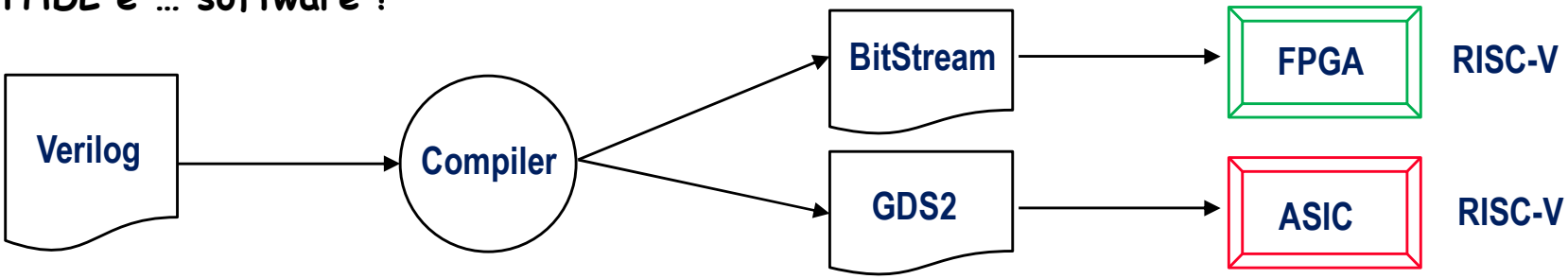


Overflow lo possiamo dare per implicito nel blocco ALU

(v. PHRV1, appendice A.4)

Lezione 4: Introduzione agli Hardware Description Languages (HDL)

- **Il Verilog: obiettivi principali**
  - Descrivere un'architettura con un linguaggio formale (es. come il *C* per gli algoritmi)
  - Standardizzare tale linguaggio per un'ampia adozione (Verilog-95→Verilog-2001→IEEE-1364-2005→IEEE-1800-2009→IEEE-1800-2017)
  - L'attuale standard è chiamato 'SystemVerilog' (IEEE-1800-2017)
- **Vantaggi di un linguaggio di descrizione dell'hardware (HDL)**
  - Permette di simulare il comportamento di un'architettura, eliminarne i bug
  - La rete progettata viene compilata («sintetizzata») generando automaticamente hardware (molto più veloce e con consumi energetici estremamente più bassi rispetto al solo software)
  - Nota: l'HDL è … software !



- **Il Verilog è basato sul *C* (assomiglia al *C*)**
  - Permette di costruire dalle reti usando la programmazione strutturata
  - Il VHDL (standard IEEE-1067-2019) assomiglia al linguaggio Ada
- **La differenza fondamentale Verilog vs. *C***
  - Il *C* descrive un'evoluzione temporale di un algoritmo che usa risorse computazionali «fisse», mentre il Verilog descrive un'estensione spaziale di un sistema con risorse potenzialmente tutte operanti in parallelo

Verilog: Tipi di dato e Operatori

- Tipi di dato
  - **wire** indica un segnale (che viaggia su un filo, quindi in hardware è uno «wire»)
  - **reg** (register) indica una variabile che memorizza un valore (un registro a 1 bit)
    - Un reg non necessariamente corrisponde ad un registro fisico, anche se spesso è così

- Un registro o wire a 32 bit si dichiara con
  - wire[31:0] Y
  - reg[31:0] X

  - Si può fare accesso ad un sottoinsieme di tali bit indicandolo con la notazione [starting_bit : ending_bit]

- Si possono combinare più registri per formare una struttura "register file" ovvero una "memoria" *
  - reg[31:0] registerfile[0:31]
  - Si fa accesso al singolo registro con la classica notazione registerfile[num_registro]

* Nota: Memorie e Reti sequenziali verranno affrontate in una lezione successive.
Per ora si può assumere che una memoria o registro sia l'equivalente di una "lavagna"

Verilog: valori e costanti

- I possibili valori di un registro o un filo sono:
  - 0 o 1 per rapprentare i valori logici falso o vero rispettivamente
  - X per rappresentare un "don't care" (non-specificato) o un valore non noto
  - Z per rappresentare lo stato di alta impedenza (filo "scollegato" o uscita "libera")

- I nomi simbolici delle costanti si dichiarano con parameter
  - parameter A=1

- I valori delle costanti possono essere specificati in varie basi
  - (la <base> 2,8,10,16 si indica rispettivamente con b,o,d,h)

    Notazione: <n.bit>" ' "<base><valore>

    Esempio:
  - 4'b0101 indica una costante binaria a 4 bit che vale 5
  - 8'h4 indica una costante binaria a 8 bit che vale 4

- Repliche e Concatenazioni
  - {x{ bitfield }} replica i bit indicati x volte, es. {3{2'b01}} crea 010101
  - {bitfiled1, bitfield2} concatena i due bitfield, es. {A[7:4],B[5:2] }

Verilog: operatori (1/2)

- **In gran parte simile al linguaggio C**

Relational Operators

Operator	Application
<	a < b // is a less than b? // return 1-bit true/false
>	a > b // is a greater than b?
>=	a >= b // is a greater than or // equal to b
<=	a <= b // is a less than or // equal to b

Arithmetic Operators

Operator	Application
*	c = a * b ; // multiply a with b
/	c = a / b ; // int divide a by b
+	sum = a + b ; // add a and b

-	diff = a - b ; // subtract b
	// from a

%	amodb = a % b ; // a mod(b)

Logical Operators

Operator	Application

&&	a && b ; // is a and b true?
	// returns 1-bit true/false

||	a || b ; // is a or b true?
	// returns 1-bit true/false

!	if (!a) ; // if a is not true
	c = b ; // assign b to c

Verilog: operatori (2/2)

- Gli operatori sono in gran parte simili a quelli del linguaggio C

Nota: evidenziate in blu le parti con comportamento diverso rispetto al C

Equality and Identity Operators

Operator	Application
=	`c = a ; // assign a to c`

==	`a == b; //equal ?`
	`//0==0→1,1==1→1,0==1→0,`
	`// 0==X→X,1==X→X`
	`//e.g., `hx == `h5 returns X`

!=	`c != a ; // is c not equal to`
	`// a, retruns 1-bit true/`
	`// false logic equality`

===	`a === b; //identical? (returns 1 bit)`
	`//also 0,1,X,Z must be identical to get 1`
	`//`
	`//e.g., `hx === `h5 returns 0`

!==	`a !== b; //not identical? (returns 1 bit)`
	`//also 0,1,X,Z must be identical to get 1`
	`//`
	`//e.g., `hx !== `h5 returns 1`

Unary, Bitwise and Reduction Operators

Operator	Application
+	Unary plus & arithmetic(binary) addition
-	Unary negation & arithmetic (binary) sub-traction
=&	`b = &a ; // AND all bits of a` *
=|	`b = |a ; // OR all bits` *
=^	`b = ^a ; // Exclusive or all bits of a`*
~&, ~|, ~^	NAND, NOR, EX-NOR all bits to-gether* `c = ~& b ; d = ~| a; e = ^c ;`

~,&, |, ^	bit-wise NOT, AND, OR, EX-OR
	`b = ~a ; // invert a`
	`c = b & a ; // bitwise AND a,b`
	`e = b | a ; // bitwise OR`
	`f = b ^ a ; // bitwise EX-OR`

~&, ~|, ~^	bit-wise NAND, NOR, EX-NOR
	`c = a ~& b ; d = a ~| b ;`
	`e = a ~^ b ;`

Shift Operators and other Operators

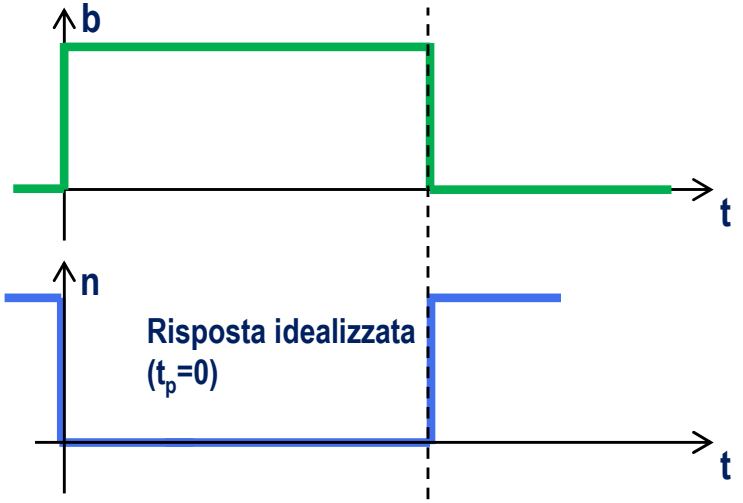
Operator	Application
<<	`a << 1 ; // shift left a by 1-bit`
>>	`a >> 1 ; // shift right a by 1`
?:	`c = sel ? a : b ; /* if sel is true c = a, else c = b , ?: ternary operator */`
{}	`{co, sum } = a + b + ci ; /* add a, b, ci assign the overflow to co and the result to sum: operator is called concatenation */`
{{}}	`b = {3{a}} /* replicate a 3 times, equivalent to {a, a, a} */`

* Opera riducendo tutti i bit a uno solo:
in altri termini opera "orizzontalmente"

Verilog: Reti Combinatorie e Timing associato

- Una rete combinatoria si indica con «assign» seguita dall'equazione booleana:

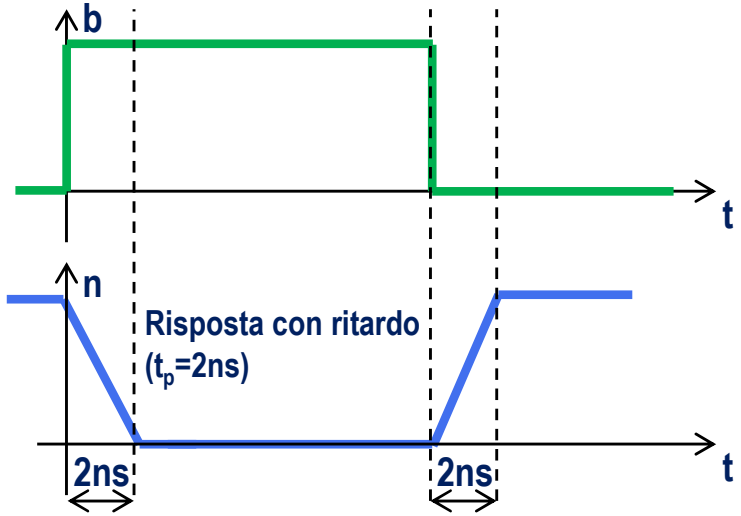
```
assign   n=~b
```



- Per indicare il tempo richiesto da tale operazione («tempo di propagazione») si aggiunge questa notazione:

```
assign  #2  n=~b
```

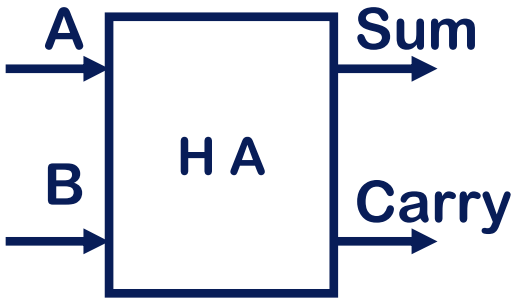
  - '#2' indica un ritardo di 2 ns ('ns' è il default – ma si può cambiare la scala)

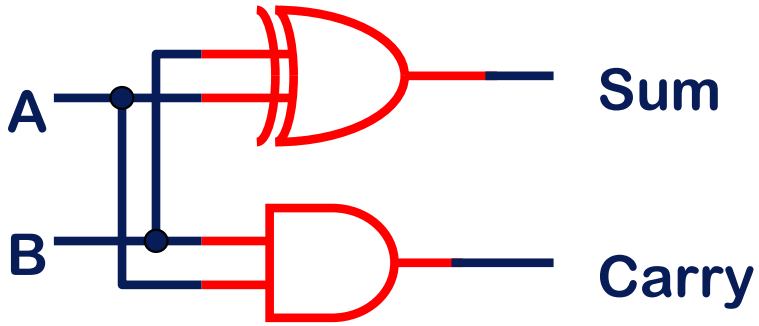


- Nota: "assign" indica quindi un assegnamento continuo
  - Ad ogni variazione delle variabili a destra, viene generata (col ritardo specificato) una variazione delle variabili a sinistra del segno di '='
  - Inoltre ciò avviene in parallelo ad ogni altro assign

Esempio: Half Adder in Verilog

- Dichiaro innanzitutto un modulo con determinati nomi per gli ingressi e le uscite:





ingressi uscite comportamento definizione di modulo

```
module half_adder(a,b,  sum,carry);
    input  a,b;          //ingressi su 1-bit
    output sum,carry;    //uscite su 1-bit
    assign sum = a ^ b;  // a XOR b
    assign carry = a & b; // a AND b
endmodule
```

- Verilog keywords: input, output, module/endmodule

Assegnamento procedurale: initial e always

- Quando si assegna un valore ad un registro (keyword 'reg'), l'assegnamento DEVE essere fatto:

  i) All'inizio della simulazione (caso tipico del segnale di reset) tramite la keyword 'initial', es.:

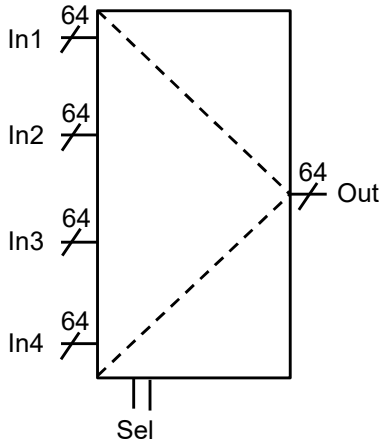
```
reg reset;
initial reset=0; //initial viene eseguito nel primo timestep di simulazione
```

  ii) Al verificarsi di un evento (es. al variare di segnale) tramite la keyword 'always':

```
reg out; wire in1, in2;
always @(in1 or in2) out=1; //always → check ad ogni timestep in cui variano
                            //i segnali indicati nella lista di sensibilità @(...)
```

- Assegnamento non-bloccante ('<=') e bloccante ('=')
  - Negli assegnamenti procedurali posso avvalermi di due diverse sintassi:
    - Assegnamento NON-BLOCCANTE: es. `out1<=1; out2<=0; // i due registri vengono assegnati in parallelo`
    - Assegnamento BLOCCANTE: es. `out1=1; out2=0; // viene prima assegnato out1 e poi out2`
  - Nota: gli assegnamenti all'interno di uno stesso comando initial o always possono essere racchiusi in un cosiddetto «blocco begin-end»; es. `begin out1<=1; out2<=0; end`
- Modello di esecuzione: ad ogni timestep di simulazione
  - Vengono valutate tutte le espressioni bloccanti e assegnate (in successione se all'interno di begin-end)
  - Vengono valutate le parti a destra degli assegnamenti non-bloccanti e poi assegnate tutte in parallelo

Esempio: multiplexer a 4 ingressi di 64 bit




```
module Mult4to1(In1,In2,In3,In4,Sel,  Out);
   input[63:0] In1,In2,In3,In4; //four 64-bit inputs
   input[1:0] Sel;              //selection signal
   output[63:0] Out; reg[63:0] Out; //64-bit output
   always @(In1 or In2 or In3 or In4 or Sel)
   casex(Sel)  //4->1 mux
      0: Out <= In1;
      1: Out <= In2;
      2: Out <= In3;
      default: Out <= In4;
   endcase
endmodule
```

Notazione per indicare più bit

Notazione "reg":
memoria interna (registro)

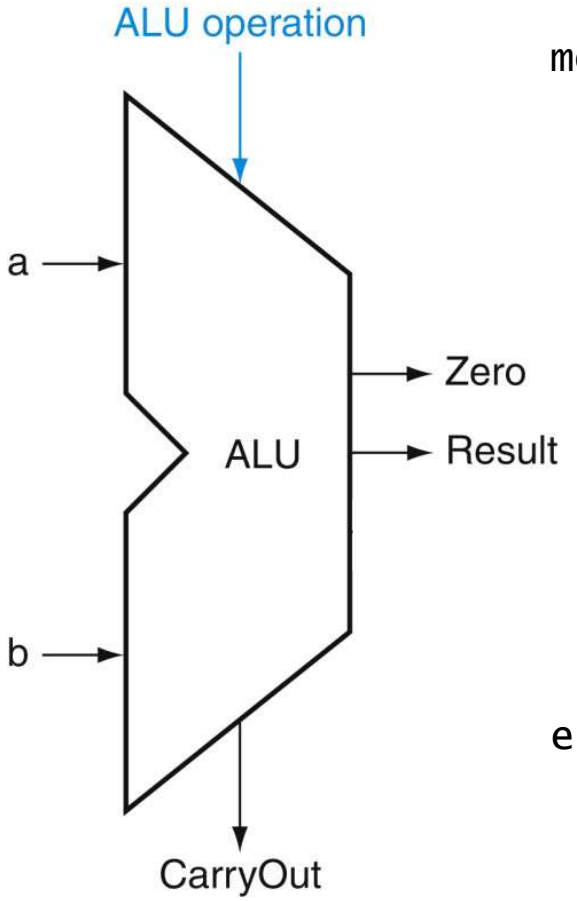
Notazione «always @(…)» :
qualcosa accade quando vengono
variati i seguenti segnali indicati in "@(.....)"

Notazione «casex/endcase»
funziona come il «switch» del linguaggio C

- **Verilog keywords:** reg, always @(…), casex/endcase

Nota1: il registro non c'era nella definizione iniziale del multiplexer; è stato messo per comodità di avere un valore stabile in uscita

Esempio: ALU

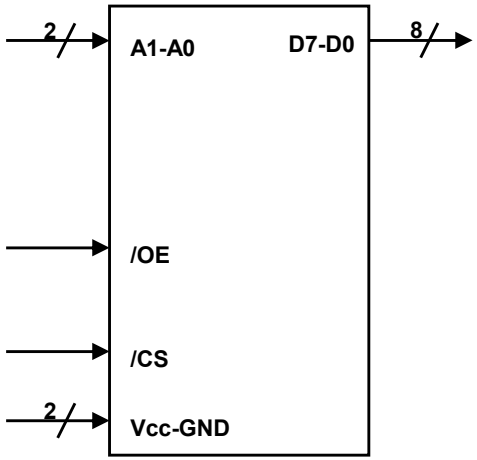


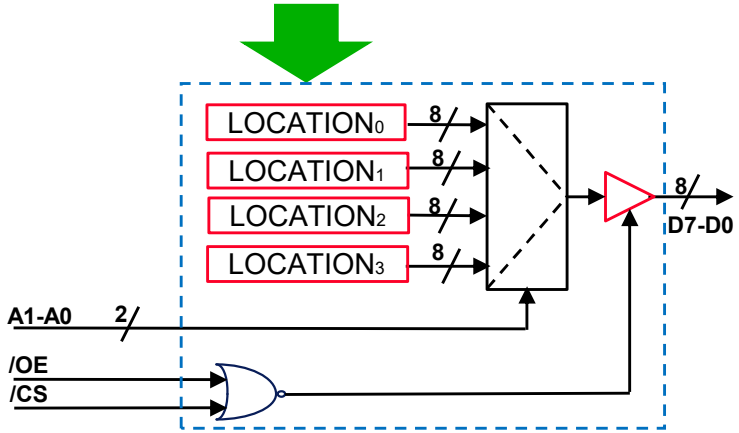

```
module alu(a,b,aluoperation,  result,zero, carryout);
   input[63:0] a, b; input[2:0] aluoperation;
   output[63:0] result; reg[63:0] result;
   output zero, carryout;
   assign zero = (result==0);
   always @(aluoperation or a or b) //re-evaluate if these change
      casex (aluoperation)
         0: result <= a & b;
         1: result <= a | b;
         2: result <= a + b;
         6: result <= a - b;
         7: result <= a<b ? 1:0;
         default: result<=0; //default to 0, should not happen
      endcase
endmodule
```

Nota1: sul Patterson viene fatto il caso con 4 bit di ALUctl: noi ne useremo 3 (il che implica che non vedremo il caso della 'nor' della fig. A.5.15).

Esempio: ROM 4x8

```
module ROM(d7_d0, address, cs_, oe_);
  input  [1:0]  address;
  output [7:0]  d7_d0;
  input         cs_;
  input         oe_;

  wire alfa=~(cs_|oe_);
  assign d7_d0=
    (alfa==0)? 'BZZ : LOCATION(address);

  function [7:0] LOCATION;
   input [1:0] address;
   casex(address)
    0:       LOCATION = 0;
    1:       LOCATION = 1;
    2:       LOCATION = 2;
    3:       LOCATION = 3;
   endcase
  endfunction
endmodule
```



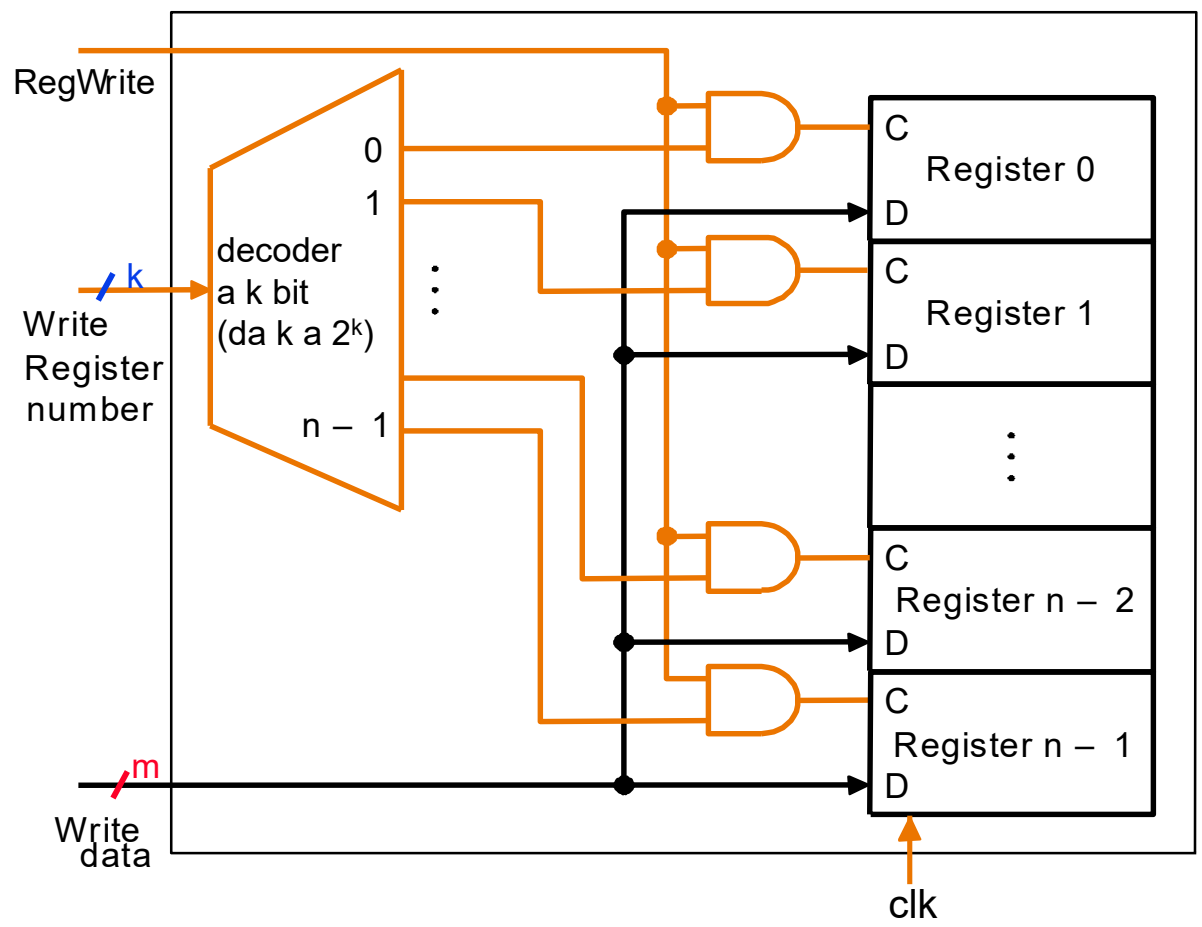


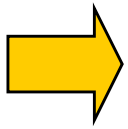
- Verilog keyword: function

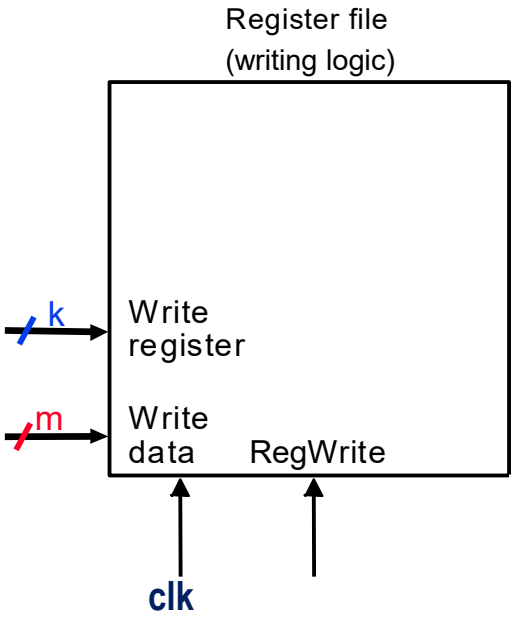
REALIZZAZIONE DEL PROCESSORE RISC-V

(V. PHRV1, APPENDICE A.8, CAP. 4.3)

Register File: logica di scrittura







n = numero di registri disponibili (e.g. 32)

k = $\log_2(n)$ (e.g. 5)

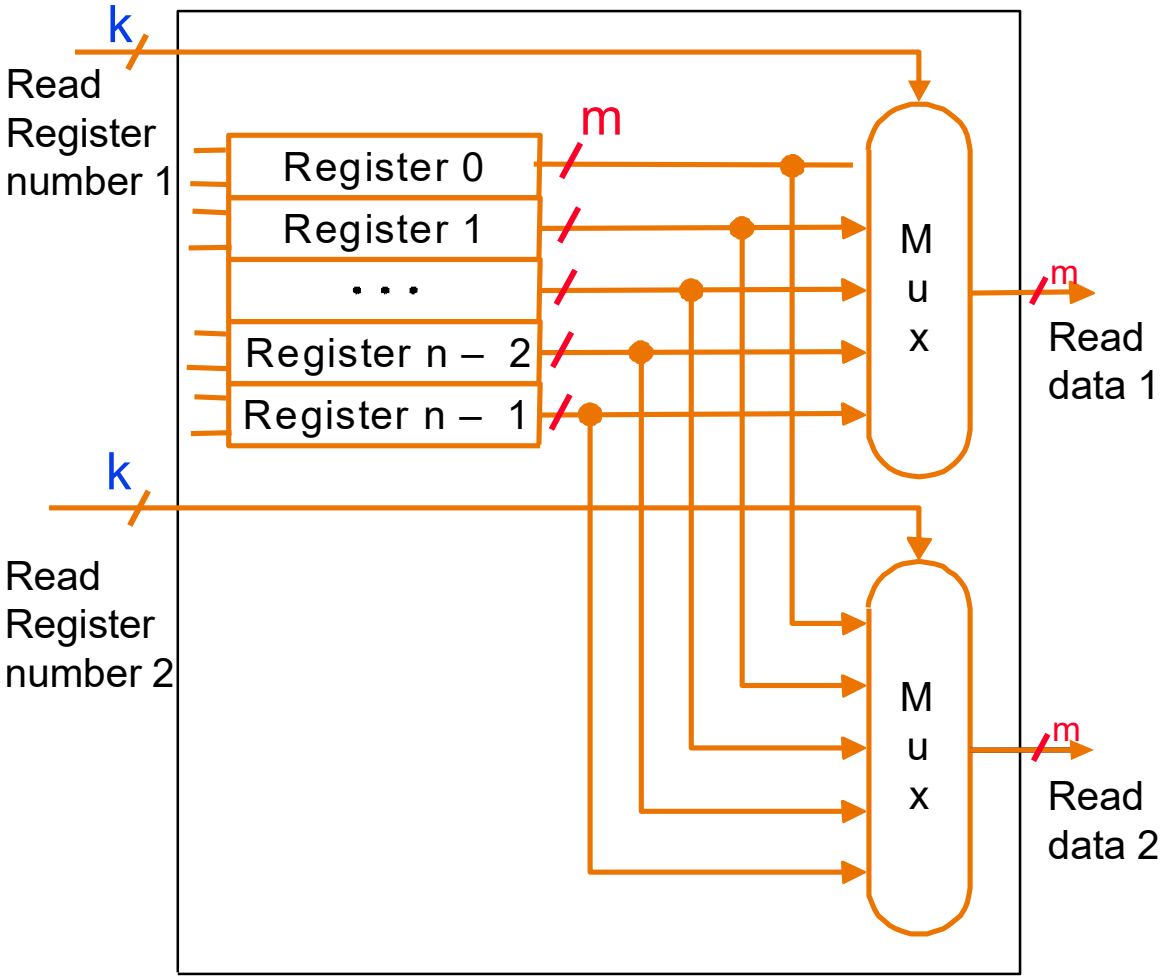
m = larghezza dei registri (e.g. 64)

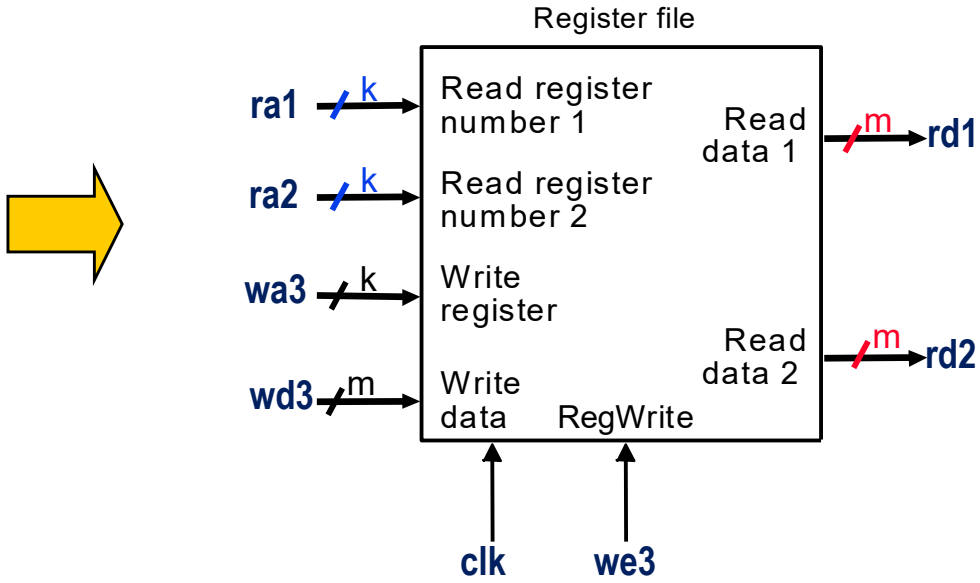
Nota: la struttura di un registro verrà esaminata in dettaglio in una lezione successiva

Nota: sul Patterson viene dato per implicito il segnale clk.

Register File: logica di lettura

- Si usano i flip-flop di tipo D



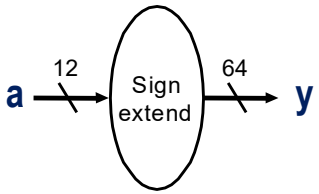



```
module regfile(clk, we3, ra1, ra2, wa3, wd3,   rd1, rd2);
   input clk,we3; input[4:0] ra1, ra2, wa3; input[63:0] wd3;
   output[63:0] rd1, rd2; reg[63:0] rd1, rd2;
   reg[63:0] rf[0:63];
   // three ported register file: read two ports
combinationally
   // write third port on falling edge of clk; register 0
hardwired to 0
   always @(negedge clk) if (we3) rf[wa3] <= wd3;
   always @(ra1) rd1 <= (ra1 != 0) ? rf[ra1] : 0;
   always @(ra2) rd2 <= (ra2 != 0) ? rf[ra2] : 0;
Endmodule
```

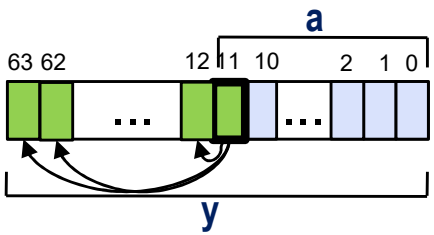
Altra logica sparsa e Memoria principale

- Nel caso della beq, l'offset di salto può essere sia positivo che negativo e specificato su 12 bit:
  occorre una rete di estensione dell'operando con segno da 12 a 64 bit
  - Questo è anche il caso della load e della store

Gen. Immediate Unit

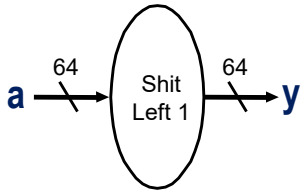



```
module signext(input [11:0] a,
               output [63:0] y);
  assign y = {{52{a[11]}}, a};
endmodule
```

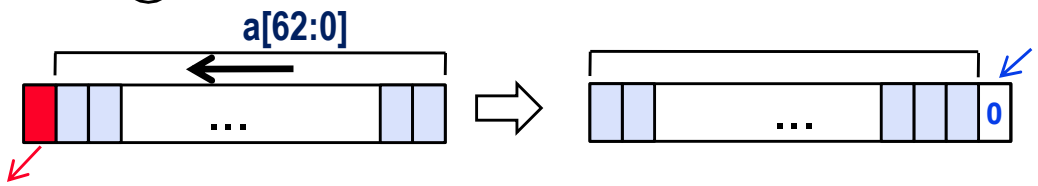


- Inoltre: rete per moltiplicare per due (branch)

Multiply-by-2 Unit

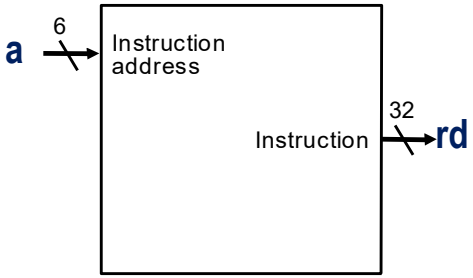



```
module sl1(input [63:0] a, output [63:0] y);
  // shift left by 1
  assign y = {a[62:0], 1'b0};
endmodule
```

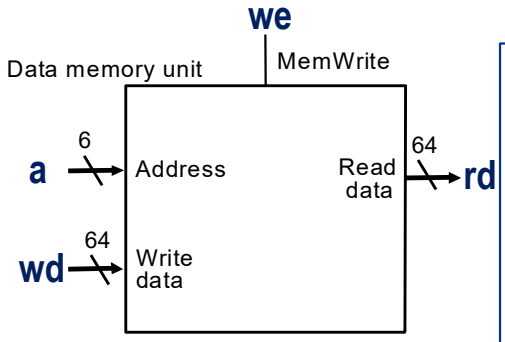


- La memoria instruzioni e la memoria dati possono essere viste così:

Instruction memory




```
module imem(input [5:0] a,
                  output [31:0] rd);
  reg [31:0] ROM[0:63]; //256 Bytes
                        // (i.e., 64 words)
  assign rd = ROM[a];
endmodule
```



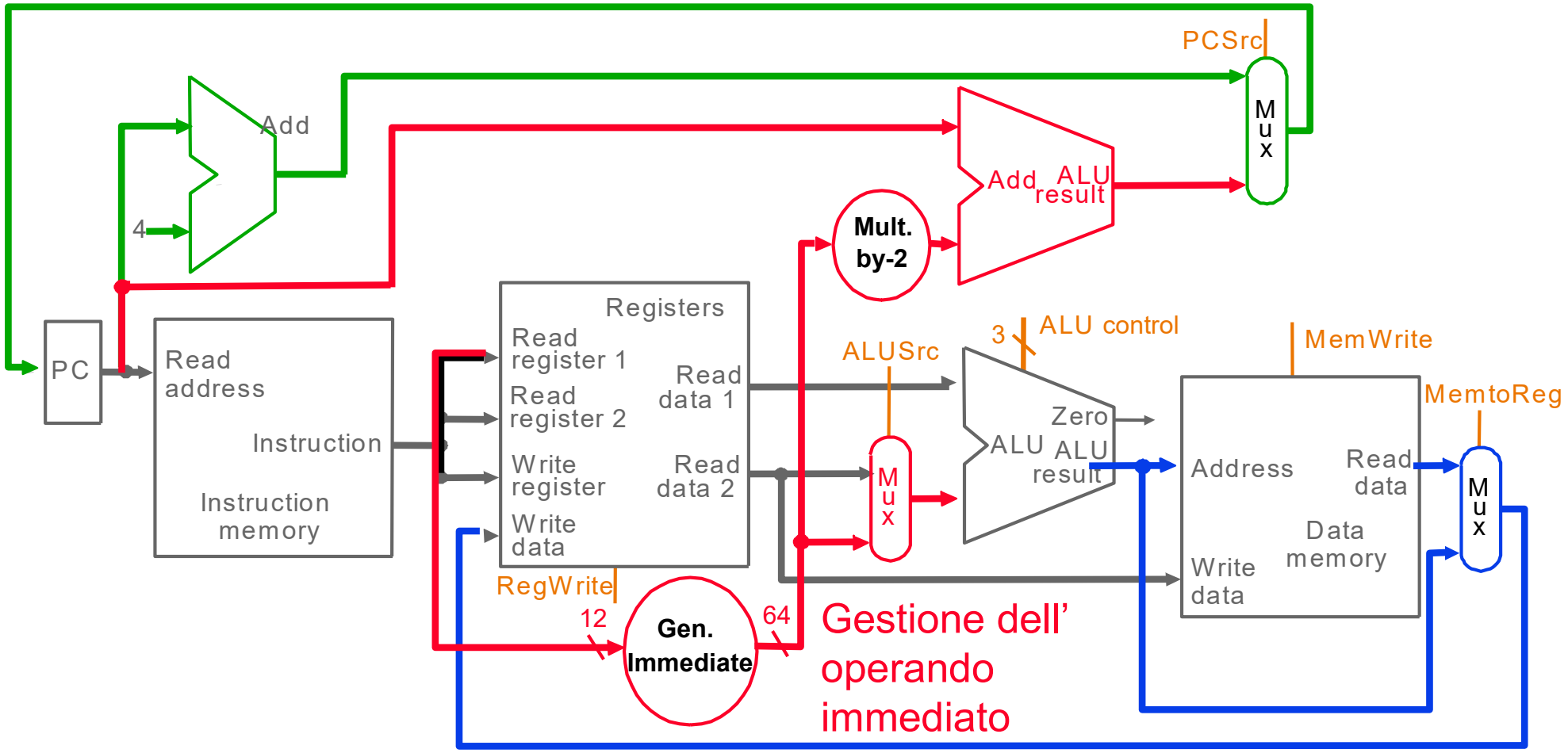

```
module dmem(input clk, input we,
     input[5:0] a; input[63:0] wd;
     output[63:0] rd);
  reg[63:0] RAM[0:7]; //64 Bytes
                      //(i.e., 8 dwords)
  //a=byte aligned,RAM=dword alig.
  assign rd=RAM[a[5:3]];
  always @(posedge clk)
  if (we) RAM[a[5:3]] <= wd;
endmodule
```

Nota: sul Patterson la memoria dati ha anche il segnale MemRead (qui lo eliminiamo per semplicità)

Costruzione del Datapath del processore RISC-V

Gestione indirizzo istruzione successiva

I segnali di controllo vengono pilotati in base a "op-code" e "funct" dell'istruzione:



Gestione del risultato della ALU o dato dalla memoria

- c.f. PHRV1 figura 4.11

*Nota: "Gen. Immediate" comprende estensione del segno e raggruppamento dei bits relativi all'operando immediato nei vari formati di istruzione (per ora vediamo solo la estensione del segno)

(v. PHRV1, appendice A.7, A.8, A.10)

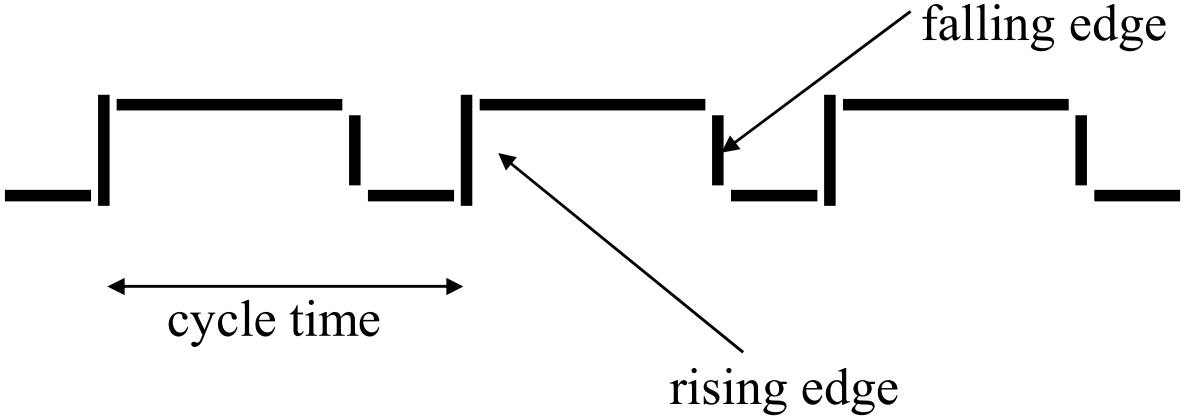
Lezione 5: Reti Logiche Sequenziali Elementari

- (richiamo) Reti COMBINATORIE (RC o CN==Combinatorial Network)
  - In qualunque istante
    le uscite sono funzione del valore che gli ingressi hanno in quell'istante
  - Il comportamento (uscite in funzione degli ingressi)
    è descritto da una tabella
  - NON CONTENGONO ANELLI DI RICHIUSURA

- (richiamo) Reti SEQUENZIALI
  - In un determinato istante
    le uscite sono funzione del valore che gli ingressi hanno in quell'istante
    e dei valori che gli ingressi hanno assunto precedentemente
  - Presentano Stati Interni (sono quindi reti dotate di *MEMORIA*)
    → La descrizione è più complessa di una semplice tabella
  - PRESENTANO ANELLI DI RICHIUSURA

Reti Sequenziali Asincrone e Sincrone

- Le Reti Sequenziali possono essere di tipo
  - asincrono (non richiede un segnale di clock)
  - sincrono (richiede un segnale di clock)
- Definiamo quindi cosa è un SEGNALE DI CLOCK
  - Il segnale di clock (o semplicemente «clock») è un segnale periodico* (idealmente un'onda quadra) che serve per fornire un riferimento temporale ben preciso per effettuare le operazioni
    - Ad esempio si può prendere come riferimento il fronte in salita (rising edge) o quello in discesa (falling edge)



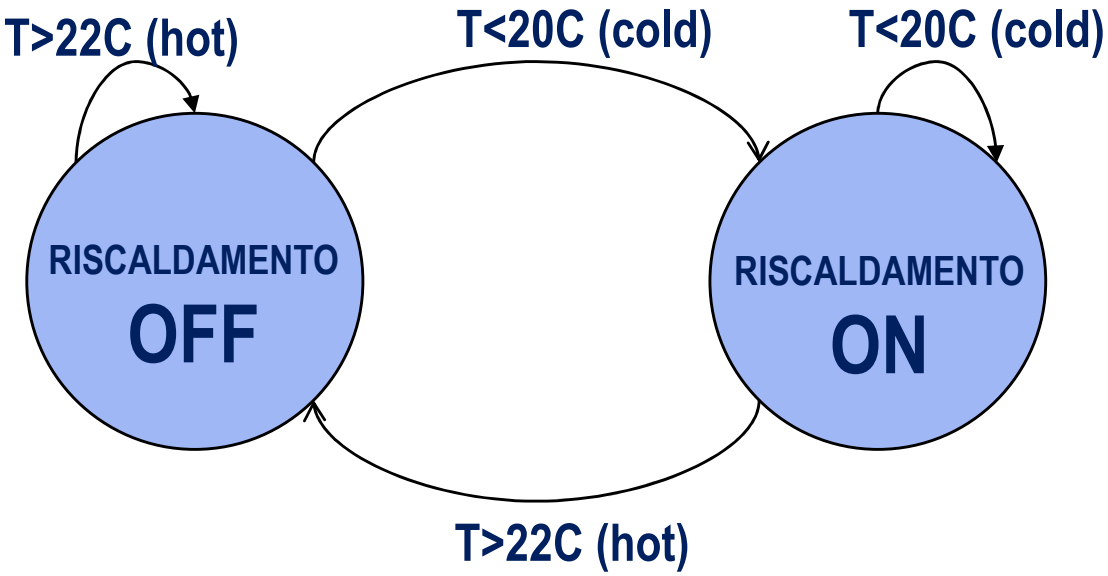
* Nota: caratterizzato quindi da un periodo Tc (cycle time) ovvero dalla sua frequenza fc=1/Tc

Nota2: Il segnale di clock viene tipicamente generato da un circuito elettronico chiamato "oscillatore" che usa la risonanza meccanica di un quarzo per generare una forma d'onda a una data frequenza

- Dobbiamo inoltre definire l'elemento di MEMORIA: per fare questo ci serviamo delle Macchine a Stati Finiti (FSM)

Macchine a Stati Finiti (o FSM==Finite State Machine)

- Un gruppo di stati e di transizioni



- Ci si sposta da uno stato all'altro seguendo una linea di transizione SE la condizione dalla transizione è soddisfatta

Macchine a Stati Finiti (2)

- Nel campo dell'architettura dei calcolatori, le FSM sono molto importanti in quanto godono delle proprietà di essere:
  - complete
    → tutti gli stati definiscono transizioni per tutti gli input
  - deterministiche
    → se si presenta un dato ingresso seguo sempre un dato arco

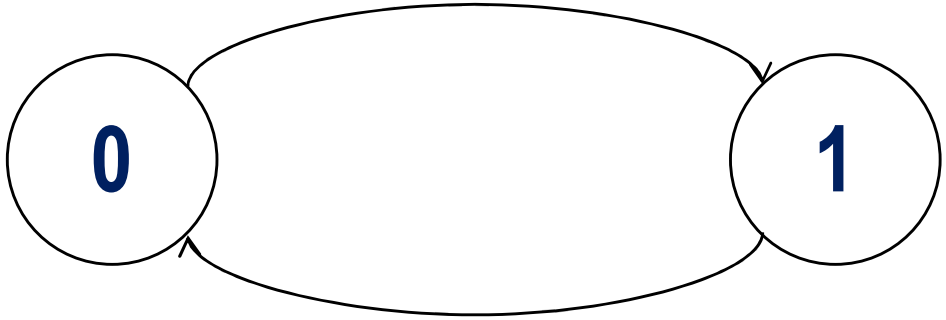
- Inoltre, le FSM possono essere implementate fisicamente tramite Reti Logiche dette Reti Sequenziali

- Le Reti Sequenziali possono avvicinarsi al comportamento ideale di una FSM facendo in modo che le transizioni fra uno stato ed il successivo avvengano in intervalli di tempo molto limitati:
  - o utilizzando particolari accorgimenti costruttivi (riduzione ALEE*)
  - o effettuando le transizioni su un fronte del clock

*Nota: la motivazione di rendere sensibili le reti agli ingressi solo in intervalli molto limitati risiede nella necessità evitare che oscillazioni temporanee dell'uscita si propaghino agli ingressi di altre reti generando ALEE (v. slide successive).

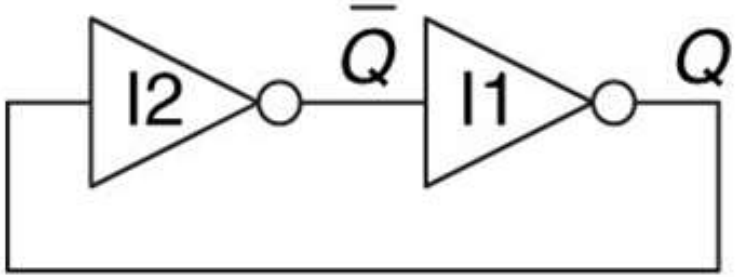
L'elemento di memoria

- La FSM più semplice che si possa immaginare (e che varia il proprio stato) è una rete a due soli stati
  - Tale rete non rappresenta altro che un bit di informazione e quindi permette di implementare un elemento di memoria



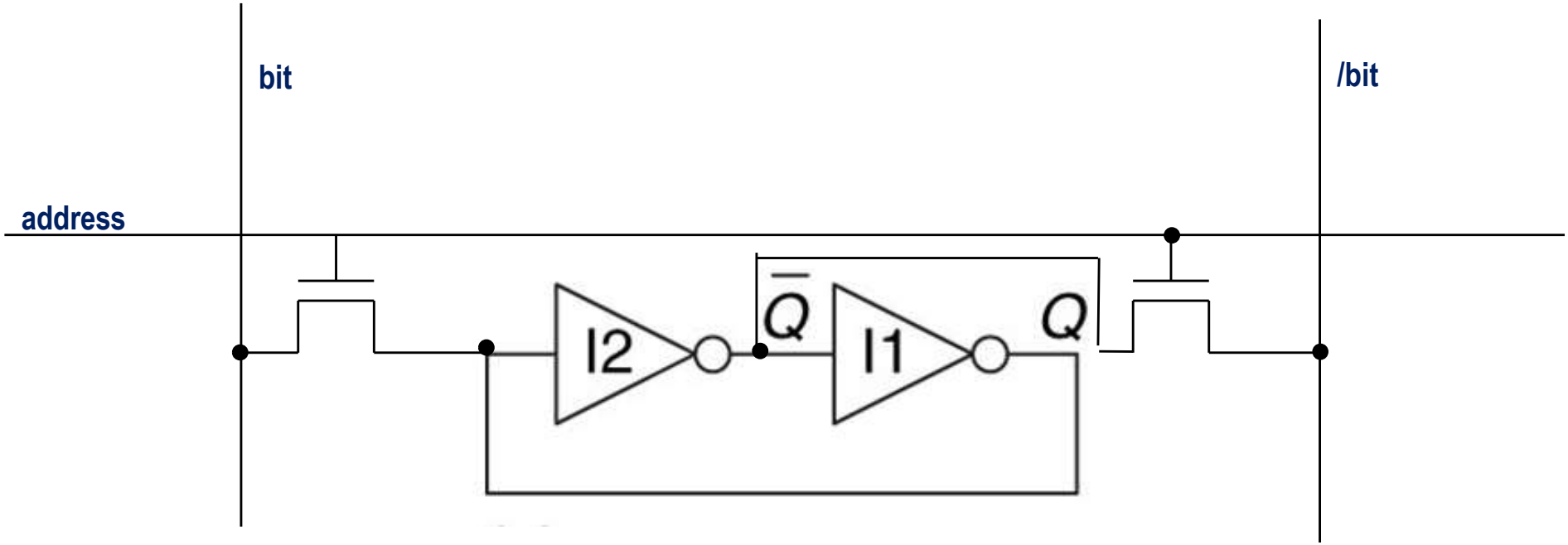
Realizzazione di elementi di memoria con reti combinatorie

- Abbiamo visto che l'elemento più semplice che riusciamo a realizzare facilmente su silicio è un inverter…
- Il modo più semplice per realizzare un elemento di memoria usando inverter è di utilizzare due inverter… richiusi ad anello



- Questo circuito digitale presenta DUE STATI stabili (si dice pertanto che è bistabile):
  - i) Q=0 (e di conseguenza /Q=1)
  - ii) Q=1 (e di conseguenza /Q=0)
- Però presenta solo due uscite e nessun ingresso
- Se ben dimensionato fisicamente funziona, altrimenti potrebbe anche oscillare (comportamento metastabile)

Perfezionando il bistabile usando la tecnologia CMOS

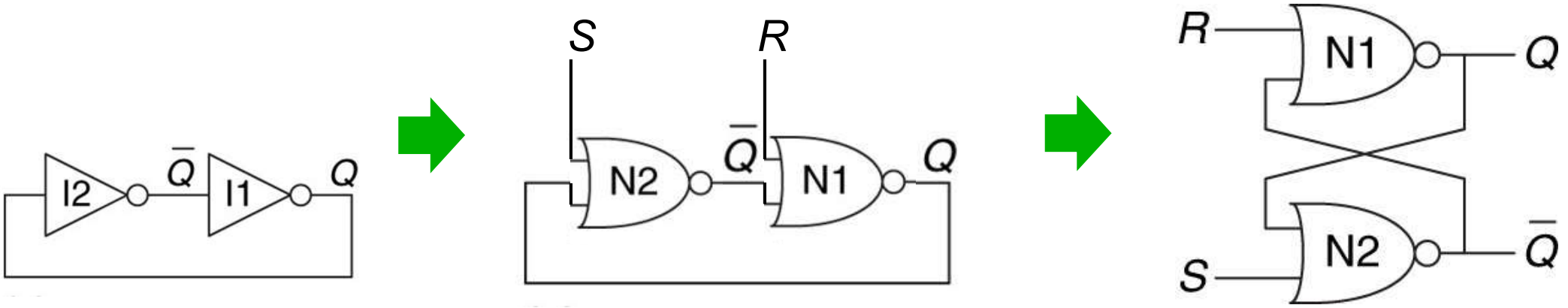


- In questo modo si può attivare l'elemento tramite il filo address e:
  - Imporre un 1 logico (bit=1,/bit=0) oppure uno 0 logico (bit=0,/bit=1)
  - Oppure leggere il valore Q senza distruggere il contenuto
- Questo tipo di cella di memoria viene usata per realizzare le SRAM (Static RAM)* → l'informazione rimane memorizzata fintanto che la cella è alimentata
  - In tecnologia CMOS si realizza con 6 transistor (2 per ogni inverter e due per i due transistor di passo)

*Nota: approfondiremo le SRAM nella lezione 16, dedicate al sottosistema di memoria

Latch SR

- L'elemento di memoria realizzato con due inverter è in grado di memorizzare l'informazione ma non ha una forma "autocontenuta" per memorizzare il dato (bit e /bit devono essere collegati solo in certi istanti, via address)
- Posso allora utilizzare porte NOR (anziché porte NOT) e usare uno dei due ingressi per modificare lo stato (analoghi a /bit e bit della singola cella) realizzando così il cosiddetto latch SR



- Nota - chiameremo:
  - Latch: elementi che variano lo stato su un livello di un segnale di clock
  - Flip-flop: elementi che variano lo stato su un fronte di un segnale di clock

Elemento di memorizzazione in logica sequenziale asincrona

- Il latch SR
  - l'uscita dipende dai segnali presenti sugli ingressi e dai segnali precedentemente presentati in ingresso

S	R	Q	/Q
0	0	Q	/Q
0	1	0	1
1	0	1	0
1	1	---	---

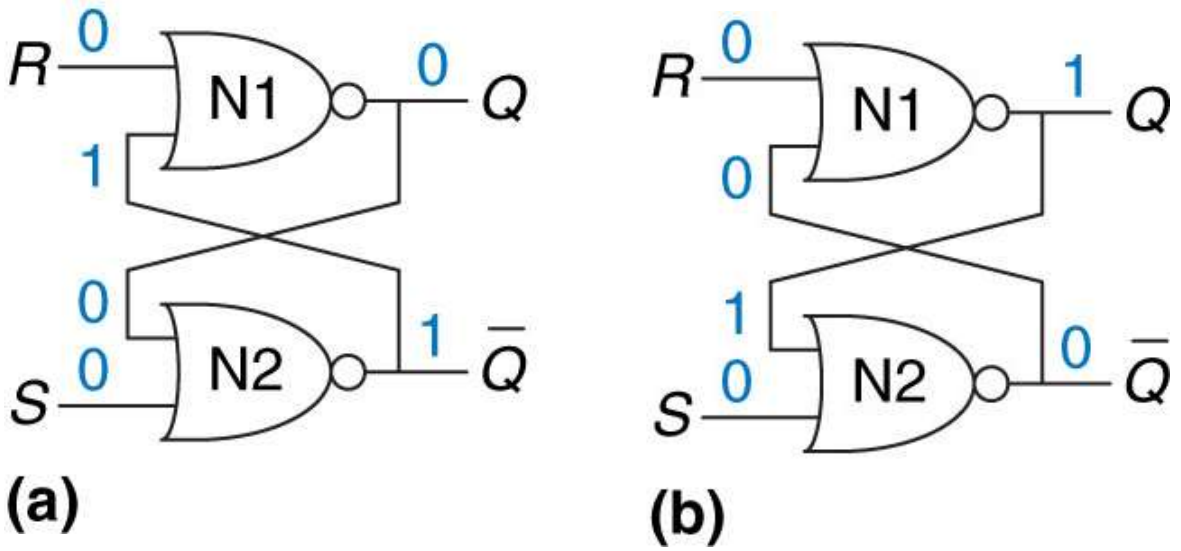
SET

RESET

Si dice che "conserva"

Tabella di Verità

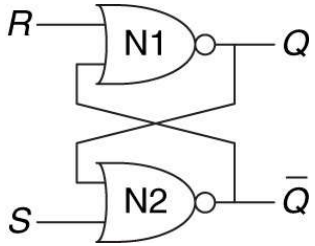
Come si vede da questo esempio per SR=00, sia Q=0 (a) che Q=1 (b) sono situazioni possibili:



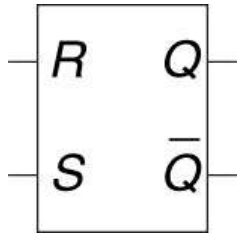
Schema logico

Latch SR in Verilog

```
module Latch_SR(Q,QN,S,R)
   input S,R;
   output Q,QN;
   assign #1 Q=~(R|QN);
   assign #1 QN=~(S|Q);
endmodule
```



Simbolo →



Transistor necessari (8 transistor=4x2NOR)

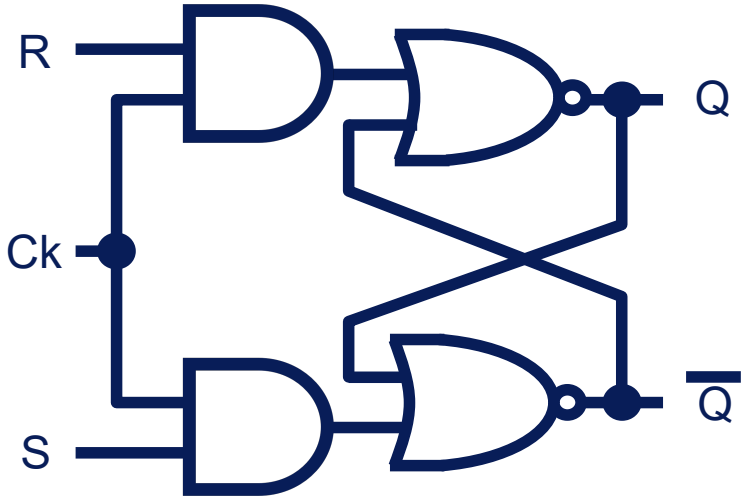
S-R cloccato

Il Latch SR è sempre sensibile ad ogni variazione di S e di R: per fare in modo che S e R vengano ascoltati solo in determinati istanti si può usare un segnale di abilitazione Ck

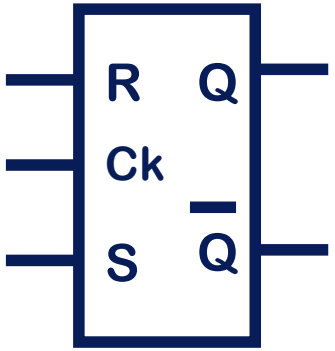
L'SR-cloccato, sente S e R solo quando il segnale Ck (clock) vale 1

Ck	S	R	Q	/Q
1	0	0	Q	/Q
1	0	1	0	1
1	1	0	1	0
1	1	1	---	---
0	X	X	Q	/Q

Tabella di Verità



Schema logico



Simbolo

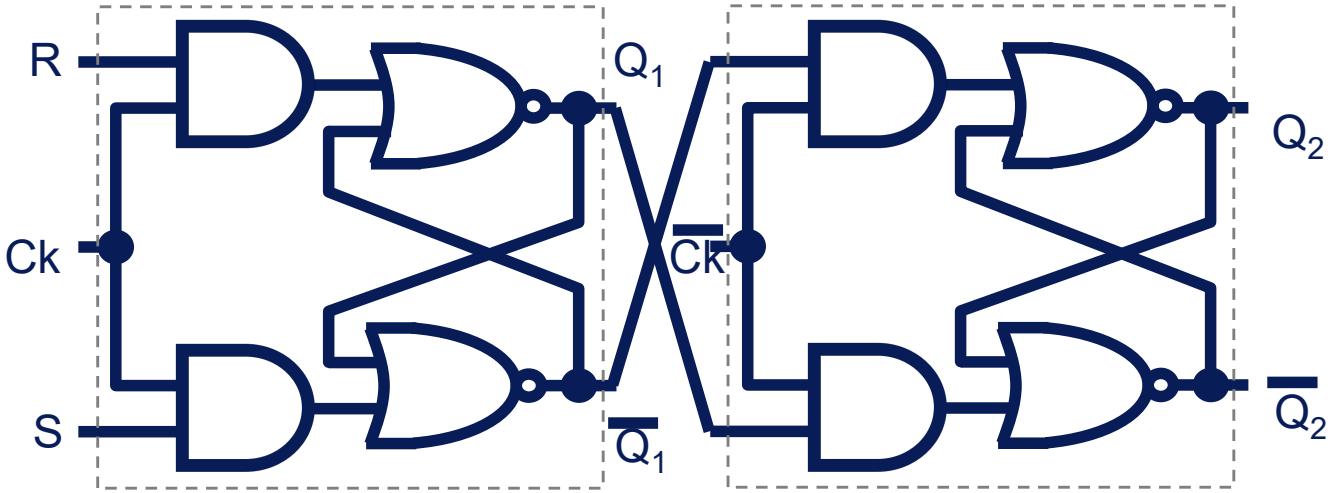
Layout simbolico: 20 transistor (=4x2NOR+6*2AND)

S-R edge triggered

In questo caso vogliamo sensibilità solo sul fronte in discesa

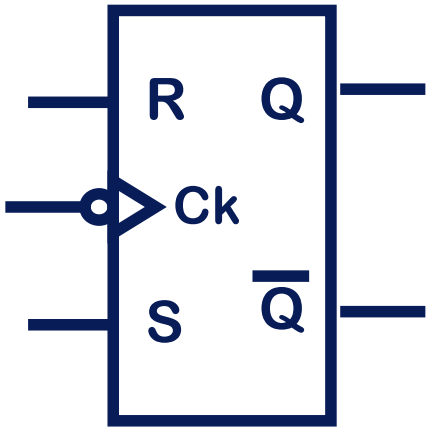
Ck	S	R	Q	/Q
↓	0	0	Q	/Q
↓	0	1	0	1
↓	1	0	1	0
↓	1	1	---	---
0	X	X	Q	/Q
1	X	X	Q	/Q
↑	X	X	Q	/Q

Tabella di Verità



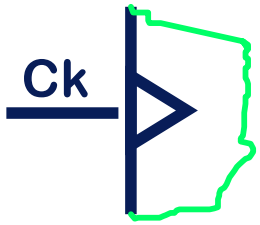
Schema logico

Layout simbolico: 42 transistor
4*NOR+4*(NAND+NOT)+ 2 per /Ck=4*4+4*(4+2)+2

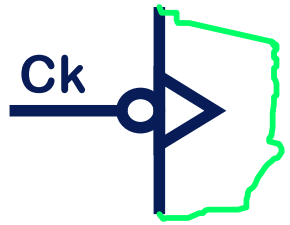


Simbolo

Può essere:



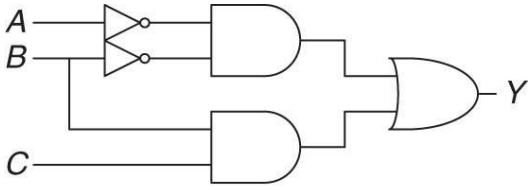
Sensibile sul fronte in salita

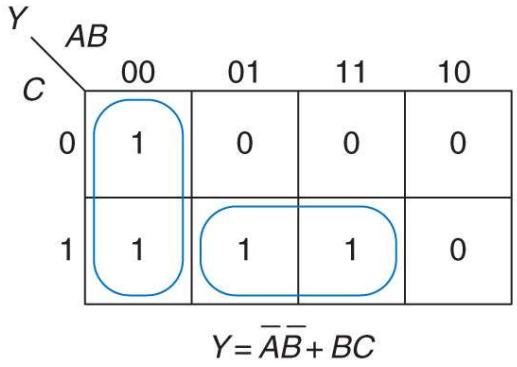


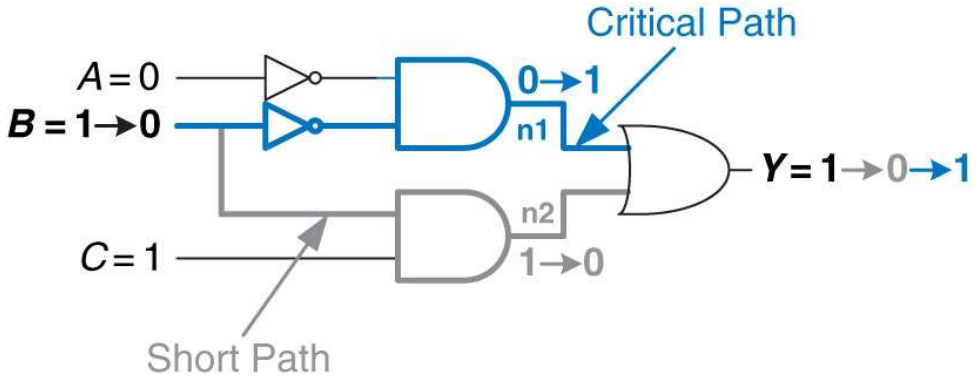
Sensibile sul fronte in discesa

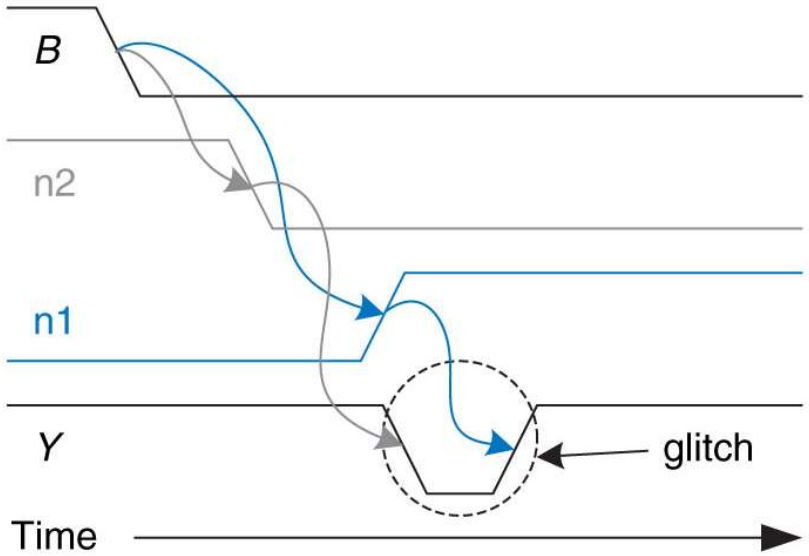
Alee (Glitches/Hazards)

- Può accadere che variando un ingresso, si creino variazioni multiple dell'uscita
- Esempio: rete con alee





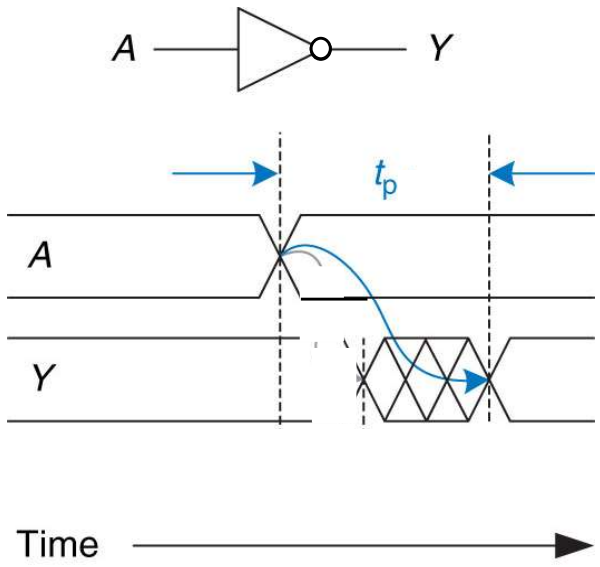




→ È importante essere in grado di analizzare il comportamento temporale di una rete logica (timing)

Timing

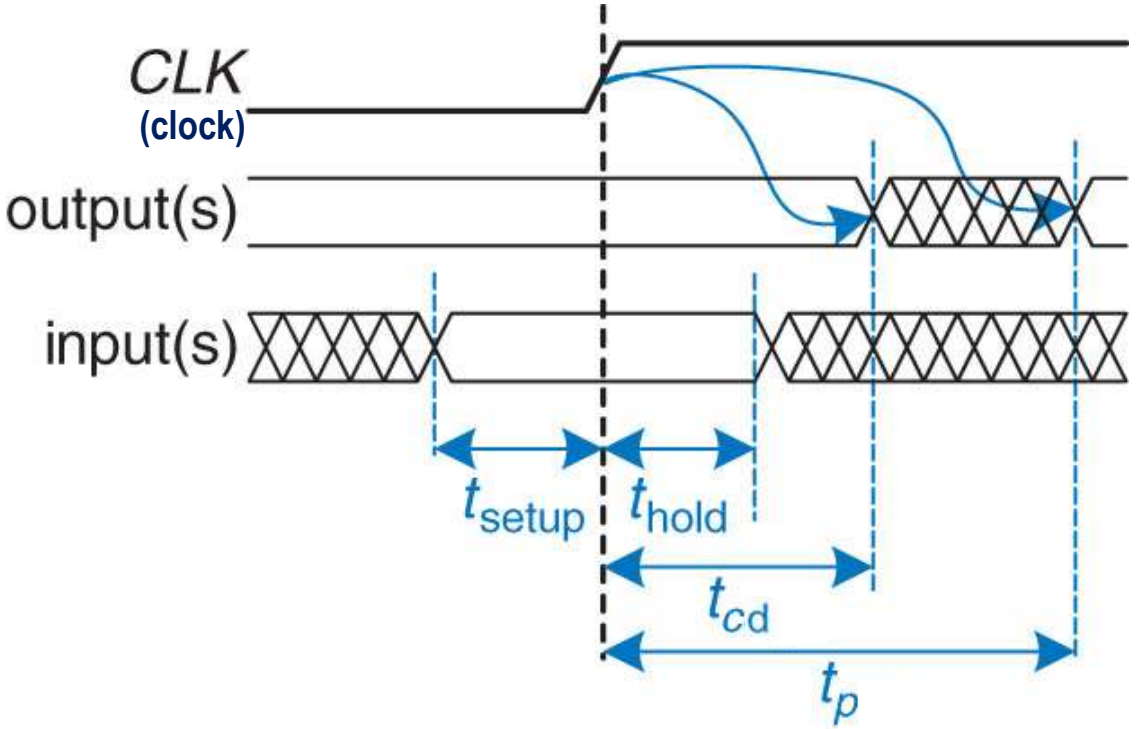
- Finora abbiamo analizzato solo l'aspetto funzionale della rete senza tenere conto dei tempi di propagazione del segnale fra ingresso e uscita (behavior)
- Abbiamo però visto come si possa calcolare il tempo di propagazione per un semplice inverter CMOS



- Il tempo di propagazione t_p è il tempo necessario affinché l'uscita sia stabile dal momento in cui ho variato l'ingresso
- Si definisce invece tempo di contaminazione t_{cd} il tempo che intercorre da quando ho variato l'ingresso a quando inizia a variare l'uscita

tsetup e thold

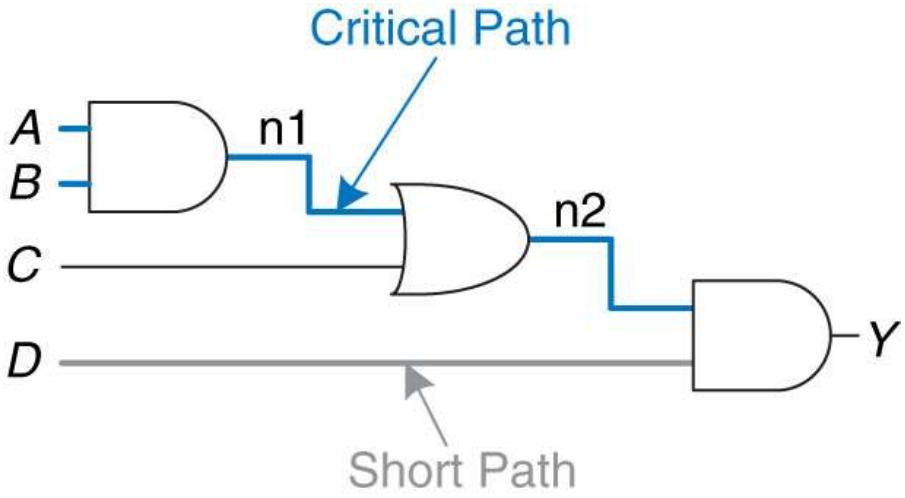
- I segnali di ingresso SR devono essere stabili durante un fronte del clock (vanno preparati un po' prima → tsetup)
- Dopo il fronte si possono tranquillamente variare i segnali (dopo aver atteso thold)



- Il tempo di propagazione tp è il tempo necessario affinché l'uscita sia stabile dal momento in cui ho variato l'ingresso
- Si definisce invece tempo di contaminazione tcd il tempo che intercorre da quando ho variato l'ingresso a quando inizia a variare l'uscita

Critical path

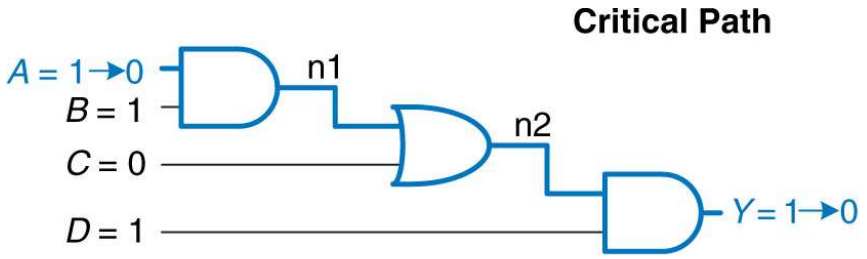
- Oltre ai ritardi dovuti alla singola porta logica, quando si interconnettono più porte in cascata può accadere che determinati segnali giungano alle varie porte intermedie con ritardi diversi. Esempio:

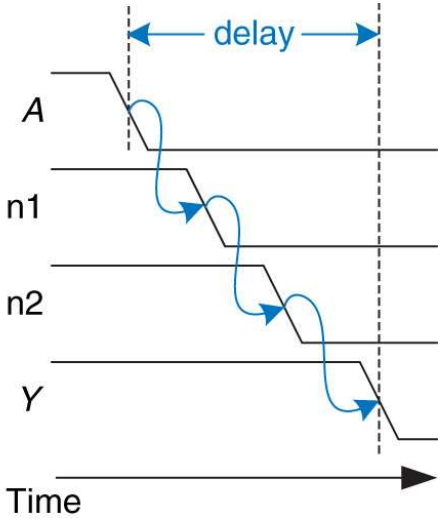


- In particolare il cammino più lungo fra un ingresso e l'uscita è chiamato cammino critico (critical path)

Calcolo dei tempi

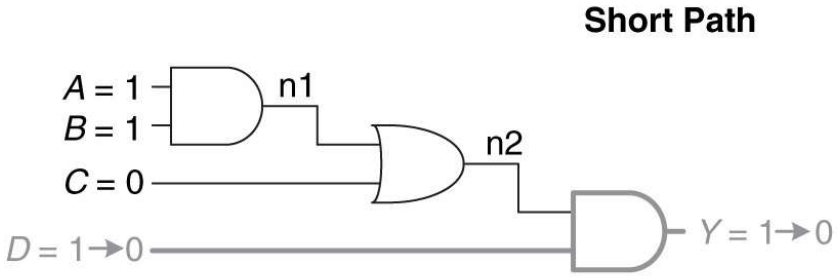
- Cambiando l'ingresso A da 1 a 0

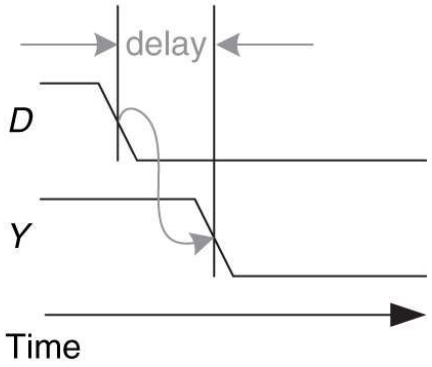




$$t_p = 2t_{p\_AND} + t_{p\_OR}$$

- Cambiando l'ingresso D da 1 a 0





$$t_{cd} = t_{cd\_AND}$$

Tempi di propagazione tipici

Gate	t_p (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (*A* to *Y*)	50
tristate (enable to *Y*)	35

D-Latch

- Il latch SR è problematico perché si comporta in modo impredicibile quando SR=11
- I segnali S e R definiscono sia *cosa* che *quando* le uscite vengono cambiate
  - Il progetto dei circuiti digitali è più semplice se si definisce separatamente cosa cambiare e quando cambiarlo
  - Questo è esattamente cosa fa il D-latch

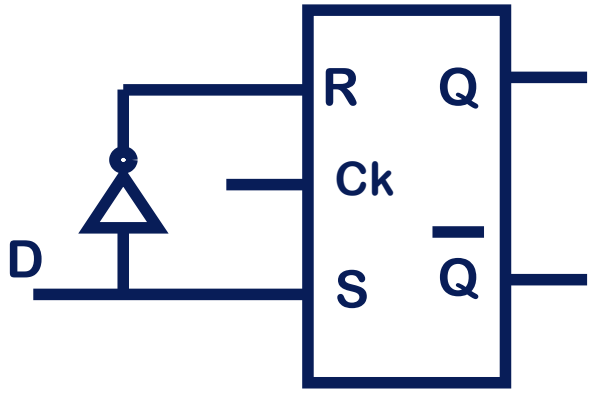
- D-latch
  - Il segnale D controlla il dato ovvero *cosa* deve essere l'uscita
  - Il segnale CLK controlla *quando* lo stato deve essere cambiato

D-Latch o D-Transparent

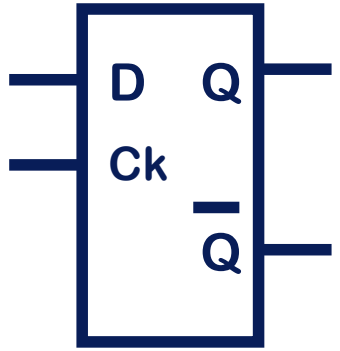
- Ha due ingressi (D e Ck):
  - D rappresenta il dato (indica COSA verrà memorizzato)
  - Ck è il clock (indica QUANDO memorizzare)

D	Ck	Q
X	0	Q
0	1	0
1	1	1

Tabella di Verità



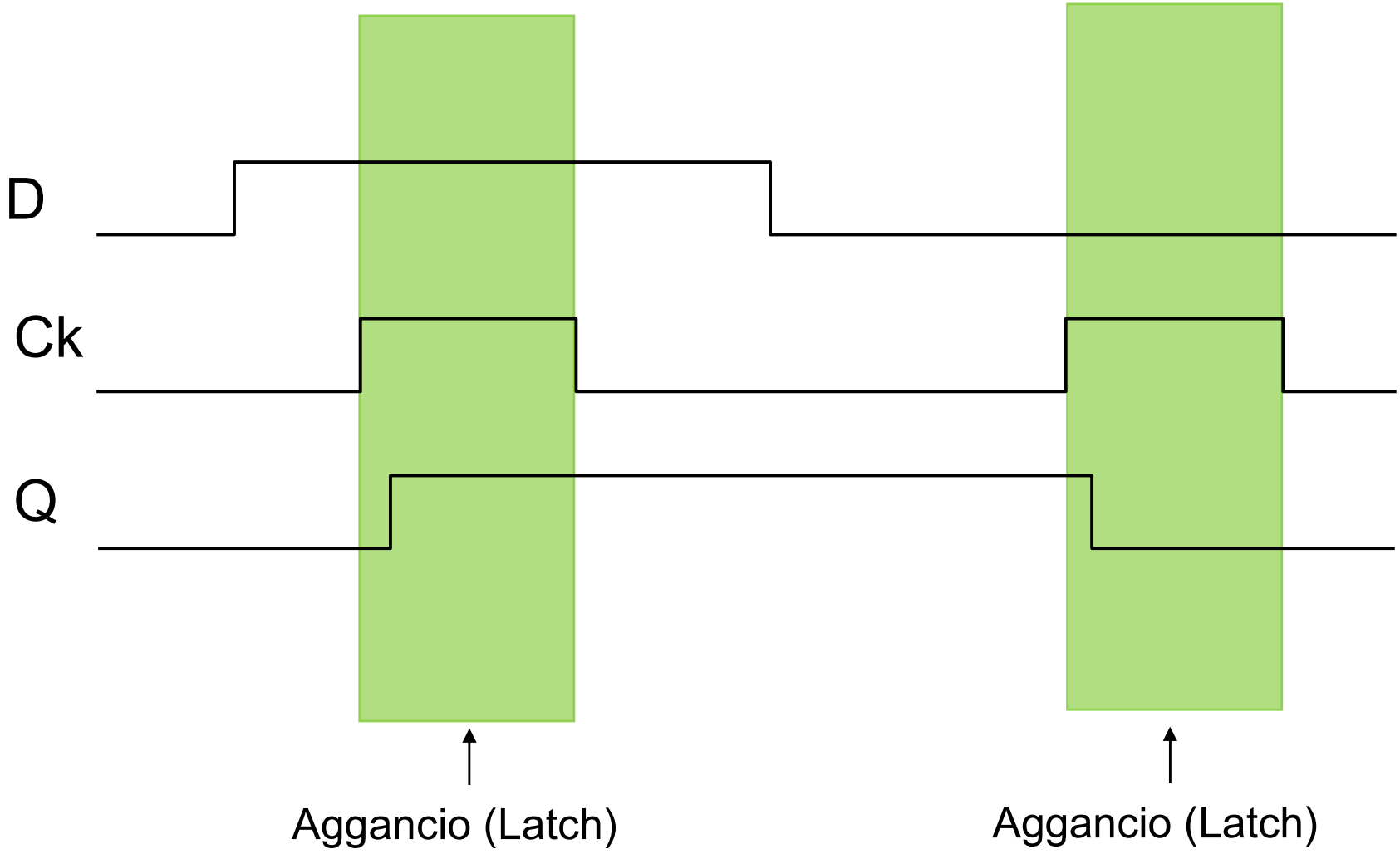
Schema logico



Simbolo del D-Latch

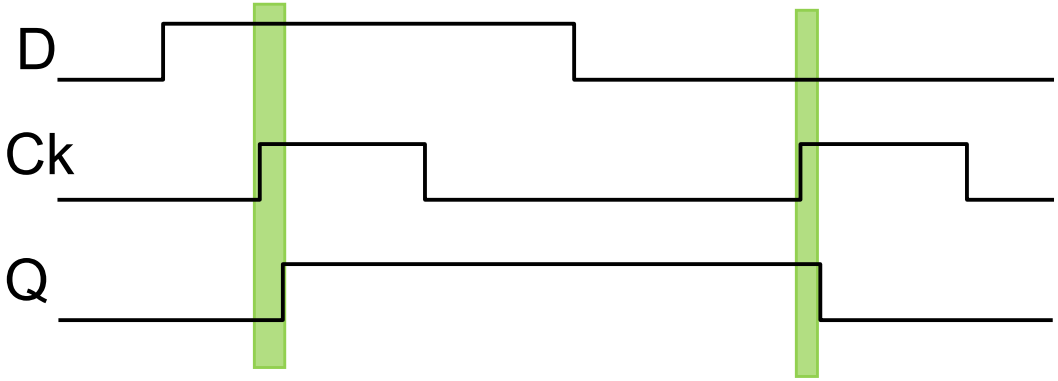
Layout simbolico: (22 transistor=20 SR-cloccato + 2 per NOT)

D-Latch o D-Transparent (esempio)



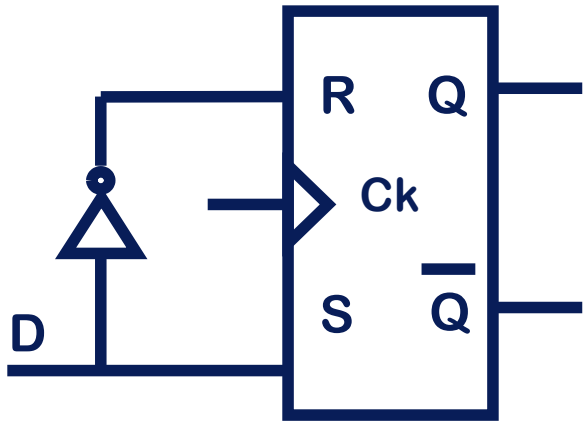
D-edge triggered

- L'uscita cambia solo sul fronte in salita del clock



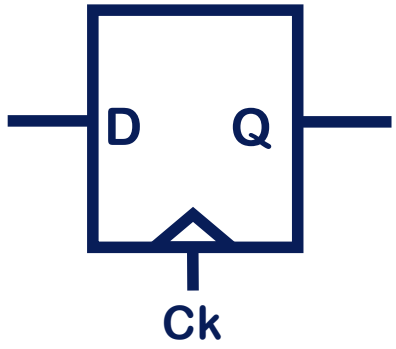
D	Ck	Q
X	1	Q
X	0	Q
X	⌐	Q
0	⌐	0
1	⌐	1

Tabella di Verità



Schema logico

Layout simbolico: 44 transistors
(42 per SR-ET + 2 per NOT)



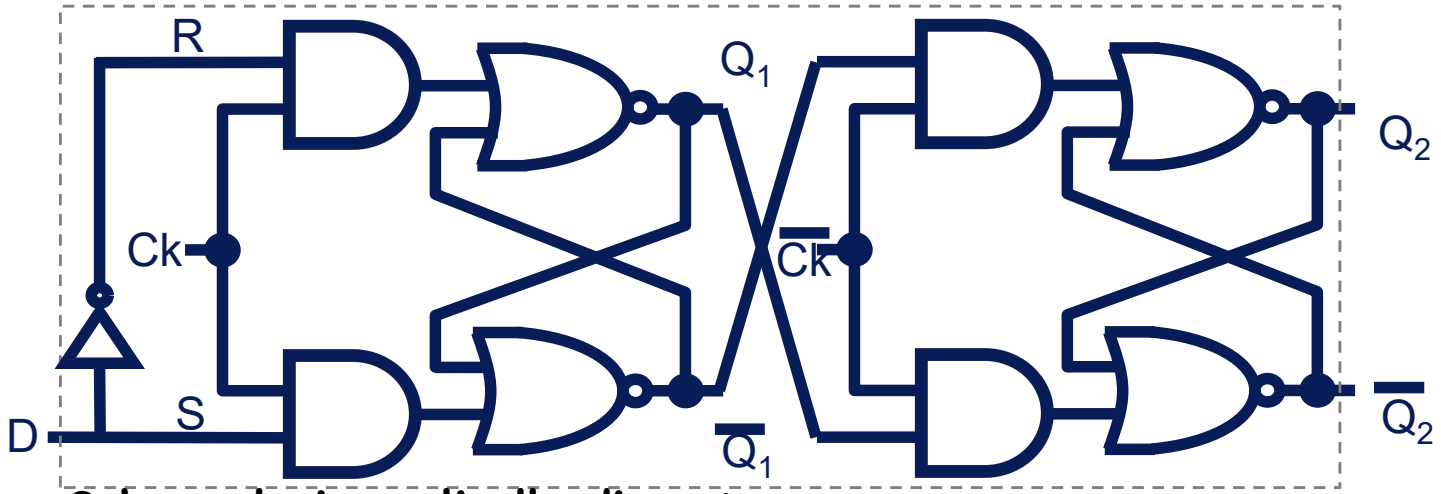
Simbolo del
D-Edge-Triggered

FLIP FLOP D *negative edge triggered*

In questo caso vogliamo sensibilità solo sul fronte in discesa

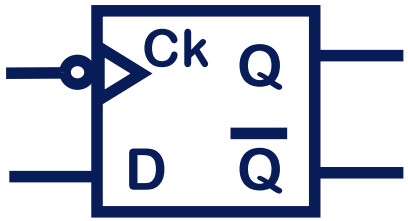
D	Ck	Q
X	1	Q
X	0	Q
X	⌐	Q
0	⌐	0
1	⌐	1

Tabella di Verità

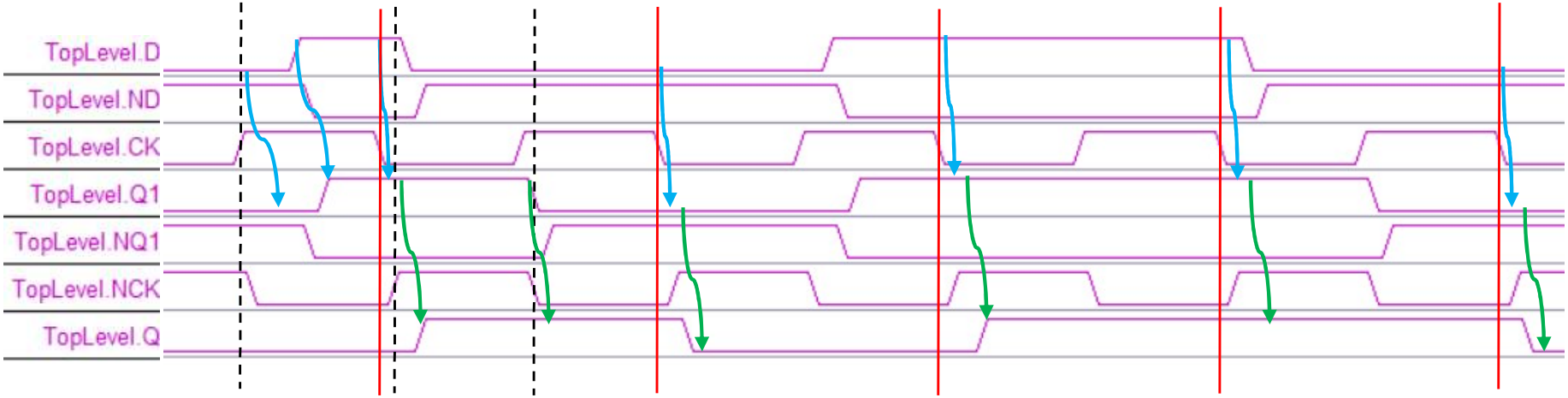


Schema logico a livello di porte

Layout simbolico: 44 transistor
=4*NOR+4*(NAND+NOT)+ 2 per /Ck
+ 2 per la not su D =4*4+4*(4+2)+2+2



Simbolo



Realizzazione in Verilog

```
`timescale 1ns/1ps
module TopLevel;
 reg reset_; initial begin reset_=0; #22 reset_=1; #500; $stop; end
 reg clock;  initial clock=0; always #10 clock<=(!clock);
 reg D;
 wire ND = dsrms.r;
 wire  CK = dsrms.mysrms.ck;
 wire  Q1 = dsrms.mysrms.q1;
 wire NQ1 = dsrms.mysrms.qbar1;
 wire NCK = dsrms.mysrms.nck;
 wire Q;
 initial begin
   wait(reset_==1); D<=0; #32 D<=1; #8; D<=0; #30 D<=1; #30; D<=0; #1000
   $finish;
 end
 DET_SRMS dsrms(D,clock,Q);
endmodule
```

```
module CSR(s,r,ck,q,qbar); // Clocked SR
  input  s, r, ck;
  output q, qbar;
  wire   s1, r1, s2, r2;
  assign s2 = q;
  assign r2 = qbar;
  assign #1 qbar = ~(s1|s2);
  assign #1 q    = ~(r1|r2);
  assign s1      = s & ck;
  assign r1      = r & ck;
endmodule
```

```
module SRMS(s,r,ck,q);  // SR Master-Slave
  input s, r, ck; output q;
  wire  nck, q1, qbar1, q2, qbar2;
  assign #1 nck = ~ck;
  assign q = q2;
  CSR master( s,    r, ck,  q1,qbar1);
  CSR slave (q1,qbar1,nck,  q2,qbar2);
endmodule
```

```
module DET_SRMS(d, ck, q);  // D-Edge Triggered via SR Master-Slave
  input d, ck; output q;
  wire  s, r;
  assign s = d;
  assign #1 r = ~d;
  SRMS mysrms(s,r,ck,q);
endmodule
```

Esercizio

- Descrivere un D-FF in Verilog

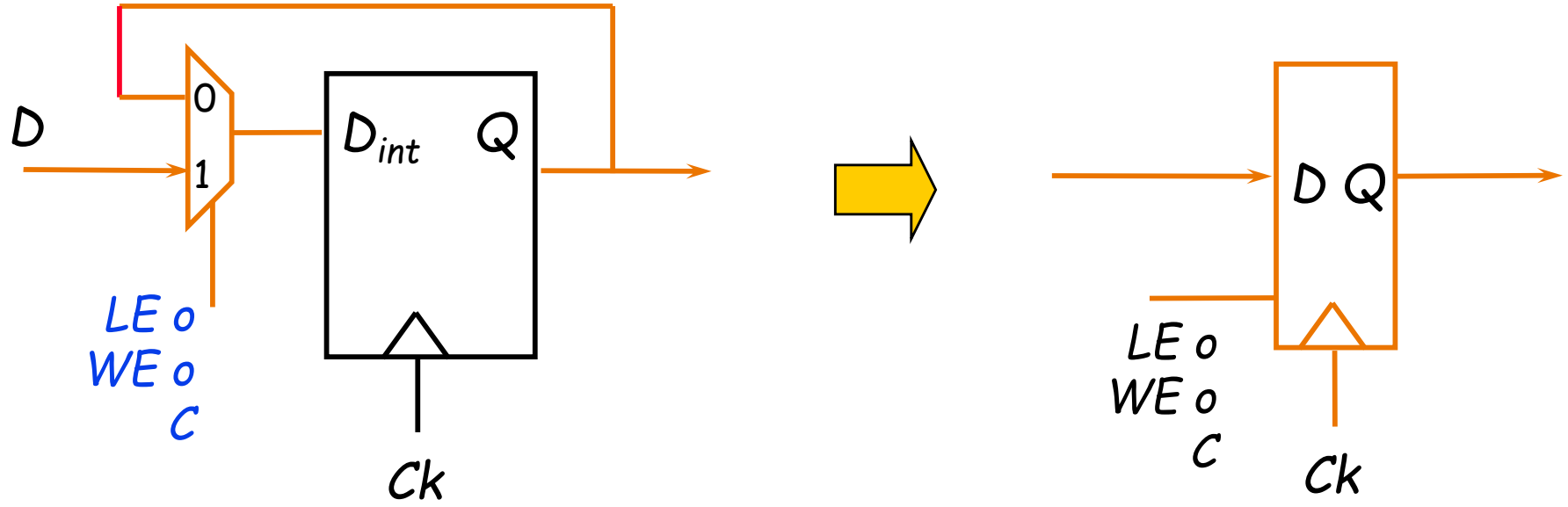
```
module DFF(clock,D,Q,Qbar);
  input clock,D;
  output Q,Qbar;
  reg STAR;
  assign Q=STAR, Qbar=~STAR; //Qbar è l'opposto di Q
  always @(posedge clock) //assegna sul fronte in salita del clock
  STAR<=D;
endmodule
```

Confronto Latch vs. Flip-flops

- Latch:
  l'uscita cambia non appena cambia l'ingresso
  E
  il clock ha valore alto
- Flip-flop:
  l'uscita cambia solo su un fronte del clock
  (metodologia edge-triggered)
- Elementi in comune:
  - l'uscita è pari al valore memorizzato nell'elemento
    (non è necessario fare azioni particolari per leggere il valore)
  - il cambiamento dell'uscita avviene rispetto a un segnale di clock
- Nota:
  nella logica sincrona si definisce l'istante in cui
  un valore è pronto per essere letto sugli ingressi del flip-flop
  - non si vuole leggere un dato D
    mentre la rete che genera D sta ancora "calcolandone" il valore

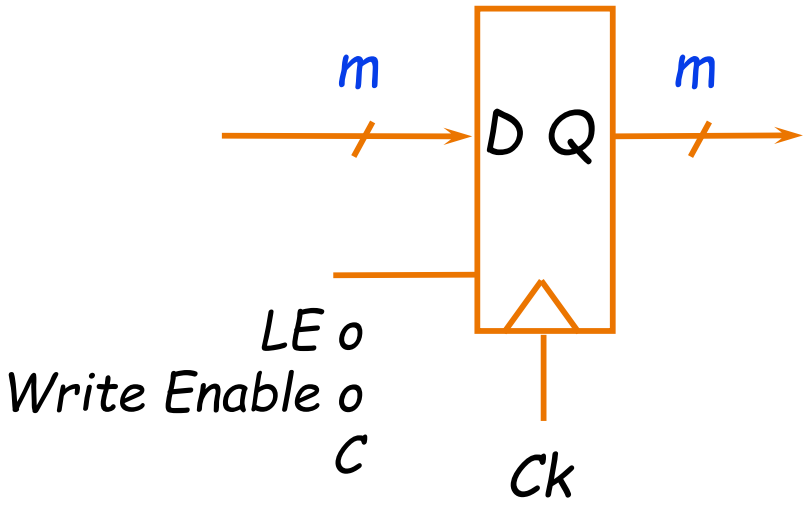
Cella di registro

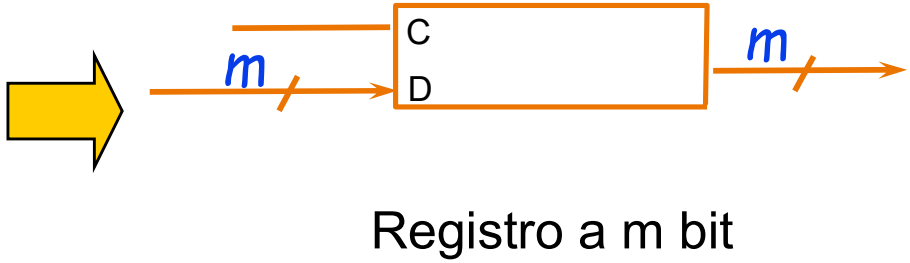
- Aggiungo un multiplexer il cui controllo si chiama
  - LE = Latch Enable o anche
    WE = Write Enable o anche
    C = Control



Simbolo della cella di registro da 1 bit

Registro a m bit

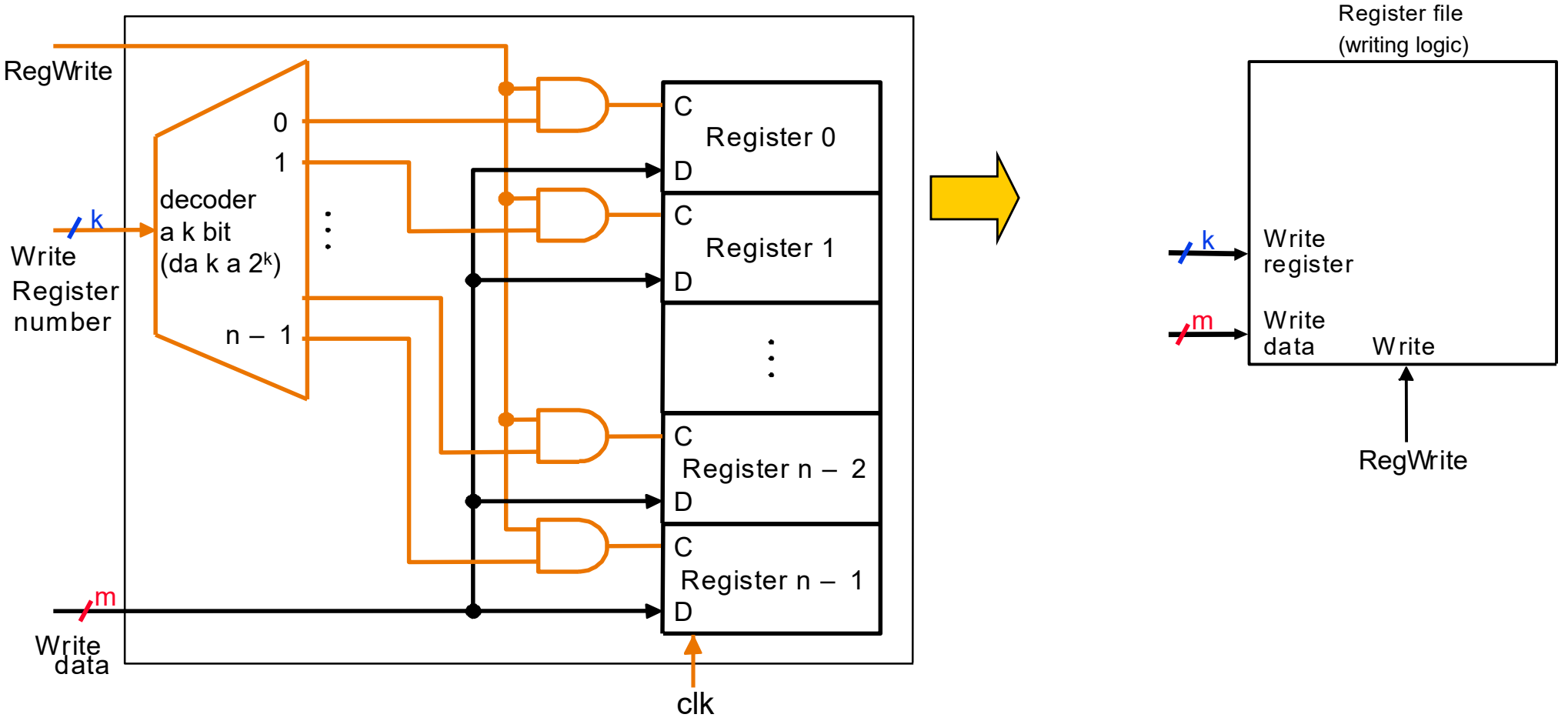




- Nell'ultima rappresentazione si intende implicitamente che sia presente un segnale Ck per il clock

- Register file:
  - Composto da n registri di questo tipo
  - Occorre della logica di scrittura
  - Occorre della logica di lettura

Register File: logica di scrittura (richiamo)



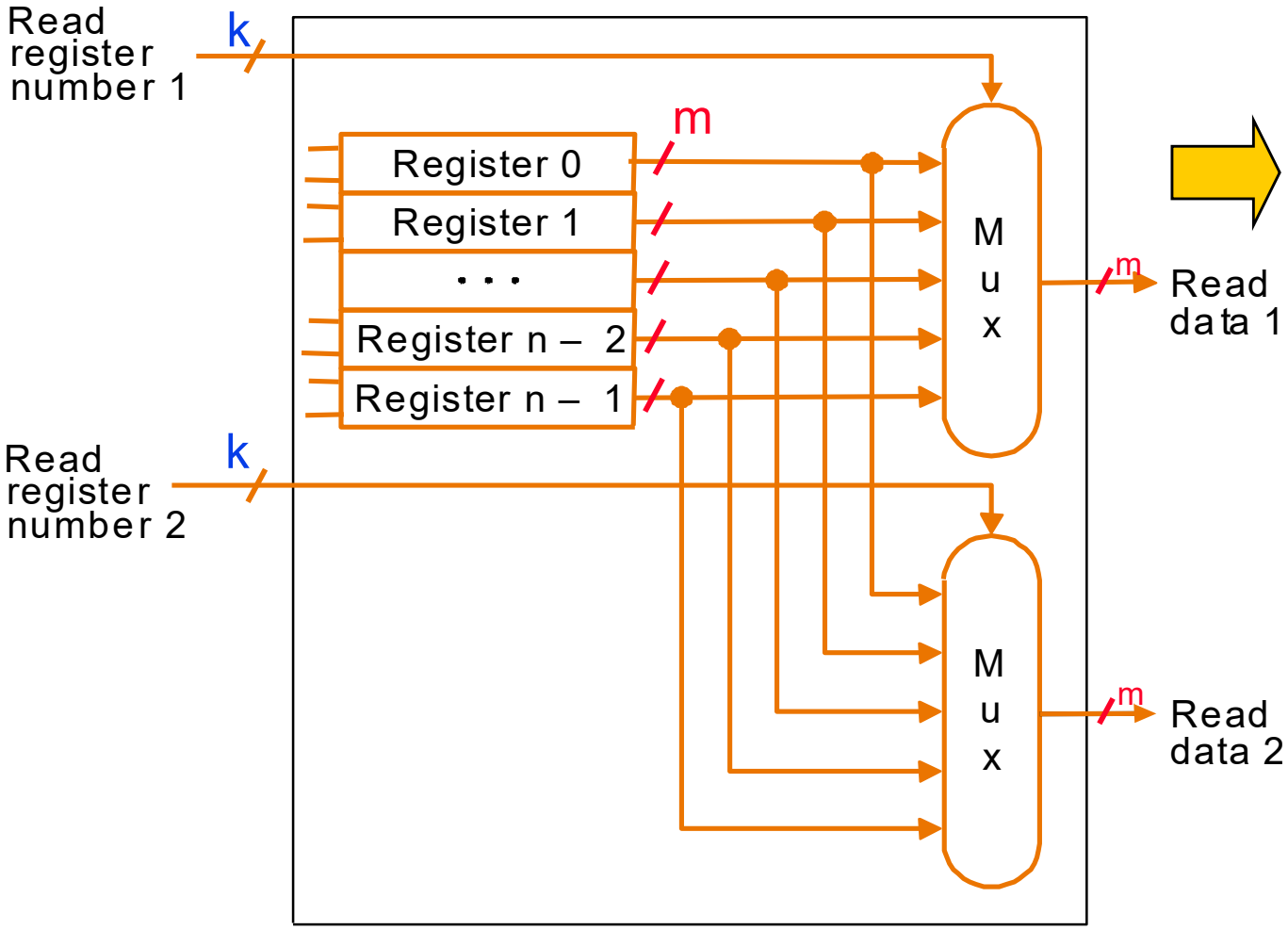
n = numero di registri disponibili (e.g. 32)

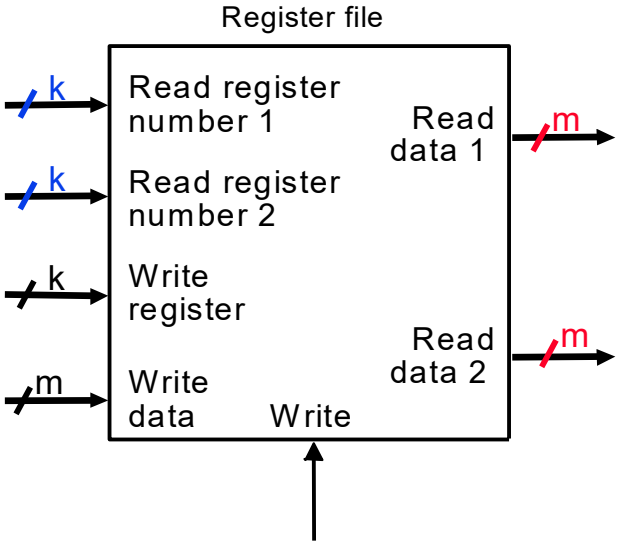
k = $\log_2(n)$ (e.g. 5)

m = larghezza dei registri (e.g. 64)

Register File: logica di lettura (richiamo)

- Si usano i flip-flop di tipo D





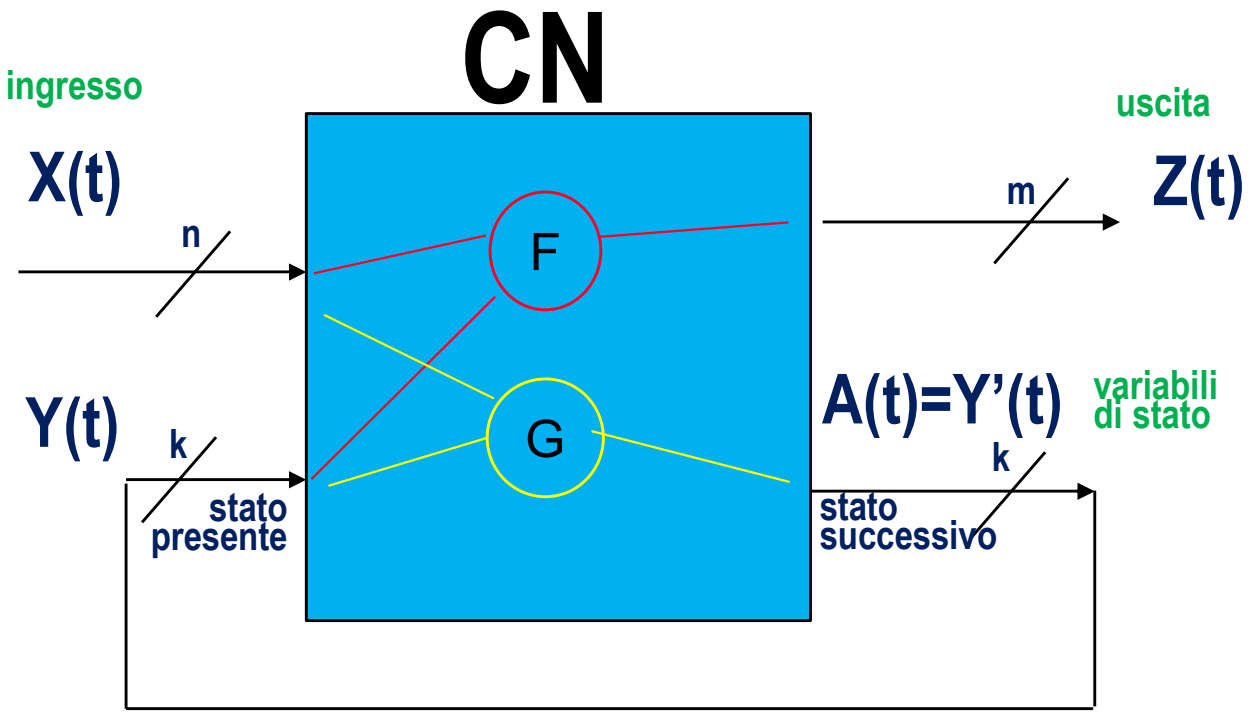
Lezione 6
Reti Logiche Sequenziali di Mealy e di Moore, Contatori e Addizionatori

Macchina di Mealy asincrona

- Le variabili d'uscita, in un determinato istante, sono funzione del valore degli ingressi e delle variabili di stato

$$Z(t)=F(\ X(t),\ Y(t)\ )$$

$$Y'(t)=G(\ X(t),\ Y(t)\ )$$



La rete CN è una rete combinatoria

Macchina di MEALY (sincronizzata)

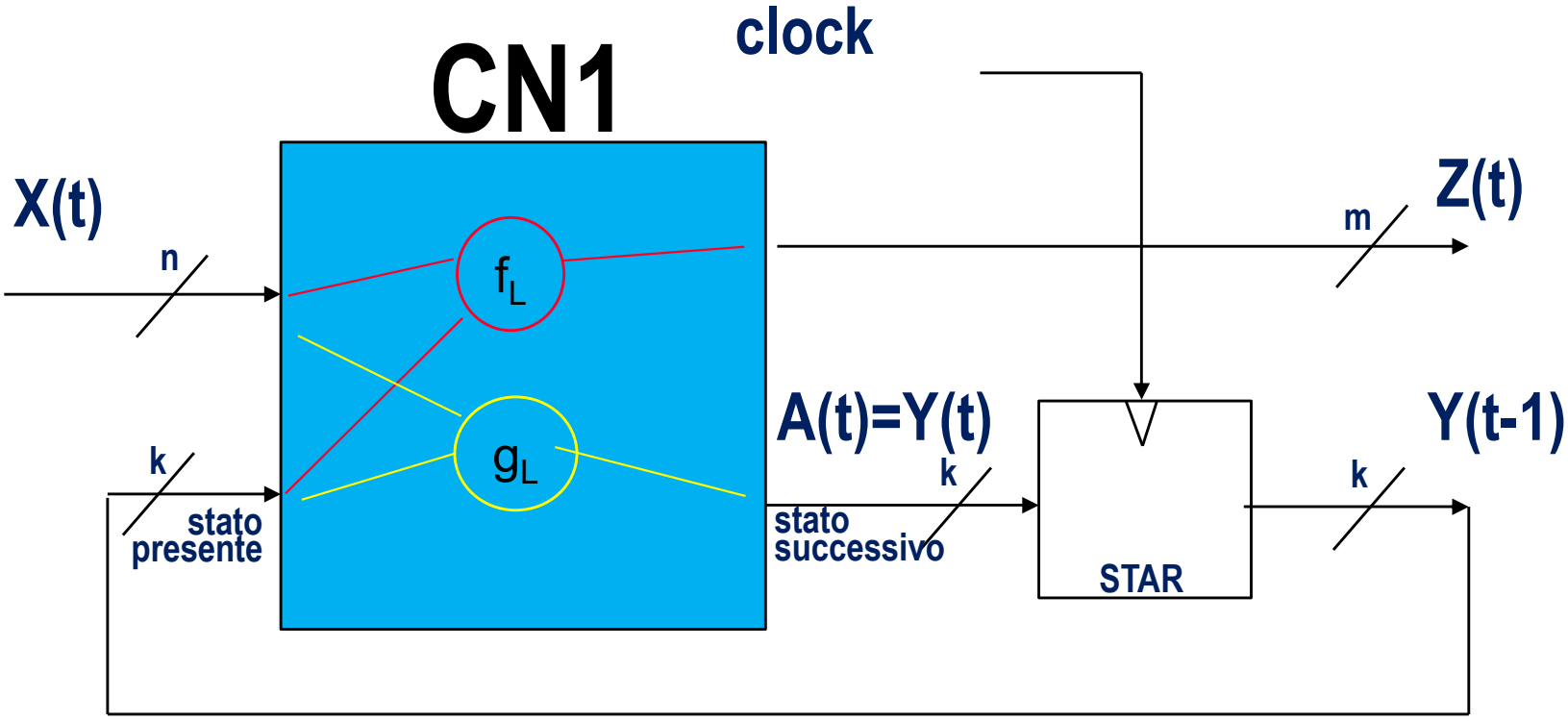
- Le uscite sono funzioni delle variabili di stato e degli ingressi

$$Z(t)=f_L(\ X(t),\ Y(t-1)\ )$$

$$Y(t)=g_L(\ X(t),\ Y(t-1)\ )$$

In MEALY: l'uscita (Z) dipende DIRETTAMENTE sia dallo stato (Y) che dagli ingressi (X)

Y(t-1) è lo stato presente
Y(t) è lo stato successivo



STAR (STATUS REGISTER) è costituito da Flip-Flop D

Macchina di MOORE (sincronizzata)

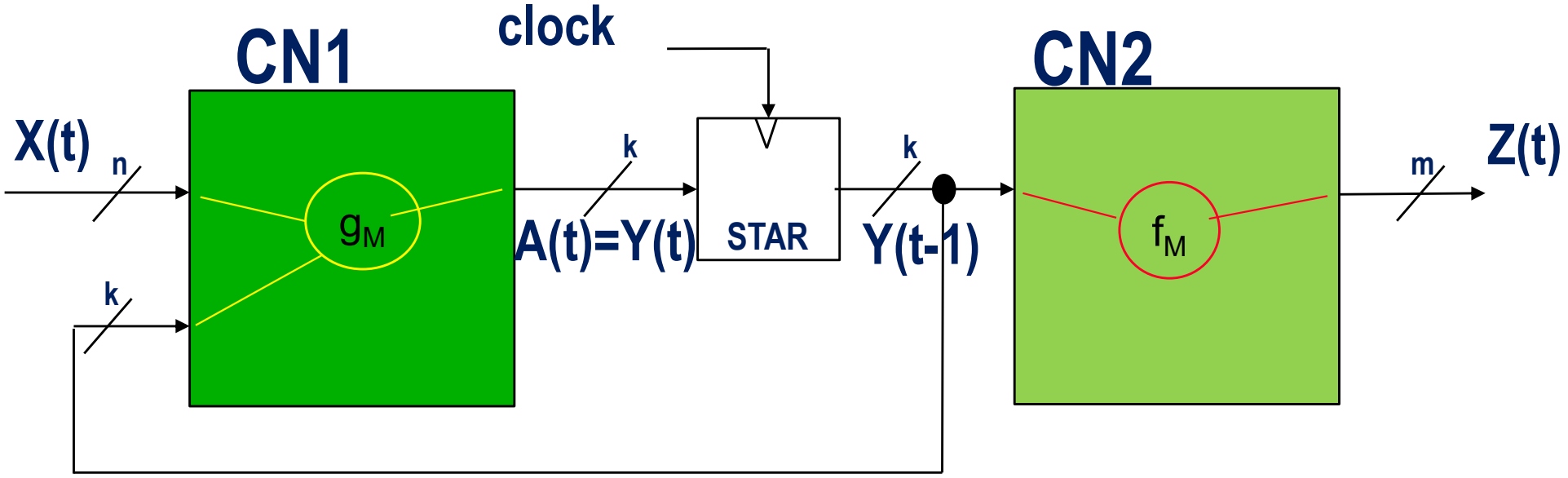
- Le variabili d'uscita, in un determinato istante, sono funzione delle sole variabili di stato

$$Z(t)=f_M(\ Y(t-1)\ )$$

$$Y(t)=g_M\ (\ X(t),\ Y(t-1)\ )$$

In MOORE: l'uscita (Z) dipende DIRETAMENTE solo dallo stato (Y)

Y(t-1) è lo stato presente
Y(t) è lo stato successivo



Macchina di Mealy Ritardata (sincronizzata)

- Le uscite sono funzioni delle variabili di stato e degli ingressi, ma risultano sincronizzate

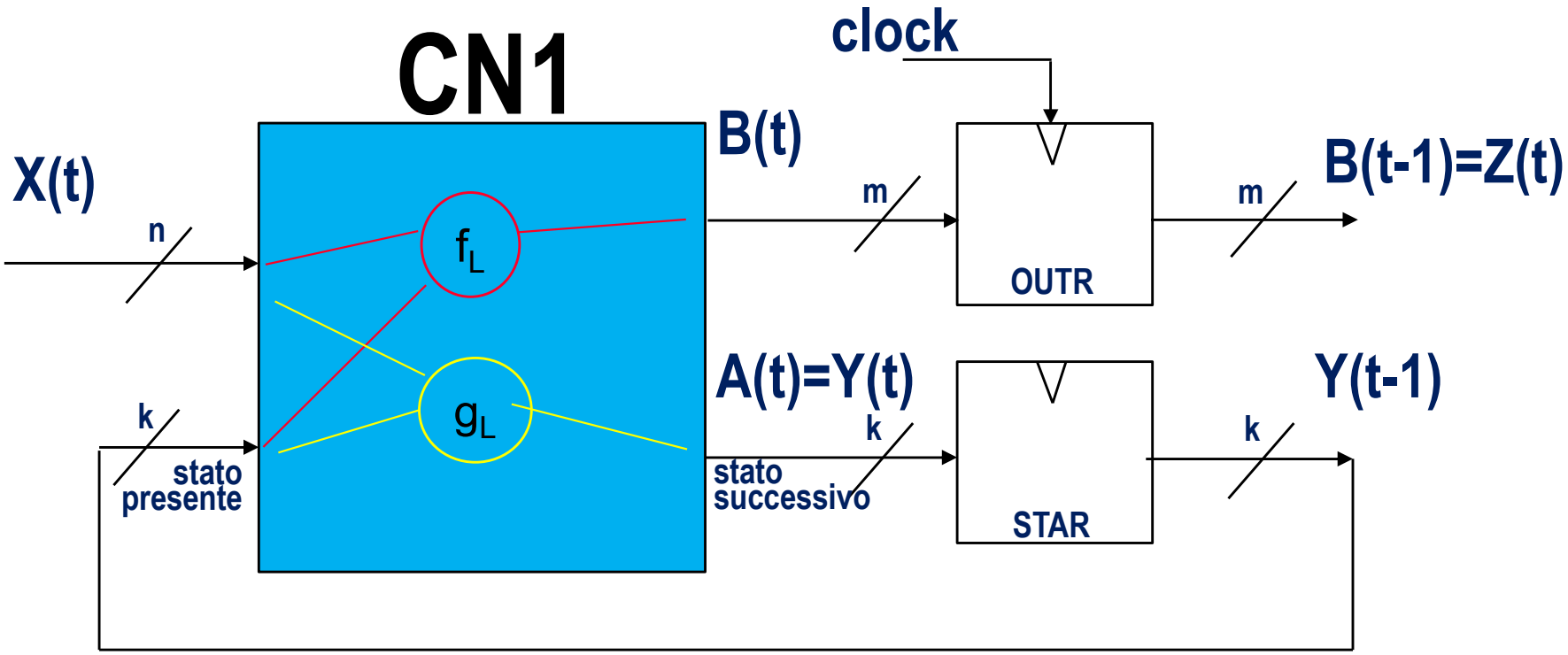
essendo B(t)=f(X(t),Y(t-1))...

$Z(t)=B(t-1)=f_L(X(t-1), Y(t-2))$

$Y(t)=g_L(X(t), Y(t-1))$

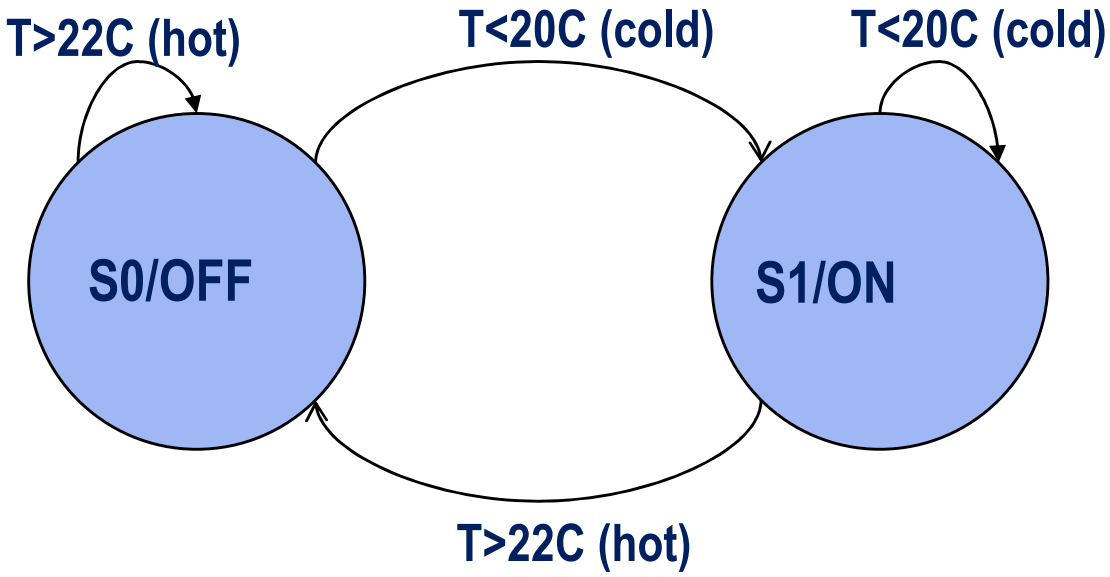
In MEALY: l'uscita (Z) dipende DIRETTAMENTE sia dallo stato (Y) che dagli ingressi (X)

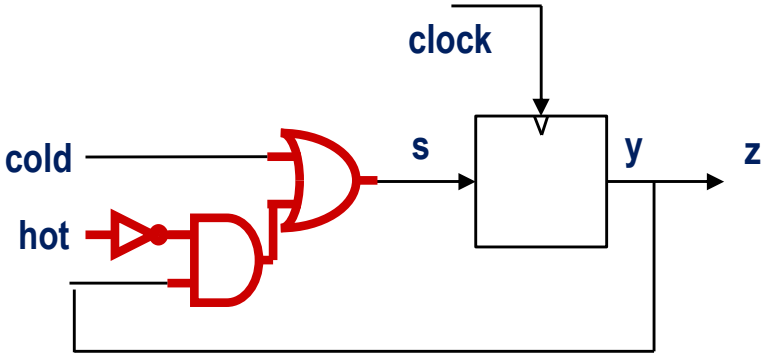
Y(t-1) è lo stato presente
Y(t) è lo stato successivo



Tecnica generale per implementare una FSM con Moore

- S0 → Riscaldamento Spento (z=OFF)
- S1→ Riscaldamento Acceso (z=ON)





In genere MOORE produce più stati, ma si può implementare usando solo LUT

STEP1) TABELLA DELLE TRANSIZIONI

STATO PRESENTE (Y) \ cold,hot (X)	00	01	11	10
S0	S0	S0	-	S1
S1	S1	S0	-	S1

STATO SUCCESSIVO (S)

STEP2) CODIFICA DEGLI STATI

Stato	S=Y
S0	0
S1	1

STEP3) SINTESI DELLA MAPPA PER S

y \ cold/hot (CN1)	00	01	11	10
0	0	0	-	1
1	1	0	-	1

s

$s = cold + \overline{hot} * y$

STEP4) SINTESI DELLE USCITE Z

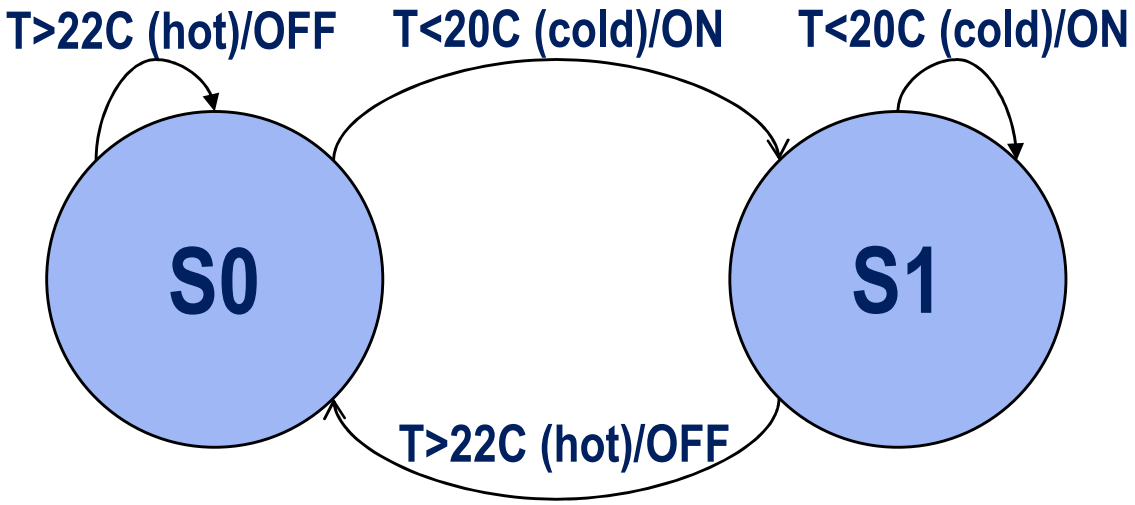
y \ CN2	0	1
0	0	1

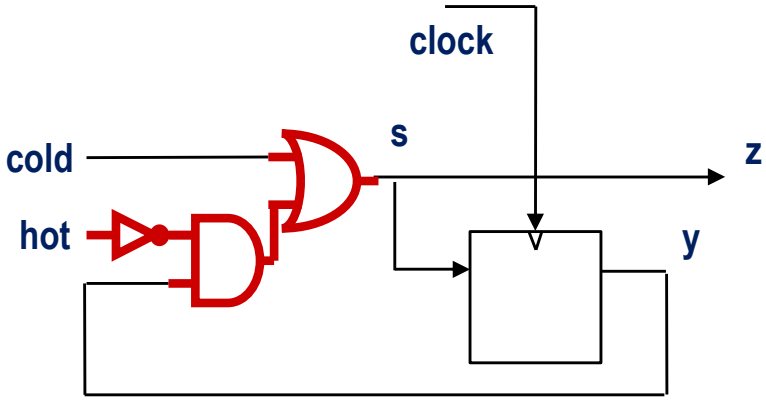
z

$z = y$

Tecnica generale per implementare una FSM con Mealy

- S0 → Riscaldamento Spento (z=OFF)
- S1→ Riscaldamento Acceso (z=ON)





In genere MEALY produce meno stati, ma può richiedere sia più FF che logica aggiuntiva

STEP1) TABELLA DELLE TRANSIZIONI

STATO PRESENTE (Y) \ cold,hot (X)	00	01	11	10
S0	S0/0	S0/0	- / -	S1/1
S1	S1/1	S0/0	- / -	S1/1

STATO SUCCESSIVO/USCITA (S/Z)

STEP2) CODIFICA DEGLI STATI

Stato	S=Y
S0	0
S1	1

STEP3) SINTESI DELLA MAPPA PER S

y \ cold/hot (CN1)	00	01	11	10
0	0	0	-	1
1	1	0	-	1

s

$s = cold + \overline{hot} * y$

STEP4) SINTESI DELLE USCITE Z (in generale diverso da S)

y \ cold/hot (CN2)	00	01	11	10
0	0	0	-	1
1	1	0	-	1

z

$z = cold + \overline{hot} * y$

=s (in questo caso)

Template generale per Moore in Verilog

```
module Moore(z,x,clock,reset_);
  input           clock, reset_;
  input  [N-1:0]   x;
  output [M-1:0]   z;
  reg    [K-1:0]   STAR;
  parameter S0=codifica0, …, SKmeno1=codificaKmeno1;
  always @(reset_==0) #1 STAR<=stato_iniziale;

  assign z=(STAR==S0)?codificaf0:
           (STAR==S1)?codificaf1:
            …
           /*(STAR==SKmeno1)*/ codificafKmeno1;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=g0(x);
      S1: STAR<=g1(x);
       …
      SKmeno1: STAR<=gKmeno1(x);
    endcase
endmodule
```

$f_M(Y)$

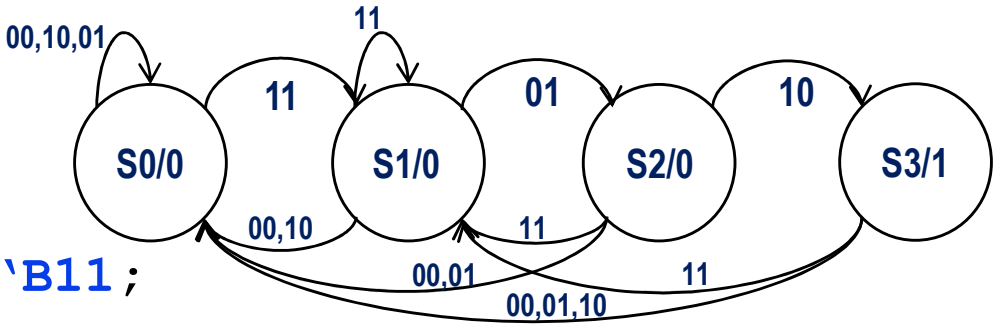
$g_M(X,Y)$

NOTA: Y non è altro che STAR

In gM la dipendenza dalla Y è attraverso il «casex(STAR)»

Esempio - riconoscitore di sequenza 11,01,10 con Moore

```
module ricseq110110moore(z,x,clock,reset_);
  input           clock, reset_;
  input  [1:0]     x;
  output           z;
  reg [1:0]        STAR;
  parameter S0='B00,S1='B01,S2='B10,S3='B11;
  always @(reset_==0) #1 STAR<=S0;
  assign z=(STAR==S3)?1:0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x=='B11)?S1:S0;
      S1: STAR<=(x=='B01)?S2:(x=='B11)?S1:S0;
      S2: STAR<=(x=='B10)?S3:(x=='B11)?S1:S0;
      S3: STAR<=(x==`B11)?S1:S0;
  endcase
endmodule
```



Template generale per Mealy in Verilog

```
module Mealy(z,x,clock,reset_);
  input           clock, reset_;
  input  [N-1:0]   x;
  output [M-1:0]   z;
  reg    [K-1:0]   STAR;
  parameter  S0=codifica0, …, SKmeno1=codificaKmeno1;
  always @(reset_==0) #1 STAR<=stato_iniziale;

  assign z=(STAR==S0)?f0(x):
           (STAR==S1)?f1(x):
            …
           /*(STAR==SKmeno1)*/ fKmeno1(x);
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=g0(x);
      S1: STAR<=g1(x);
       …
      SKmeno1: STAR<=gKmeno1(x);
    endcase
endmodule
```

$f_L(X,Y)$

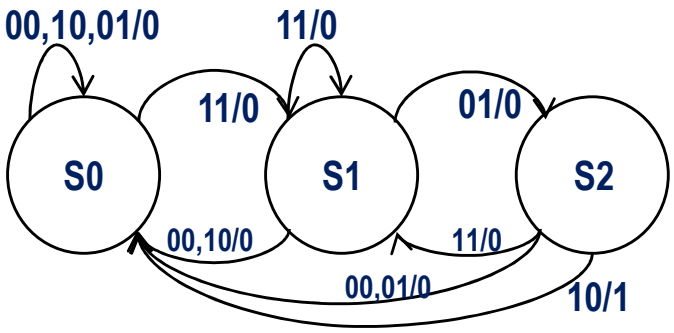
$g_L(X,Y)$

NOTA: Y non è altro che STAR

In gL la dipendenza dalla Y è attraverso il «casex(STAR)»

Esempio - riconoscitore di sequenza 11,01,10 con Mealy

```
module ricseq110110mealy(z,x,clock,reset_);
  input           clock, reset_;
  input  [1:0]     x;
  output           z;
  reg [1:0]        STAR;
  parameter  S0=`B00, S1=`B01, S2=`B10;
  always @(reset_==0) #1 STAR<=S0;

  assign z=((STAR==S2)&(x==`B10))?1:0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x==`B11)?S1:S0;
      S1: STAR<=(x==`B01)?S2:(x==`B11)?S1:S0;
      S2: STAR<=(x==`B11)?S1:S0;
  endcase
endmodule
```



Template generale per Mealy Ritardato in Verilog

```
module MealyRitardato(z,x,clock,reset_);
  input           clock, reset_;
  input  [N-1:0]   x;
  output [M-1:0]   z;
  reg    [K-1:0]   STAR;
  reg    [M-1:0]   OUTR;
  parameter  S0=codifica0, …, SKmeno1=codificaKmeno1;
  always @(reset_==0) #1 begin STAR<=…; OUTR<=…; end;

  assign z=OUTR;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin OUTR<=f0(x);          STAR<=g0(x); end
      S1: begin OUTR<=f1(x);          STAR<=g1(x); end
       …
      SKmeno1: begin OUTR<=fKmeno1(x); STAR<=gKmeno1(x); end
    endcase
endmodule
```

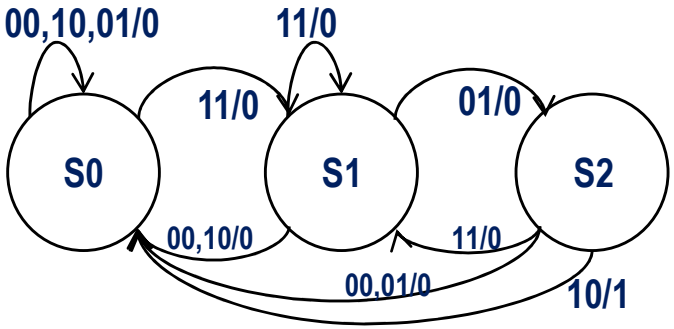
NOTA: Y non è altro che STAR

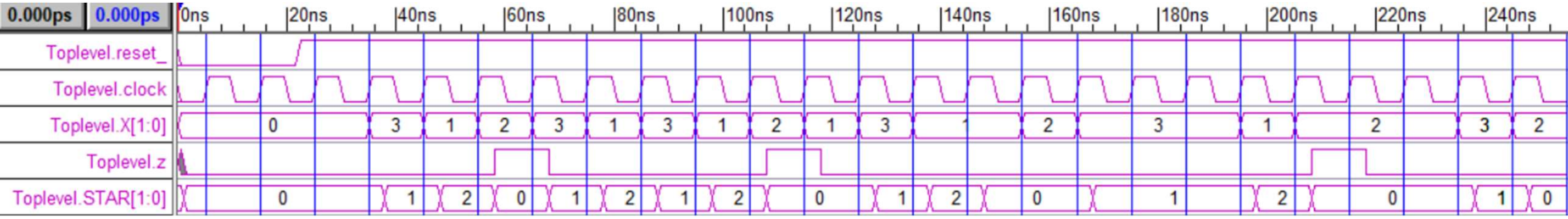
$f_L(X,Y)$

$g_L(X,Y)$

Esempio - riconoscitore di sequenza 11,01,10 con Mealy Rit

```
module ricseq110110mealyritardato(z,x,clock,reset_);
  input           clock, reset_;
  input  [1:0]     x;
  output           z;
  reg [1:0]        STAR;
  reg              OUTR;
  parameter  S0='B00, S1=`B01, S2=`B10;
  always @(reset_==0) #1 begin OUTR<=0; STAR<=S0; end
  assign     z=OUTR;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin OUTR<=0; STAR<=(x=='B11)?S1:S0; end
      S1: begin OUTR<=0; STAR<=(x=='B01)?S2:(x=='B11)?S1:S0; end
      S2: begin OUTR<=(x=='B10)?1:0; STAR<=(x=='B11)?S1:S0; end
  endcase
endmodule
```





Flip-Flop JK

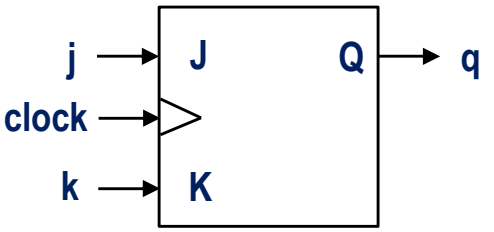
- Il FF-JK è tale che <u>in corrispondenza del fronte in salita del clock</u>:
  - per JK=10 l'uscita va a 1,
    per JK=01 l'uscita va a 0,
    per JK=00 il FF conserva (l'uscita resta inalterata)
    per JK=11 il FF commuta (l'uscita diventa il suo negato)

```
module FFJK(q,j,k,clock,reset_);
   input            clock, reset_;
   input             j,k;
   output            q;
   reg               STAR;
   parameter  S0=0, S1=1;
   assign q=(STAR==S0)?0:1;
   always @(reset_==0) #1 STAR<=0;
   always @(posedge clock) if (reset_==1) #3
     casex(STAR)
       S0: STAR<=(j==1)?S1:S0;
       S1: STAR<=(k==1)?S0:S1;
     endcase
endmodule
```

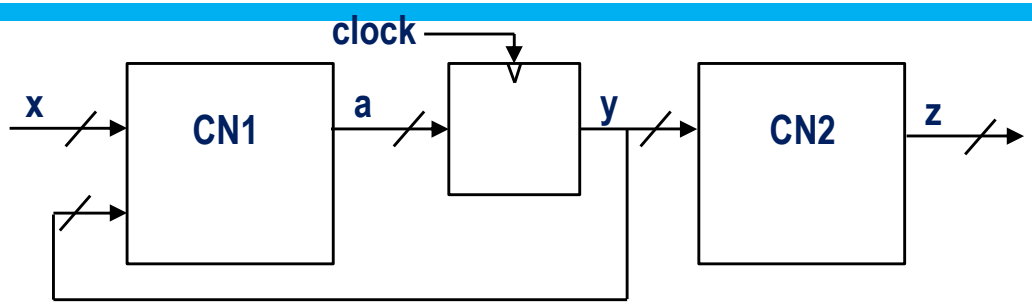
JK	Ck	Q
XX	1	Q
XX	0	Q
XX	⌐	Q
00	⌐	Q
01	⌐	0
10	⌐	1
11	⌐	/Q

Tabella di Verità

Simbolo del JK



Sintesi del FF-JK con Moore



- 2 stati → STAR ha 1 bit
- → a ha 1 bit, y ha 1 bit
  z ha 1 bit, (x ha 2 bit)

per JK=10 l'uscita va a 1,
per JK=01 l'uscita va a 0,
per JK=00 il FF conserva
per JK=11 il FF commuta

Tabella delle transizioni e delle uscite

jk / y	00	01	11	10	q
S0	S0	S0	S1	S1	0
S1	S1	S0	S0	S1	1

Stato	y0
S0	0
S1	1

CN1

y / jk	00	01	11	10
0	0	0	1	1
1	1	0	0	1

a

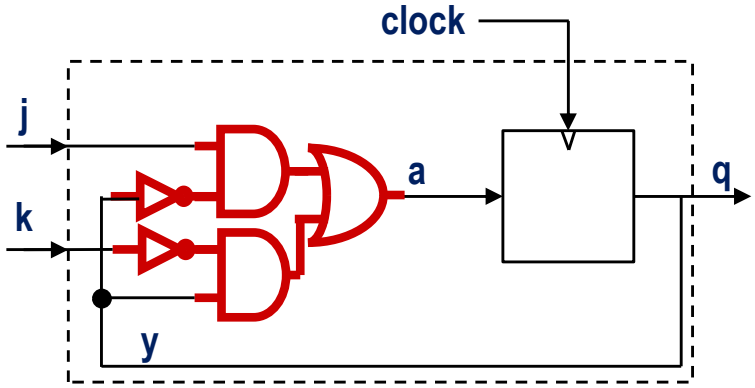
$a = j\bar{y} + \bar{k}y$

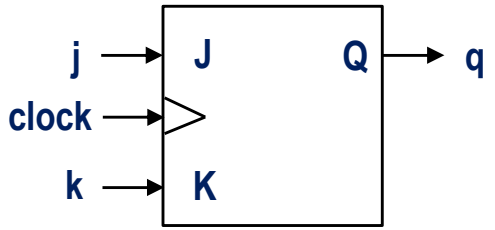
CN2

y	0	1
0	0	1
1	0	1

z

$z = y$



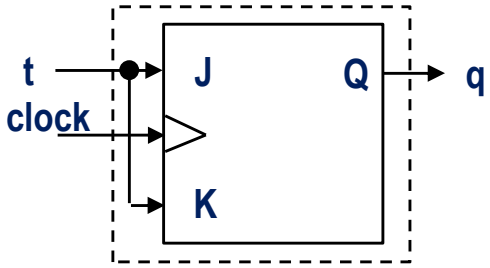


Simbolo del JK

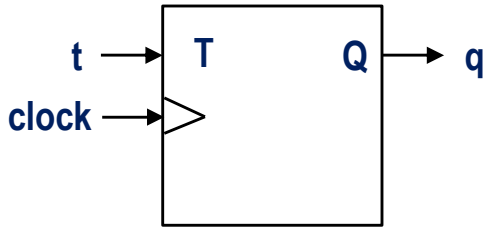
Layout simbolico: 60 transistors
(44 per D-ET + 3*NAND + 2*NOT)

Flip-Flop T (detto FF contatore)

- Il FF-T è tale che in corrispondenza del fronte in salita del clock:
- per T=0 il FF conserva
  (l'uscita rimane inalterata)
  per T=1 il FF commuta
  (l'uscita diventa il suo negato)

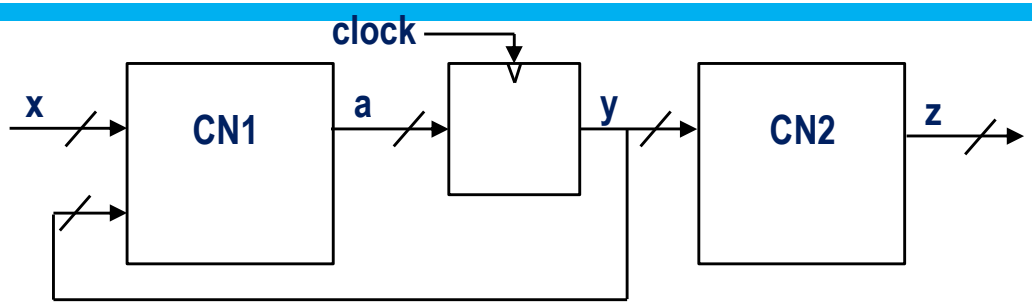



```
module FFT(q,t,clock,reset_);
  input           clock, reset_;
  input            t;
  output           q;
  reg              STAR;
  parameter  S0=0, S1=1;
  assign q=(STAR==S0)?0:1;
  always @(reset_==0) #1 STAR<=stato_iniziale;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(t==1)?S1:S0;
      S1: STAR<=(t==1)?S0:S1;
    endcase
endmodule
```



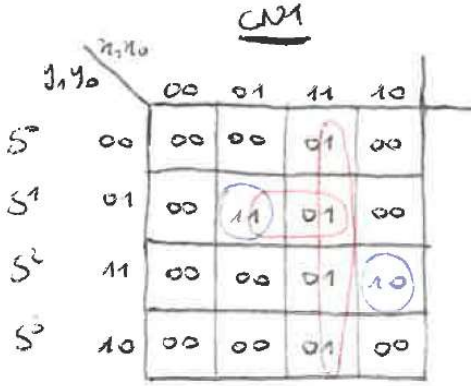
Abbiamo messo t al posto sia di j (prima riga – S0) che di k (seconda riga – S1)

Sintesi del riconoscitore di sequenza 11,01,10 con Moore



- 4 stati → STAR ha 2 bit
- → a ha 2 bit, y ha 2 bit
  z ha 1 bit, (x ha 2 bit)

Stato	y1 y0
S0	00
S1	01
S2	11
S3	10



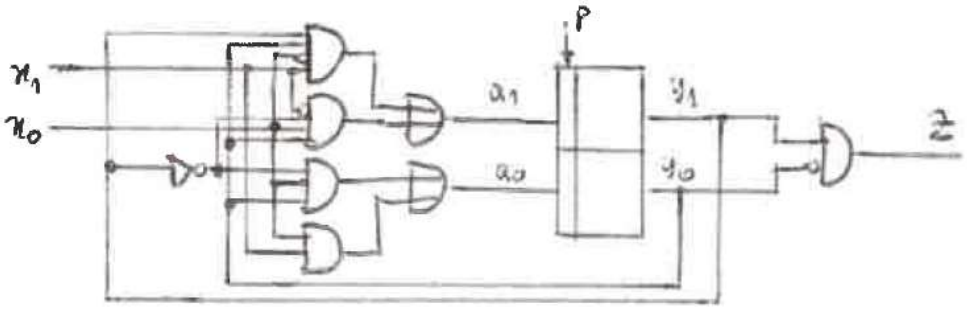
$$a_1 = \bar{x}_1 x_0 \bar{y}_1 y_0 + x_1 \bar{x}_0 y_1 y_0$$

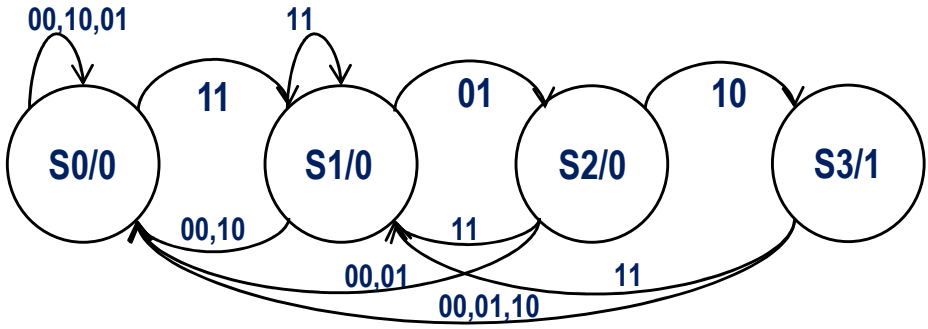
$$a_0 = x_1 x_0 + x_0 \bar{y}_1 y_0$$

CN2

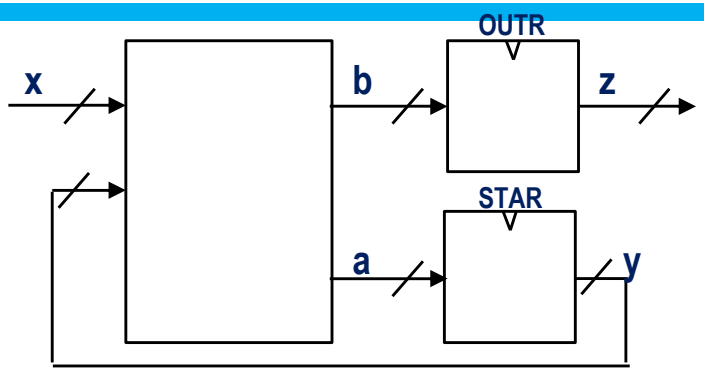
y_1 y_0	z
00	0
01	0
11	0
10	1

$z = y_1\bar{y}_0$





Sintesi del riconoscitore di sequenza 11,01,10 con Mealy Rit



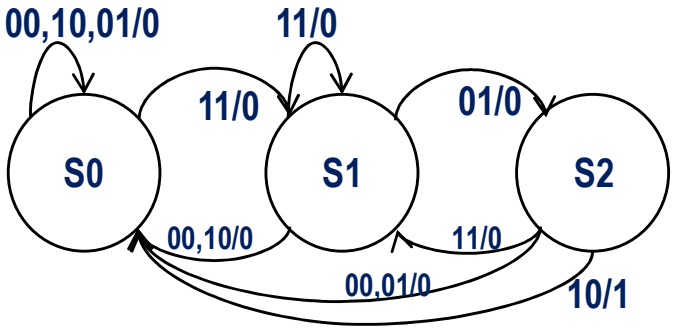


Tabella delle transizioni e delle uscite

x_1x_0 / Stato	00	01	11	10
S0	S0/0	S0/0	S1/0	S0/0
S1	S0/0	S2/0	S1/0	S0/0
S2	S0/0	S0/0	S1/0	S0/1

Stato	y1y0
S0	00
S1	11
S2	01

a_1a_0

y_1y_0 \ x_1x_0	00	01	11	10
00	00	00	11	00
01	00	00	11	00
11	00	01	11	00
10	--	--	--	--

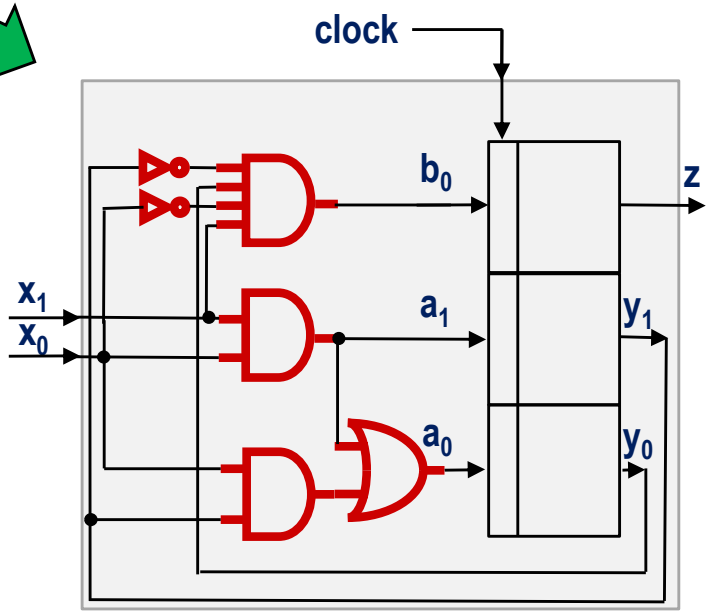
b_0

y_1y_0 \ x_1x_0	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	0	0
10	-	-	-	-

$a0 = x1\ x0 + x0\ y1$

$a1 = x1\ x0$

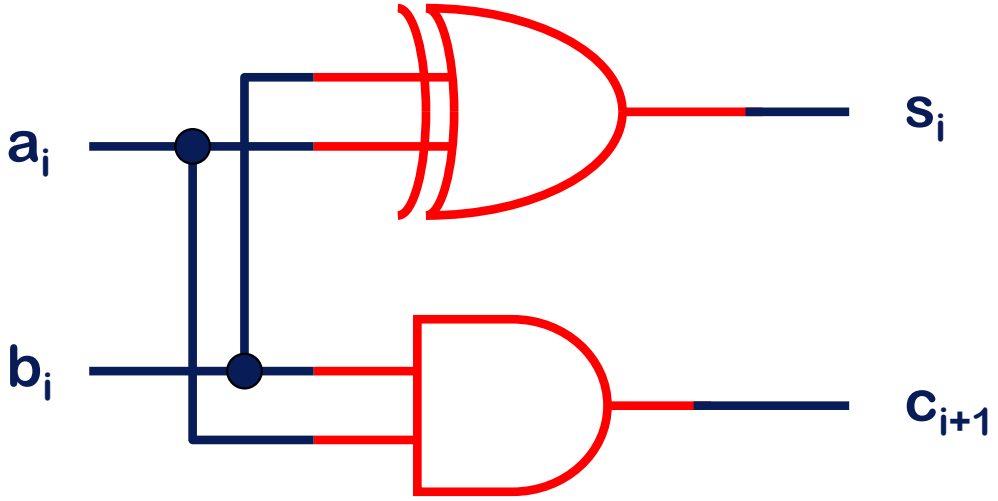
$b0 = x1\ \bar{x0}\ \bar{y1}\ y0$



Half Adder in VERILOG

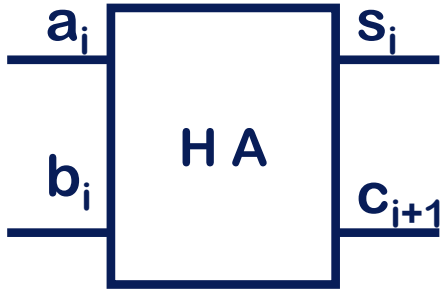
- Somma di due bit (senza riporto in ingresso)

a_i	b_i	s_i	c_{i+1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



$$s_i = a_i \oplus b_i$$

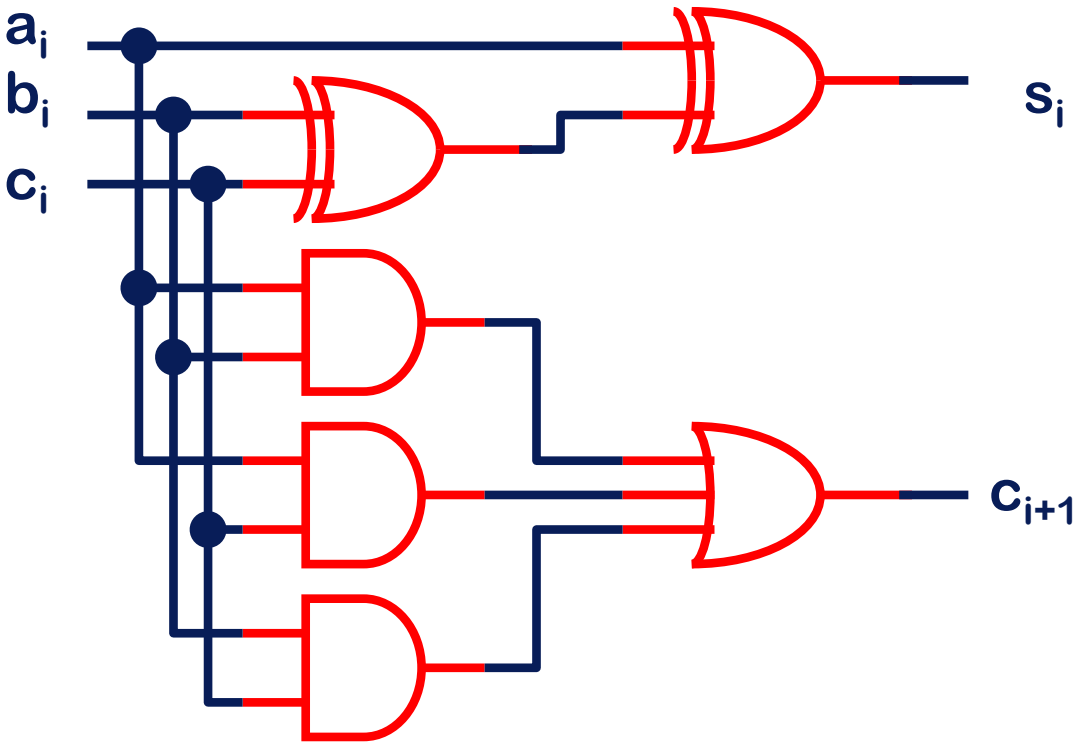
$$c_{i+1} = a_i \cdot b_i$$

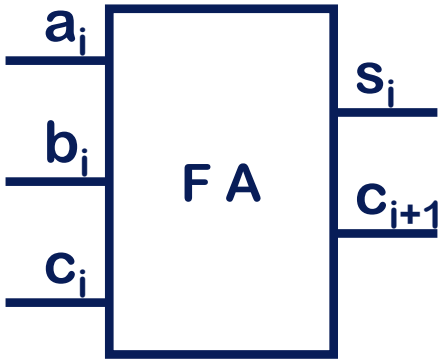


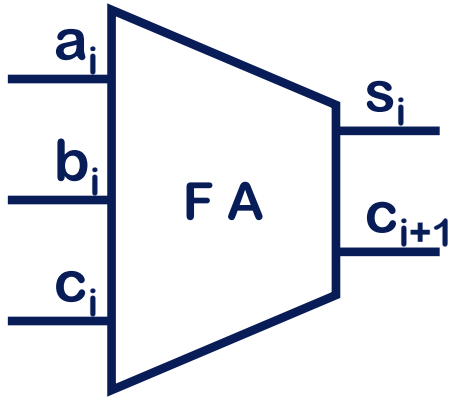

```
module half_adder(input a, input b, output s, output c);
        assign s = a ^ b;
        assign c = a & b;
endmodule
```

Full Adder in VERILOG

$$s_i = a_i \oplus b_i \oplus c_i = a_i \oplus (b_i \oplus c_i)$$
$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$

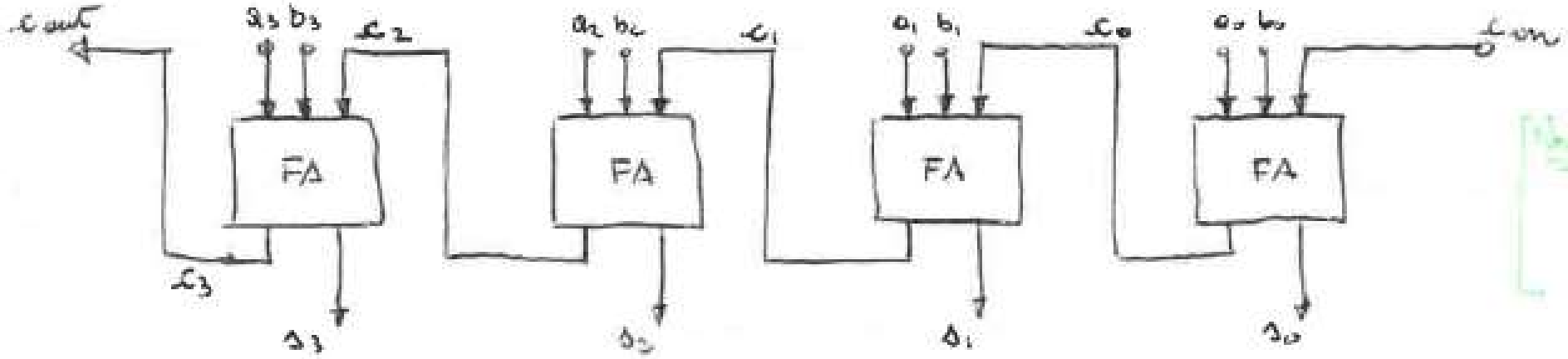







```
module full_adder(input a,input b,input c_in,  output s,output c);
        assign s = a ^ (b ^ c_in);
        assign c = (a & b) | (c_in & (a | b));
endmodule
```

Sommatore Parallelo con riporto seriale



$$\tau_{cout} = N \cdot \tau_C$$

$$\tau_{tot} = (N-1) \cdot \tau_C + \tau_{FA}$$

N=numero di bit del sommatore
τ_C = ritardo per generare il carry
τ_{FA} = ritardo del Full-Adder singolo
τ_{tot} = ritardo totale del sommatore

Sommatore Parallelo con riporto seriale in VERILOG

```
module rc_adder(input wire[N-1:0]a,input wire[N-1:0]b,input wire cin,
                output wire [N-1:0]s,output wire cout);

    parameter N = 32;  // bits
    wire [N-1:0] _c;

    assign cout = _c[N-1];
    full_adder add0(a[0],b[0],cin,  s[0],_c[0]);


    genvar i;
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_adder

            full_adder addN(a[i],b[i],_c[i-1],  s[i],_c[i]);

        end
    endgenerate
endmodule
```

"add0" rappresenta un'istanza di full_adder

Le variabili 'genvar' spariscono dopo l'elaborazione del codice (non diventano circuiti)

I nomi generati avranno il prefisso "gen_adder[i]."

"addN" rappresenta un'istanza di full_adder

Carry Look-Ahead Adder (Riporto Parallelo)

$$s_i = a_i \oplus b_i \oplus c_{i-1}$$

$$c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1} = a_i b_i + c_{i-1}(a_i + b_i) = a_i b_i + c_{i-1}(a_i \oplus b_i)$$

ai+bi si può sostituire col suo xor perché nell'espressione quando ai=1 e bi=1 anche ai*bi=1

$$\left.\begin{aligned} G_i &\triangleq a_i b_i \\ P_i &\triangleq a_i \oplus b_i \end{aligned}\right] \Rightarrow \quad \begin{aligned} s_i &= P_i \oplus c_{i-1} \\ c_i &= G_i + P_i \cdot c_{i-1} \end{aligned}$$

Gi=generate
Pi=propagate

$$c_0 = G_0 + P_0 \cdot c_{in}$$

$$c_1 = G_1 + P_1 c_0 = G_1 + G_0 P_1 + P_1 P_0 c_{in}$$

$$c_2 = G_2 + P_2 c_1 = G_2 + G_1 P_2 + G_0 P_1 P_2 + P_2 P_1 P_0 c_{in}$$

$$c_3 = G_3 + P_3 \cdot c_2 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + P_3 P_2 P_1 P_0 c_{in}$$

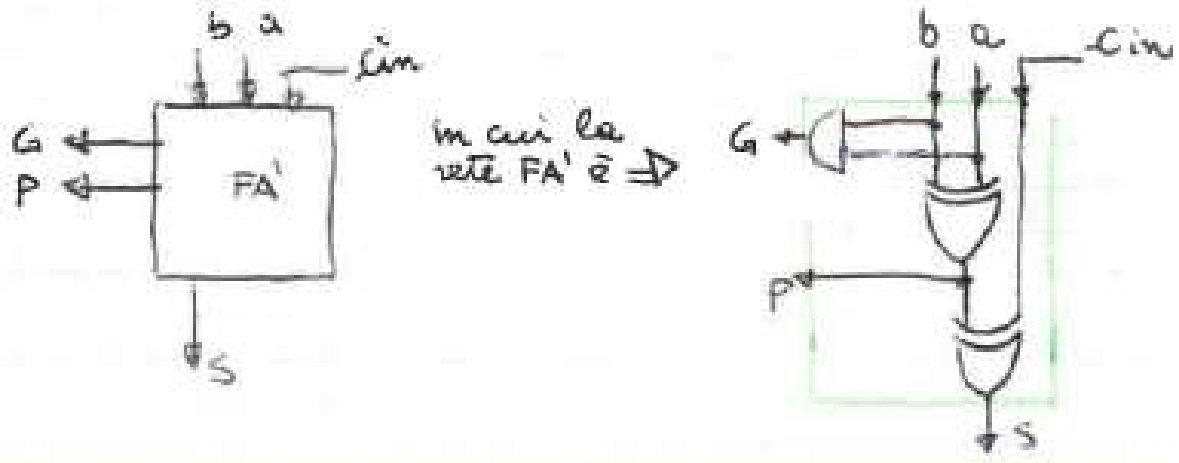
Carry Look-Ahead Adder (2)

C_{i-1}	a_i	b_i	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Un altro modo di vedere la funzione di Generate e Propagate:

In questi due casi il carry di uscita è presente perché "si genera" a causa del fatto che nella somma ai e bi sono entrambi "a uno"

In questi altri due casi il carry di uscita è presente perché "si propaga" dal carry precedente (ovvero dal Full Adder precedente)



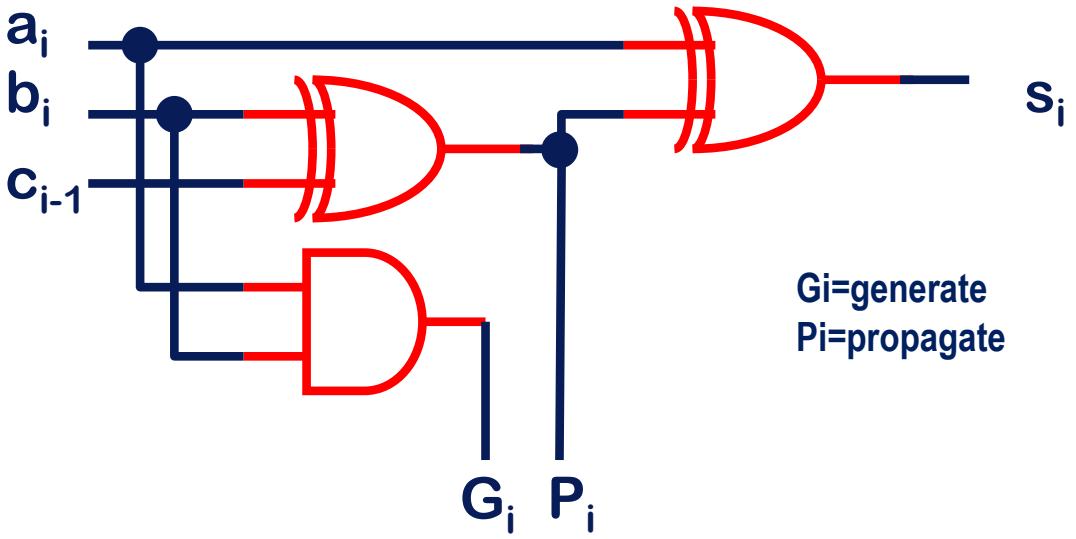
Full Adder per Carry-Look-Ahead in VERILOG

$$G_i = a_i \cdot b_i \qquad P_i = a_i \oplus b_i$$

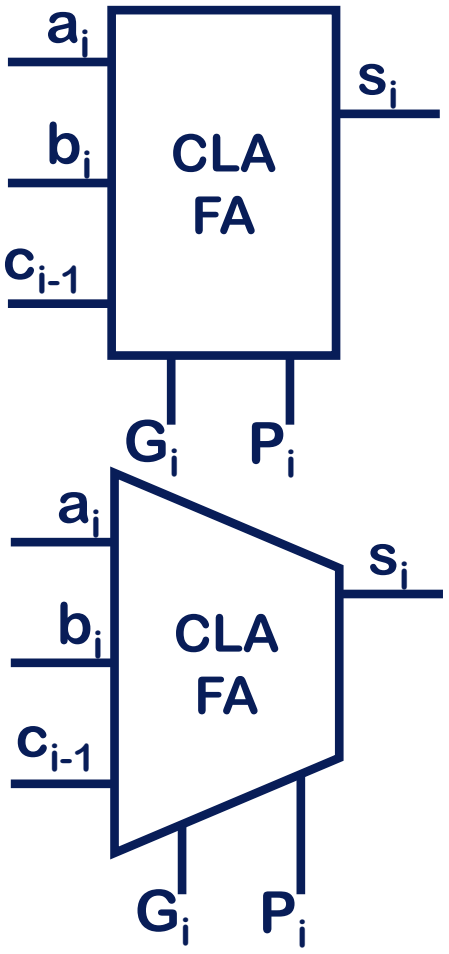
$$s_i = a_i \oplus b_i \oplus c_{i-1} = P_i \oplus c_{i-1}$$

$$\begin{aligned} c_i &= a_i \cdot b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1} \\ &= a_i \cdot b_i + a_i \cdot c_{i-1} \cdot (a_i + b_i) \\ &= a_i \cdot b_i + a_i \cdot c_{i-1} \cdot (a_i \oplus b_i) \\ &= G_i + c_{i-1} \cdot P_i \end{aligned}$$

ai+bi si può sostituire col suo xor perché nell'espressione quando ai=1 e bi=1 anche ai*bi=1



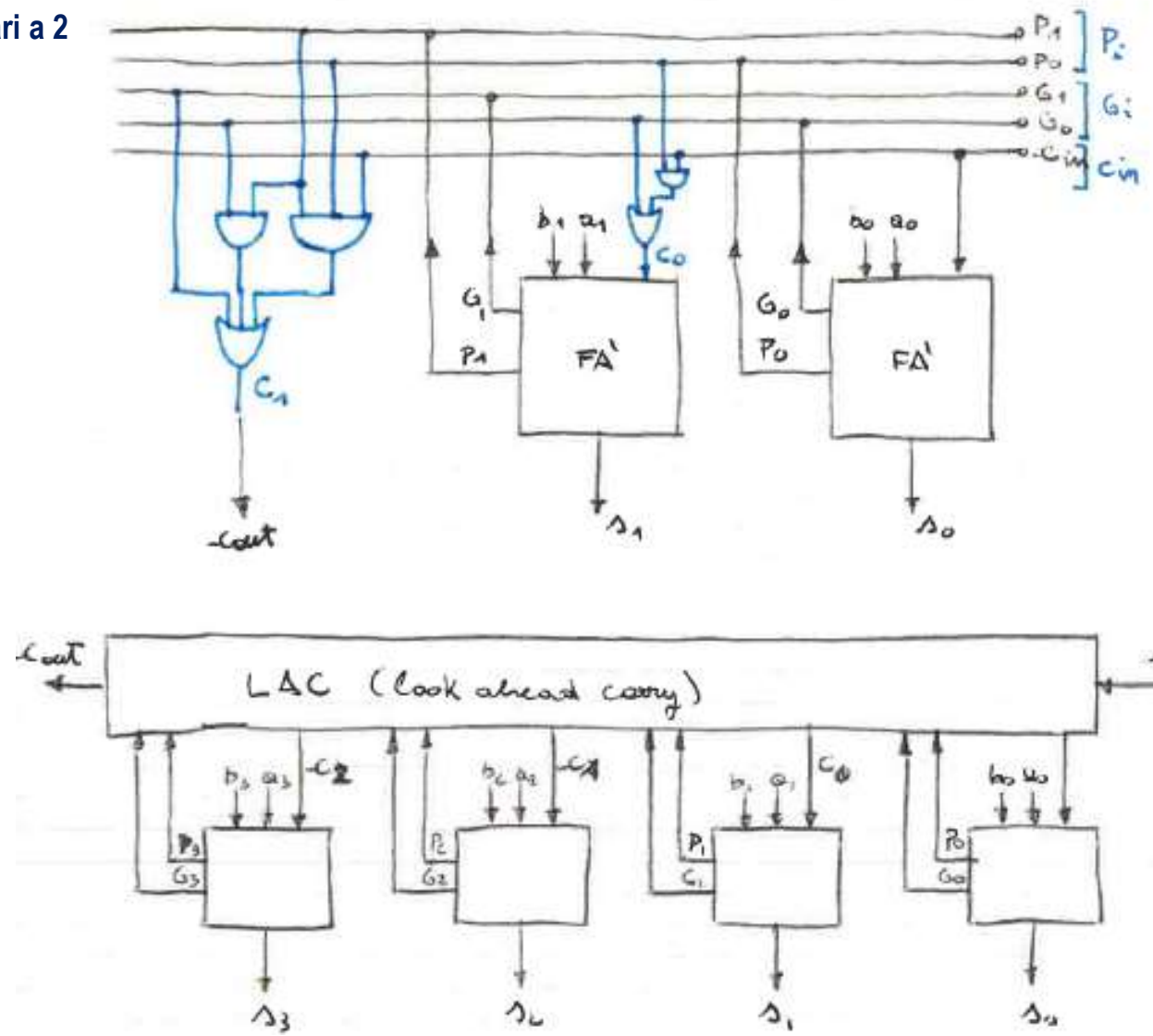
Gi=generate
Pi=propagate




```
module cla_full_adder(input a,input b,input c,output g,output p,output s);
        assign g = a & b;
        assign p = a ^ b;
        assign s = a ^ (b ^ c);
endmodule
```

Carry Look-Ahead Adder (3)

Es. Nel caso pari a 2



$$\tau_{cout} = \tau_{LAC}$$

$$\tau_{tot} = \tau_{LAC} + \tau_{FA'}$$

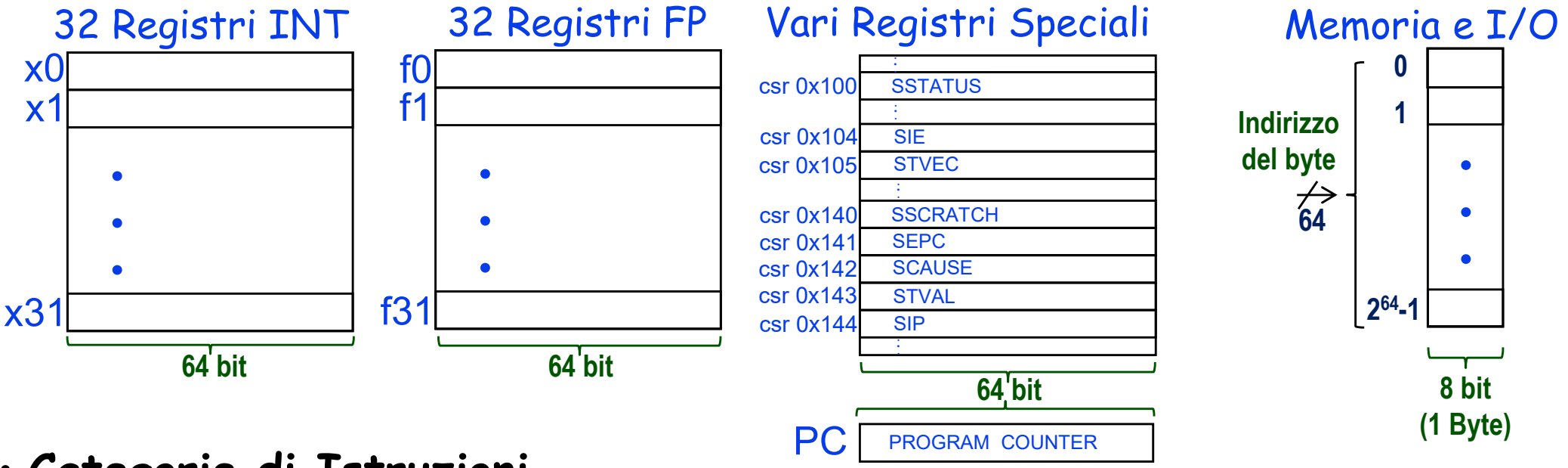
Carry Look-Ahead Adder (4 bit) in VERILOG

```
module cla_adder_4bit(input wire[3:0]a,input wire[3:0]b,input wire cin,
                      output wire[3:0]s,output wire cout);
    wire [4:0] c;
    wire [3:0] g, p;

    assign c[0] = cin;
    assign cout = c[4];

    cla_full_adder add0(a[0],b[0],c[0],g[0],p[0],s[0]);
    assign c[1] = g[0] | (p[0] & c[0]);

    cla_full_adder add1(a[1],b[1],c[1],g[1],p[1],s[1]);
    // assign c[2] = g[1] | (p[1] & c[1]);
    assign c[2] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & c[0]);

    cla_full_adder add2(a[2],b[2],c[2],g[2],p[2],s[2]);
    // assign c[3] = g[2] | (p[2] & c[2]);
    assign c[3] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0])
                | (p[2] & p[1] & p[0] & c[0]);

    cla_full_adder add3(a[3],b[3],c[3],g[3],p[3],s[3]);
    // assign c[4] = g[3] | (p[3] & c[3]);
    assign c[4] = g[3]|(p[3]&g[2])|(p[3]& p[2]&g[1])|(p[3]&p[2]&p[1]&g[0])
                  |(p[3]&p[2]&p[1]&p[0]&c[0]);

endmodule
```

Lezione 7
Il Set di Istruzioni RISC-V

"Instruction Set Architecture" o "ISA" (es. RISC-V RV64IMDF)

- ISA = OPERANDI + ISTRUZIONI
  - Le istruzioni sono codificate con un NUMERO a 32 bit o in altri termini la lunghezza delle istruzioni è FISSA a 32 bit
  - Gli operandi sono costituiti da «registri» o da «locazioni di memoria»
- Noi esamineremo la specifica RV64IMDF* del processore RISC-V



- Categorie di Istruzioni
  - Load/Store, Calcolo (per Interi e per Frazionari), Jump/Branch, Istruzioni Speciali

*Nota: "RV64IMDF" significa che il processore è un RISC-V (RV) a 64 bit, con il supporto per l'aritmetica sugli interi (I), la moltiplicazione e divisione (M) le operazioni floating point (singola precisione (F) e doppia precisione (D)

(Richiamo) Prefissi per i multipli di bit (b) e byte (B)

Valore decimale		Simbolo del Sistema Internazionale	
1000	10^3	k	kilo
1000^2	10^6	M	mega
1000^3	10^9	G	giga
1000^4	10^{12}	T	tera
1000^5	10^{15}	P	peta
1000^6	10^{18}	E	exa
1000^7	10^{21}	Z	zetta
1000^8	10^{24}	Y	yotta

Valore binario		Simbolo IEC	
1024	2^{10}	Ki	kibi
1024^2	2^{20}	Mi	mebi
1024^3	2^{30}	Gi	gibi
1024^4	2^{40}	Ti	tebi
1024^5	2^{50}	Pi	pebi
1024^6	2^{60}	Ei	exbi
1024^7	2^{70}	Zi	zebi
1024^8	2^{80}	Yi	yobi

Istruzioni Aritmetiche RISC-V (1)

Patterson 2.2

- Tutte le istruzioni aritmetiche hanno 3 operandi
- L'ordine degli operandi è fisso
  (il primo è sempre l'operando destinazione)
- Es:

  C code: `A = B + C`

  RISC-V code: `add x5, x6, x7`

  → `x5≡A, x6≡B, x7≡C`

Istruzioni Aritmetiche RISC-V (2)

- Principio di progetto:
  "la semplicità favorisce la regolarità".
- Naturalmente questo complica qualcosa…

- C code:

```
A = B + C + D;
E = F - A;
```

→ x5≡A, x6≡B, x7≡C, x8=D, x9≡E, x10≡F

- RISC-V code:

```
add x11, x6, x7
add x5, x11, x8
sub x9, x10, x5
```

- Gli operandi delle istruzioni aritmetiche devono SEMPRE essere registri
- Si hanno solo 32 registri (per gli interi) a disposizione
  - Cosa accade se il programma ha moltissime variabili o fa accesso a strutture dati complesse (es. vettori)?

Istruzioni di accesso alla memoria

Patterson 2.3

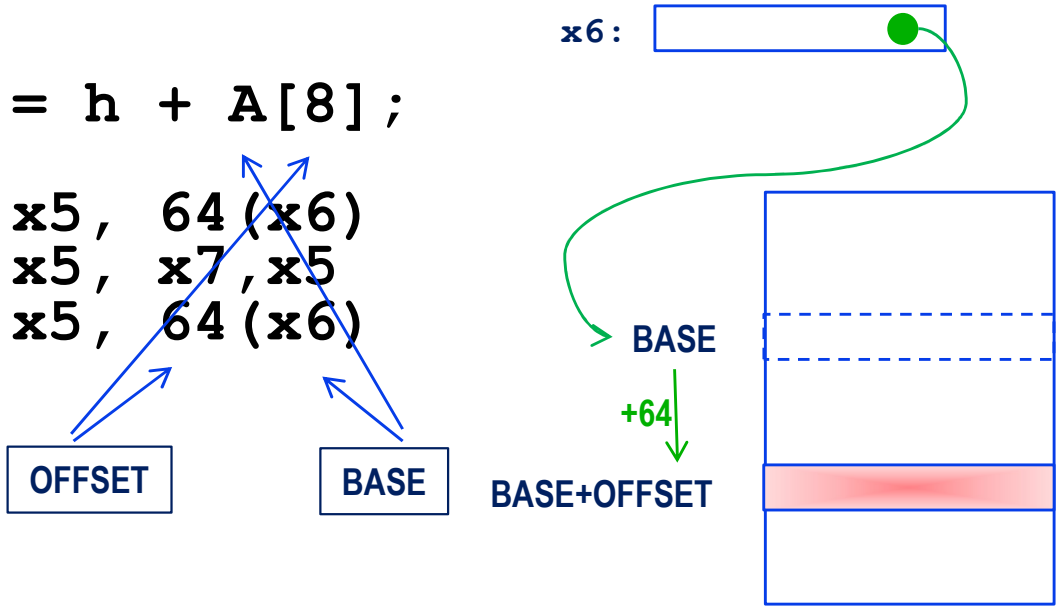
- Istruzioni Load e Store

  Es.: x6≡A, x7≡h

C code: `A[8] = h + A[8];`

RISC-V code:

```
ld   x5, 64(x6)
add  x5, x7,x5
sd   x5, 64(x6)
```



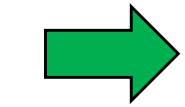
- NOTE:
  - Il dato caricato è 64 bit (1 dword=8 bytes) – suffisso 'd'
  - Nella 'sd' (al contrario delle altre istruzioni con operandi), la destinazione (in memoria) è l'ultimo operando, anziché il primo
  - gli operandi di istruzioni Aritmetiche sono SEMPRE REGISTRI, mai celle di memoria! → NEL RISC-V, NON ESISTE add x5, x7, 64(x6)

Cosa fare se l'offset è dinamico?

- Esempio: scambio di due elementi di un vettore

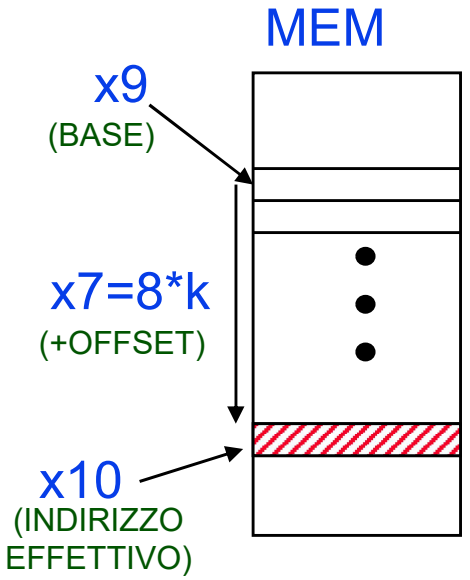
```
temp = v[k]
v[k] = v[k+1];
v[k+1] = temp;
```

Il tipo "int" del C è qui a 64 bit



```
x8 ≡ k
x9 ≡ v
x5 ≡ temp
```

```
add  x7, x8, x8 # k*2
add  x7, x7, x7 # k*4
add  x7, x7, x7 # k*8
add  x10,x9, x7 # base+8*k
ld   x5, 0(x10) # temp=...
ld   x6, 8(x10) # v[k+1]
sd   x6, 0(x10) # v[k]=...
sd   x5, 8(x10) # v[k+1]=...
```



- Nota: non abbiamo l'istruzione, es., ~~'ld x5, k(x10)'~~

Nota: v è un vettore, quindi quando (in C) scrivo "v" intendo implicitamente "l'indirizzo base del vettore v"

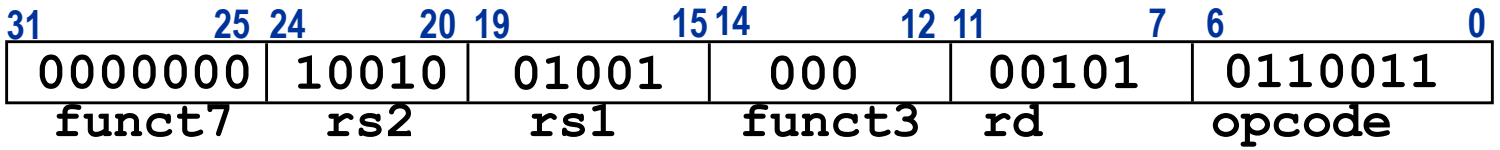
Linguaggio Macchina

Patterson 2.5

- Le istruzioni, come i registri e le word sono a 32 bit

  Es.: `add x5, x9, x18`

- Formato dell'istruzione «add»:



QUESTO FORMATO è CHIAMATO "R"

- *Che cosa significano i nomi dei campi?*
  - *rs1 = first source register*
  - *rs2 = second source register*
  - *rd = destination register*
  - *Opcode = codice operativo*
  - *funct3,funct7 = codici funzione (estendono opcode)*

Per codificare il numero di registro ho bisogno di 5 bit

Opcode, determina il formato dell'istruzione e insieme a funct… (quando presente) cosa fa l'istruzione

- Il formato R è usato es. anche dalle istruzioni: `sub,and,or,xor,slt`

Accesso alla memoria

- Per fare accesso alla memoria, si usano solo due istruzioni: load e store
  - Es. di load: `ld x5, 32(x18)` (l sta per load e d sta per dword ovvero 64 bit)
  - Es. di store: `sd x5, 32(x18)` (s sta per store e d sta per dword ovvero 64 bit)
    - Qua t0 indica sorgente (load) o destinazione (store) in un registro
    - «32(x18)» indica l'indirizzo di memoria sommando «32» al contenuto del registro x18
- Devo quindi essere in grado di memorizzare da qualche parte il numero «32»
  - Cosa dovremmo fare in base al "principio di regolarità"?
  - Nuovo principio: "un progetto ottimo richiede compromessi"
- Introduciamo quindi un nuovo tipo di formato per le istruzioni (cambiando il meno possibile, ovvero la parte evidenziata in rosso)

Es.: `ld x5, 32(x18)`

0000 0010 0000	10010	011	00101	0000011
imm12[11:0]	rs1	funct3	rd	opcode

QUESTO FORMATO è CHIAMATO "I"

- Per la store però, volendo rispettare il fatto che t0 stavolta è un registro sorgente (rs2), che si deve trovare nello stesso punto della codifica, occorre riferirsi ad un formato leggermente diverso:

Es.: `sd x5, 32(x18)`

0000 001	00101	10010	011	00000	0100011
imm12[11:5]	rs2	rs1	funct3	imm12[4:0]	opcode

QUESTO FORMATO è CHIAMATO "S"

Che tipo di compromessi abbiamo fatto?

Controllo del flusso del programma: salti condizionati e «formato B»

- L'architettura di Von Neumann segue il ciclo «fetch-and-execute» ovvero vengono prelevate in sequenza le istruzioni che si trovano in memoria
- Il flusso sequenziale delle istruzioni può però essere alterato in conseguenza di una decisione del programma → la macchina supporta questo meccanismo con le istruzioni:

```
bne x5, x6, Label
beq x5, x6, Label
```

Es.: if (b==c) a = b + c;

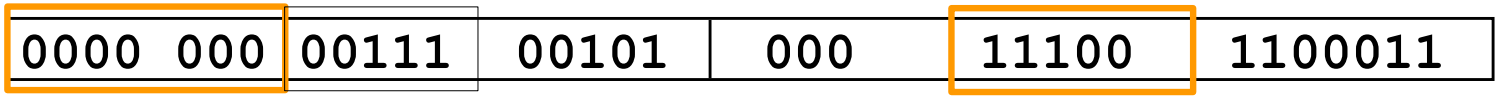
x7≡a, x5≡b, x6≡c

```
bne x5, x6, Label
add x7, x5, x6
Label:      ....
```

Es. +7 istruzioni =28 Bytes

- Per quanto riguarda la codifica, l'etichetta («Label» nell'esempio) NON PUÒ essere memorizzata come tale: viene trasformata nel n. di byte (OFFSET) che separa l'istruzione di salto dalla posizione dell'etichetta stessa (es. +7 istr. x 4B ciascuna= 28B)
- Tipicamente il salto avviene nelle vicinanze, quindi, tale OFFSET è spesso un numero piccolo che potrebbe essere memorizzato usando i 12 bit del campo immediato del formato I o S. Inoltre dato che il RISC-V usa un allineamento delle istruzioni sui 16 bit (non 32) l'ultima cifra binaria di tale numero è sempre zero quindi tali 12 bit «valgono 13»
- Volendo però conservare rs1 e rs2 nella stessa posizione viene introdotto un formato leggermente diverso (formato B):

Es.: `beq t0, t1, label`



QUESTO FORMATO E' CHIAMATO "B" (o "SB")

Nota: I bit imm13[10:5] e imm13[4:1] non cambiano di posto rispetto al formato S, ma i bit 11 e 0 del formato S sono interpretati diversamente come bit 12 e 11 rispettivamente, dato che le istruzioni RISC-V stanno a multipli di 16 bit (bit 0 sempre 0)

Nell'esempio {imm13[12:1],0} codifica 0 0000 0001 1100 ovvero 28

Gestione della condizione

- Abbiamo due sole istruzioni per gestire il salto su condizione (beq, bne)
  - come si fa un salto su "minore" (e <=, >, >=) ?
- Ci basta in realtà una sola istruzione aggiuntiva: Set on Less Than che mette in un registro l'esito del confronto con l'operatore "<"

```
slt x5, x17, x18
```

mette in x5 il valore «1» se x17<x18 altrimenti mette in x5 «0»

- Dopodiché la beq/bne può confrontare x5 con 1 o 0 e fare un salto

- Possiamo poi generalizzare:
  - per il confronto su ">=" basta invertire la condizione (`bne`)
  - per il confronto su ">" basta scambiare gli operandi della `slt`
  - per il confronto su "<=", inverto condizione e scambio operandi

Nota: la `slt` non usa un formato nuovo ☺ → usa il formato R dato che opera solo su registri

Costanti e istruzioni «con immediato»

- Le costanti "piccole" sono le più usate (50% dei casi), es.:

```
A = A + 5;
B = B + 1;
C = C - 18;
```

- Approccio RISC:
  - Mettere le costanti "piccole" nell'istruzione
  - Mettere le costanti meno frequenti in memoria
  - Creare un registro hard-wired (x0) per la costante 0
- Istruzioni:

```
addi x9, x9,  4
slti x5, x6, 10
andi x5, x7, 0x001
ori  x6, x7, 0xFF0
xori x6, x8, 0x0F0
```

  - per codificare queste istruzioni si usa ancora il formato I
    (I sta per 'immediato': uno degli operandi è dentro l'istruzione stessa)
    QUINDI la costante piccola è un numero da -2048 a 2047 (intero con segno su 12 bit)

Nota2: la "`subi`" non esiste... si fa con la `addi` coll'immediato negato

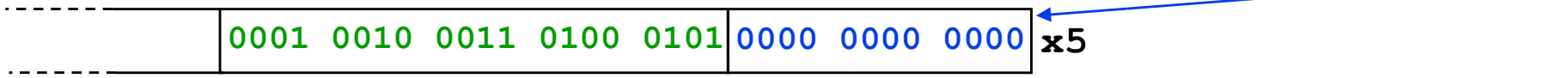
Che si fa con le costanti "grandi"?

- Vorremo caricare ad es. costanti a 32 bit in un registro
  - Es. 0x1234'5678
  - La parte bassa (0x678) la potrei caricare con `ori x5, zero, 0x678`
  - Come carico però la parte alta di 20 bit (32 bit - 12 bit = 20 bit)?

1) Si introduce una nuova istruzione ("load upper immediate" = `lui`)

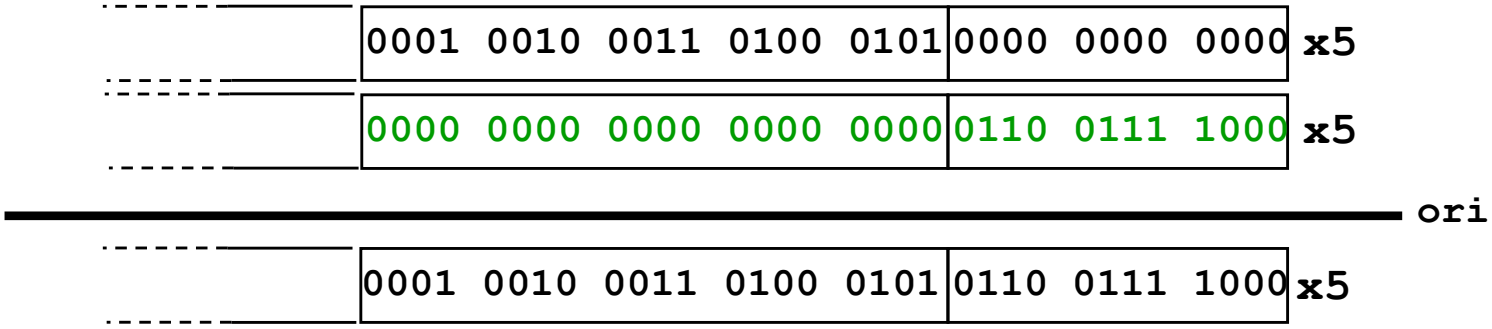
`lui x5, 0x12345` → ho però bisogno di un nuovo formato U (vedi slide successiva)

Riempita con zeri



2) Si usa successivamente la `ori` per caricare la parte bassa (12 bit)

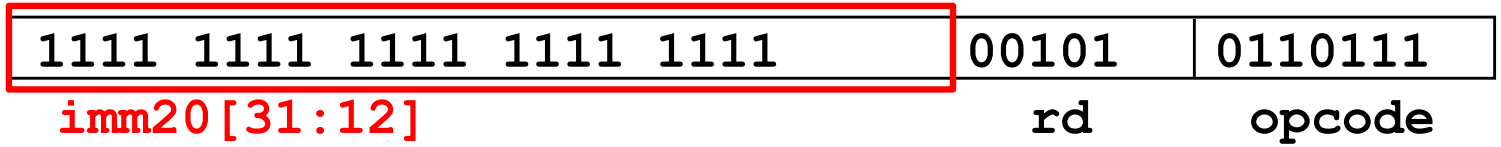
`ori x5, x5, 0x678`



Nota: per costanti più grandi (es. a 64 bit) si ripete due volte il ragionamento fatto per la costante a 32 bit, traslando verso sinistra (vedi più avanti l'istruzione per lo «shift» verso sinistra `slli`)

Formato U

Es.: `lui x5, 0xFFFFF`



QUESTO FORMATO È CHIAMATO "U"

20 bit che vanno a finire sui bit 31:12 del registro destinazione

La pseudoistruzione «LOAD ADDRESS» (la)

- La coppia di istruzioni
  - lui x5, 20bit_alti
  - ori x5, x5, 12bit_bassi

la x5, costante_a_32_bit

è talmente frequente che a livello di assembler si può creare una 'pseudoistruzione' → in questo caso: la = "load address"

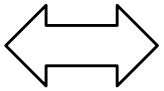
dove costante_a_32_bit:=(20bit_alti|12bit_bassi)
viene automaticamente spezzata all'interno del campo immediato delle istruzioni native lui e ori

- Questa pseudoistruzione è molto utile per caricare in un registro «il nome» di una variabile (ovvero l'indirizzo di una variabile) – esempio:

```
int MAGIC;

int main() {
…
MAGIC = 1234;
…
}
```

Nota: questa è una variabile GLOBALE che quindi risiede nella memoria dati (sezione ".data" del codice assembly)

```
.data
MAGIC: dword(0)
```



```
.text
globl main
main:
…
addi x6, zero, 1234
la   x5, MAGIC
sd   x6, 0(x5)
…
```

Nota2: questo è il programma principale ("main") che risiede nella memoria istruzioni (sezione ".text" del codice assembly)

Esistono altre pseudoistruzioni, che vedremo all'occorrenza

Linguaggio Assembly e Linguaggio Macchina

- Il linguaggio Assembly fornisce una conveniente rappresentazione simbolica
  - È più comprensibile (a noi) che sequenze di numeri (binari)

- Il linguaggio macchina è la "realtà sottostante"
  - La macchina comprende solo le istruzioni in formato binario...

- Il linguaggio Assembly fornisce "pseudoistruzioni"
  - es., MOVE: "mv x5, x6" esiste solo in Assembly
    - l'assemblatore traduce usando "add x5,x6,zero"
  - es., LOAD IMMEDIATE: "li x5, 5" esiste solo in Assembly
    - l'assemblatore traduce usando "addi x5,zero,5"

- Quando si effettua per es. il debugging è necessario ricordare quali siano le istruzioni reali

Aritmetica con segno e senza segno

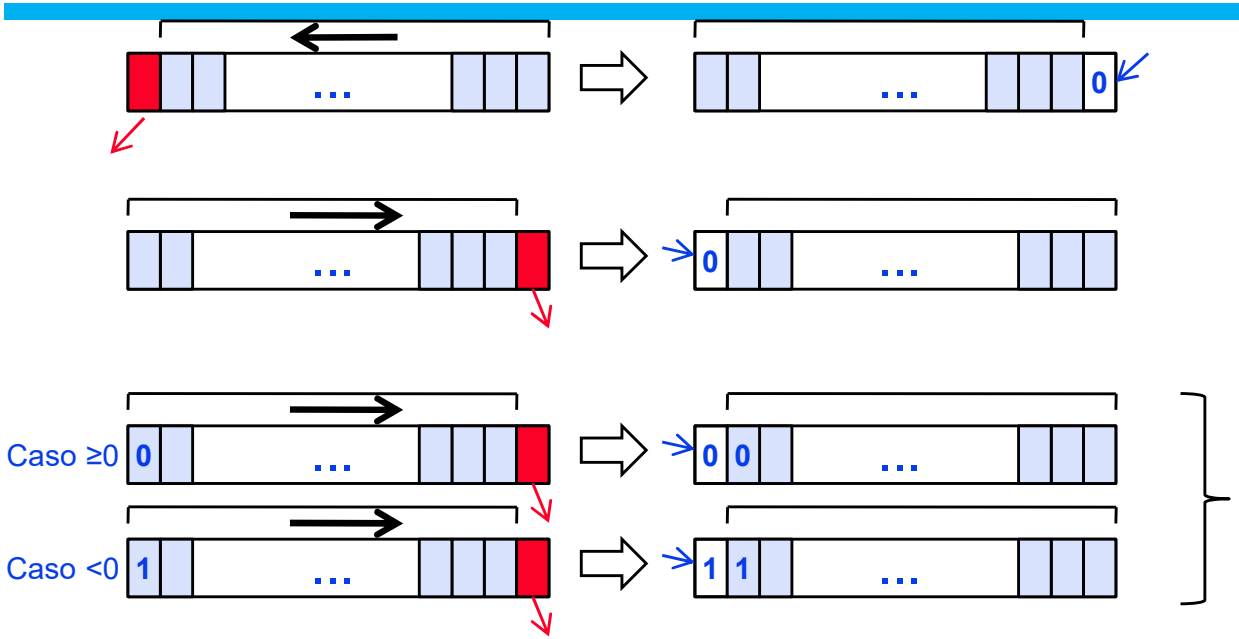
- Le istruzioni aritmetiche fin qui esaminate operano su interi con segno
  - Gli operandi (a 64 bit, nei registri) sono rappresentati in complemento a due → i valori vanno da -2^{63} a $2^{63}-1$
  - Anche i byte-offset di `ld` e `sd` e il numero di istruzioni di cui saltare nella `bne/beq` sono interi con segno da -2^{11} a $2^{11}-1$

- È possibile utilizzare anche operandi senza segno
  - In tal caso i valori vanno da 0 a $2^{64}-1$
  - Per fare un semplice esempio, consideriamo operandi a soli 3 bit:
    - Effettuando un confronto tra 0b101 e 0b001 ottengo un risultato diverso a seconda che io usi l'aritmetica con o senza segno:
    - Arit. con segno: 0b101 è -3, mentre 0b001 è 1 → -3<1 è VERO
    - Arit. SENZA segno: 0b101 è 5, mentre 0b001 è 1 → 5<1 è FALSO
  - Necessito quindi di due istruzioni diverse per fare i confronti set-on-less-then (slt):
    - `slt`, `slti` si usano per l'aritmetica con segno
    - `sltu`, `sltiu` si usano per l'aritmetica SENZA segno

Nota: in RISC-V, né "`addu`" né "`addiu`" esistono... occorre ragionarci «a software»: è il software a interpretare gli interi «con segno» o «senza segno» quando si effettuano somme o sottrazioni (per divisioni e moltiplicazioni vedremo che è diverso)

RISC-V: istruzioni di scorrimento (shift)

Patterson 2.6



Shift Left Logical (SLL)

Shift Right Logical (SRL)

Shift Right Arithmetic (SRA)

Nota: SLA non esiste ...=SLL

Instruzione	*Esempio*	*Significato*	*Commento*
shift left logical	slli x5,x6,10	x5 = x6 << 10	Shift left by constant
shift right logical	srli x5,x6,10	x5 = x6 >> 10	Shift right by constant
shift right arithm.	srai x5,x6,10	x5 = x6 >> 10	Shift right (sign extend)
shift left logical	sll x5,x6, x7	x5 = x6 << x7	Shift left by variable
shift right logical	srl x5,x6, x7	x5 = x6 >> x7	Shift right by variable
shift right arithm.	sra x5,x6, x7	x5 = x6 >> x7	Shift right arith. by variable

RISC-V: istruzioni logiche AND, OR, XOR – pseudoistruzione NOT

Istruzione	*Esempio*	*Significato*	*Commento*
and	and x5,x6,x7	x5 = x6 & x7	3 reg. operands; Logical AND
or	or x5,x6,x7	x5 = x6 | x7	3 reg. operands; Logical OR
xor	xor x5,x6,x7	x5 = x6 ⊕ x7	3 reg. operands; Logical XOR
and immediate	andi x5,x6,10	x5 = x6 & 10	Logical AND reg, constant
or immediate	ori x5,x6,10	x5 = x6 | 10	Logical OR reg, constant
xor immediate	xori x5, x6,10	x5 = x6 ⊕ 10	Logical XOR reg, constant

- **La NOT è una pseudoistruzione: perché?**
  - **Dalla teoria generale sull'algebra di Boole sappiamo che anche con la sola funzione NAND potremmo generare le altre funzioni quali NOT, AND, OR**
  - **Nel caso del processore RISC-V, seguendo l'approccio minimalista, si è deciso di non introdurre nuovi opcode per l'istruzione `not`**
  - **In particolare posso realizzare `not x5, x6` usando invece:**

    ```
    xori  x5, x6, -1
    ```

    - **Notare che prima di fare l'operazione di xor logico, l'operando «-1» verrà esteso di segno su tutti i bit (diventa 0xFFF...FF)**

Istruzioni per la manipolazione di stringhe

Patterson 2.9

- Fino a qui abbiamo visto istruzioni che operano su quantità a 32 o 64 bit
- Per accedere al singolo byte sono a disposizione

  (Utile per le stringhe di caratteri ASCII)

```
lb    x5, 0(x6)        "load byte"
sb    x5, 0(x6)        "store byte"
```

- Per accedere alla half-word (16 bit) ci sono

  (Utile per le stringhe di caratteri UNICODE (es. in Java))

```
lh    x5, 0(x6)        "load half-word"
sh    x5, 0(x6)        "store half-word"
```

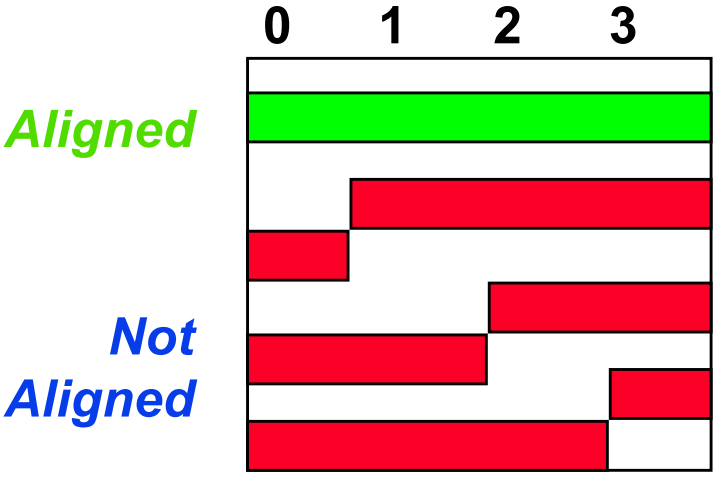
- Per accedere alla word (32 bit) ci sono

```
lw    x5, 0(x6)        "load word"
sw    x5, 0(x6)        "store half-word"
```

Nota: in fase di caricamento (load), dovendo porre la quantità da 8/16/32 bit in 64 bit, viene automaticamente effettuata l'estensione del segno. Se ciò non si vuole, si devono usare `lbu` (al posto di `lb`) e `lhu` (al posto di `lh`) e `lwu` (al posto di `lw`) → estensione con 0

Restrizioni sull'allineamento degli indirizzi

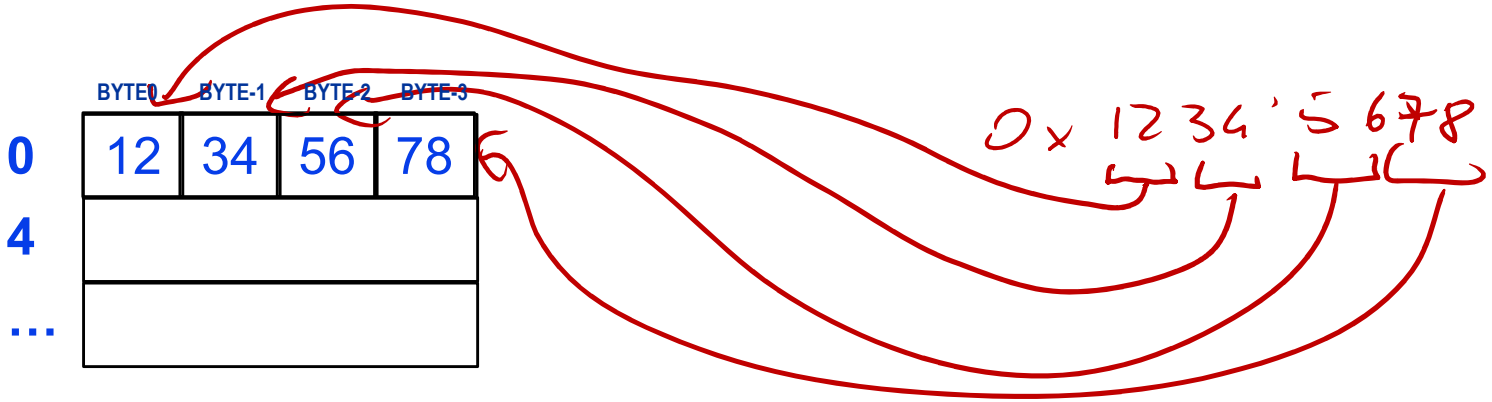
- La memoria è classicamente indirizzata "al byte"
  - Quindi, le istruzioni di load e store usano indirizzi al byte, però
    - `lw`, `lwu` e `sw` trasferiscono 32 bit
    - `lh`, `lhu` e `sh` trasferiscono 16 bit
    - solo `lb`, `lbu`, `sb` trasferiscono 8 bit
- È conveniente pertanto che l'indirizzo sia opportunamente allineato…
  - per `lw`, `lwu`, `sw` dovrebbe essere allineato ad un multiplo di 4
  - Per `lh`, `lhu`, `sh` dovrebbe essere allineato ad un multiplo di 2
- Esempi di dati ALLINEATI e NON ALLINEATI "alla word"



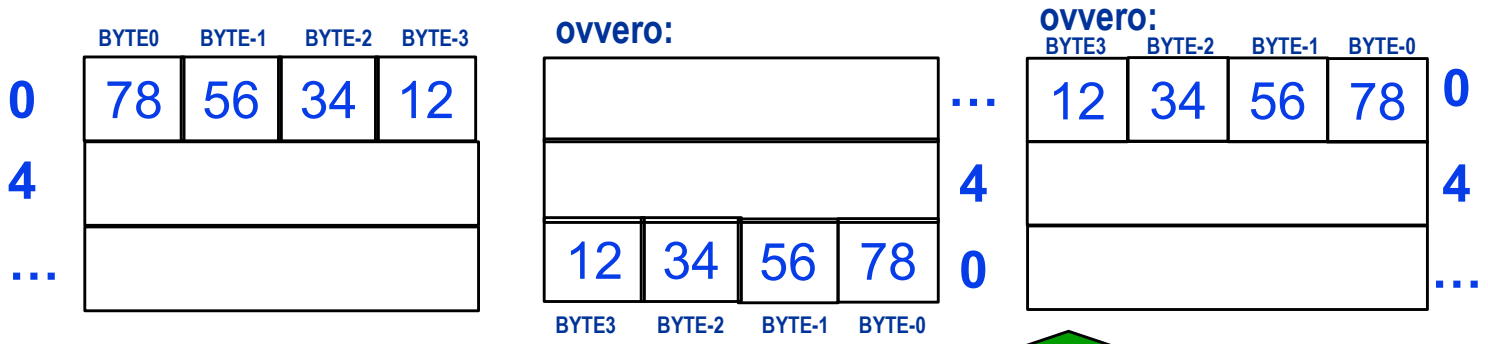
Nota: se si specifica un indirizzo non allineato rispetto a quanto l'istruzione desidera, il RISC-V genera uno warning (avvertendo che il tempo per l'accesso al dato risulterà 2 volte più lento)

Ordinamento dei byte (Endianess)

- **Big Endian: si inizia con la parte "più grossa" ovvero a indirizzi più bassi i byte più significativi: IBM 360/370, Motorola 68k, MIPS R4000, Sparc(pre-v9)**
  - Se ad esempio devo memorizzare 0x12345678 in B.E.



- **Little Endian: si inizia con la parte "più piccola" ovvero a indirizzi più bassi i byte meno significativi: Intel 80x86, DEC Vax, DEC Alpha (Windows NT), RISC-V**



Useremo questo

Nota: è importante sapere l'endianess di una macchina quando si scandiscono stringhe o dati byte-a-byte SPECIALMENTE dopo un TRASFERIMENTO DA RETE in cui la macchina remota può avere una endianess diversa

Operazione di moltiplicazione a 64 bit – cast implicito

- Si osservi il seguente frammento di codice C:

```
int a, b, c;
…
a = b * c;
…
```

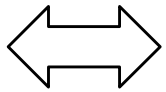
- Se la macchina definisce l'intero 'int' come una quantità a 64 bit, lo statement di assegnamento 'a=b*c' contiene un cast implicito; ovvero è come se avessi scritto:

```
int a, b, c;
…
a = (int)(b * c);
…
```

- Quindi il risultato del prodotto di due interi (con segno) a 64 bit, che sta quindi in generale su 128 bit, viene troncato scartando IMPLICITAMENTE i 64 bit più significativi!

- In altri termini:

```
int a, b, c;
…
a = b * c;
…
```

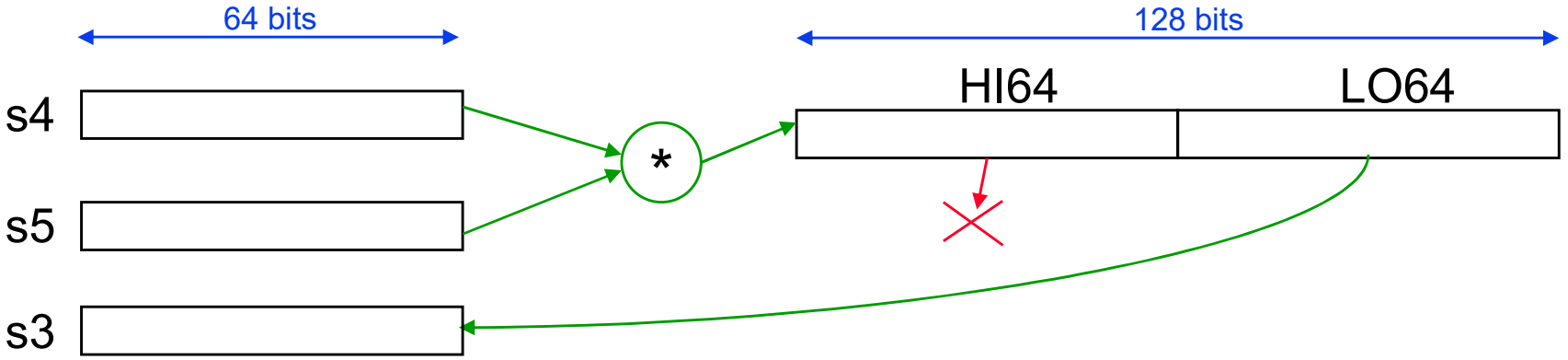


```
s3≡a, s4≡b, s5≡c

mul  s3, s4, s5
```

Moltiplicazione

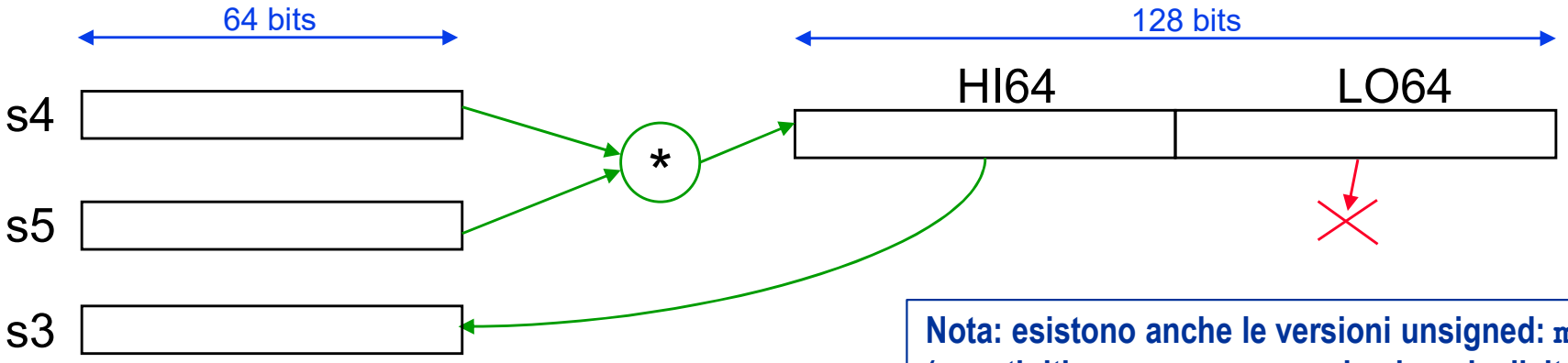
`mul s3, s4, s5` ⟺ `s3 = (int)(s4 * s5)`

- Esiste quindi un registro interno (nascosto all'utente)



- Ma si possono ottenere i bit più significativi (HI64) con l'istruzione

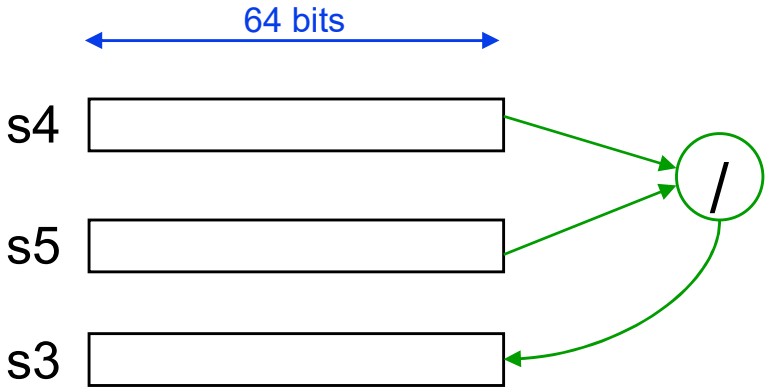
`mulh s3, s4, s5`



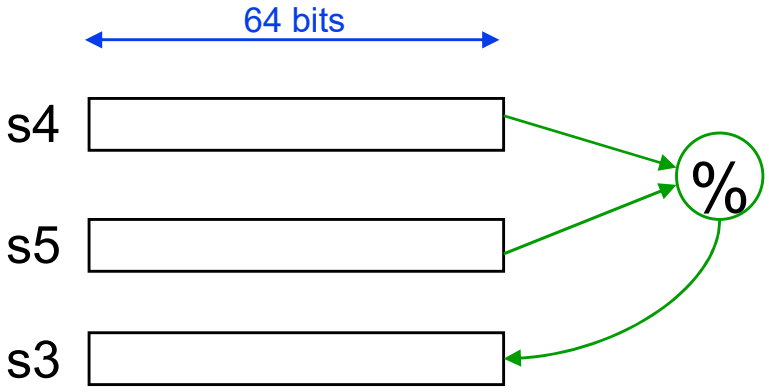
Nota: esistono anche le versioni unsigned: `mulhu` e `mulhsu` (quest'ultima con un operando signed e l'altro unsigned)

Divisione e Resto

`div s3, s4, s5` ⟺ **s3 = s4 / s5**



`rem s3, s4, s5` ⟺ **s3 = s4 % s5**



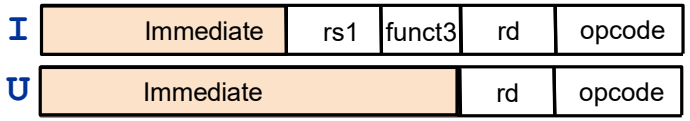
Nota: esistono anche le versioni unsigned: `divu` e `remu`

Ricapitolazione istruzioni RISC-V per categoria

- Aritmetiche
  - ADD/ADDI/SUB/MUL/DIV/REM/MULH
  - ADDW/ADDIW/SUBW/MULW/DIVW/REMW
  - SLT/SLTI/SLTU/SLTIU
  - MV/LI/LA (pseudoistruzioni, usano ADD/ADDI/LUI)
- Logiche
  - AND/OR/XOR/ANDI/ORI/XORI
  - SLL/SLLI/SRL/SRLI/SRA/SRAI
  - SLLW/SLLIW/SRLW/SRLIW/SRAW/SRAIW
- Accesso alla memoria
  - LB/LBU/LH/LHU/LW/LWU/LD
  - SB/SH/SW/SD
- Diramazione, Salto, Eccezioni
  - BEQ/BNE
  - B/JAL/RET (pseudoistruzioni, usano BEQ/JALR)
  - ECALL/SRET

Tabella riassuntiva modi indirizzamento del RISC-V:

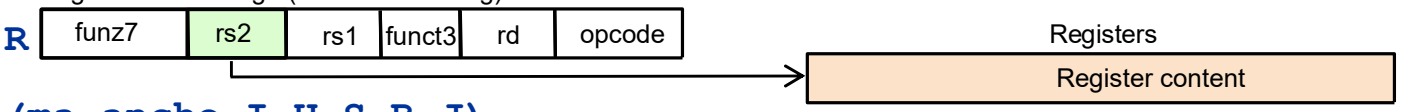
1. Immediate addressing



Usato da:
ADDI/ADDIW, SLTI, ANDI/ORI/XORI
SLLI/SLLIW/SRLI/SRLIW/SRAI/SRAIW,ECALL

Usato da: LUI

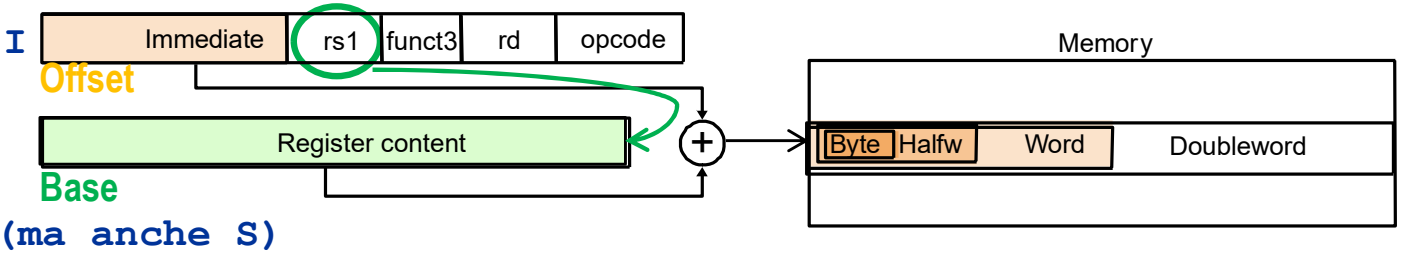
2. Register addressing: (direct addressing)



(ma anche I,U,S,B,J)

Usato in modo esclusivo da:
ADD/SUB/MUL/DIV/REM
ADDW/SUBW/MULW/DIVW/REMW
SLT/SLTU SRET
ma anche (parzialmente) da tutte le altre istruzioni e formati ove c'è almeno un registro

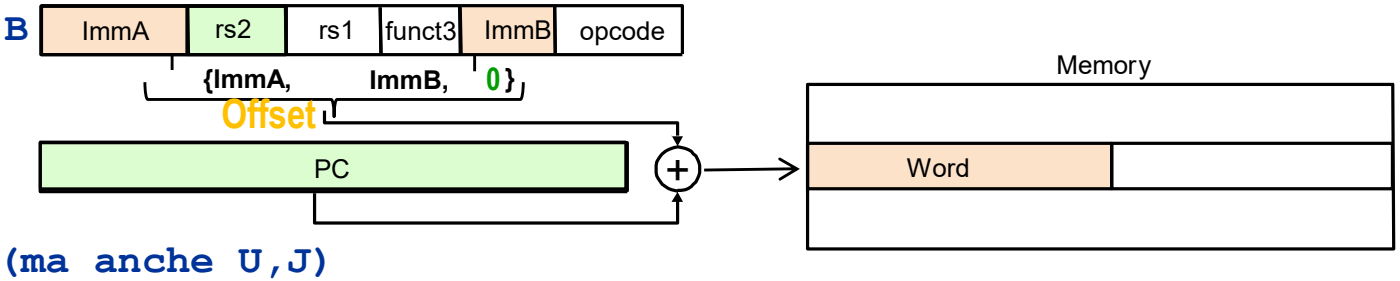
3. Base addressing (Base + Offset)



(ma anche S)

Usato da:
LB/LBU/LH/LHU/LW/LWU/LD
(formato I)
SB/SH/SW/SD
(formato S)

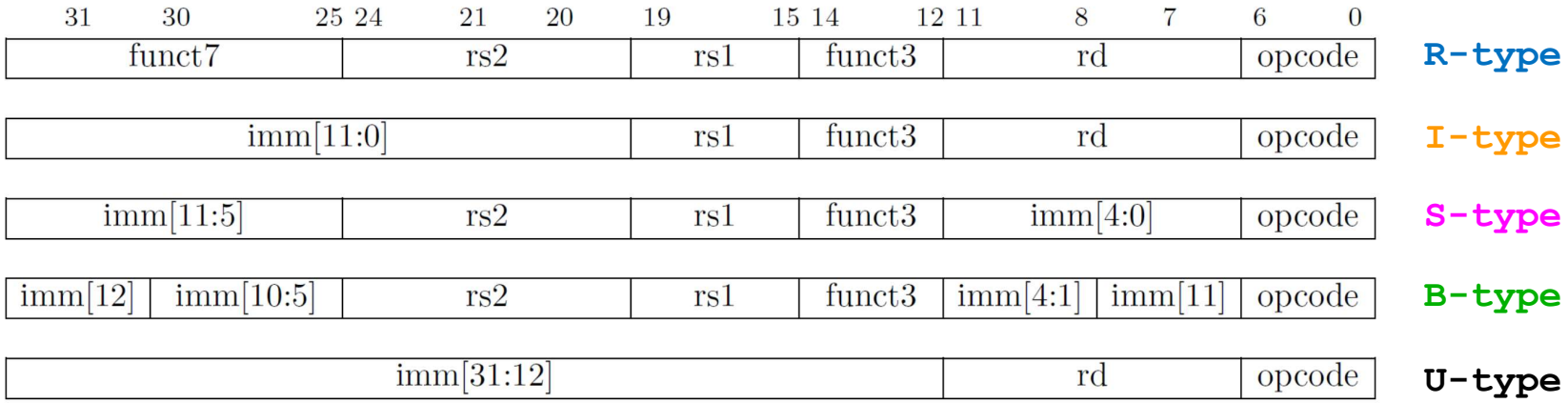
4. PC-relative addressing (Base + Offset) PC + 2*Immediate



(ma anche U,J)

Usato da:
BEQ/BNE/B
(formato B)
JAL/RET
(ovvero JALR - formato J)
AUIPC
(formato U)

Ricapitolazione dei formati e istruzioni rappresentative



Istruzione	Significato	Variante	Significato
`add s1,s2,s3`	`s1 = s2 + s3`	`addi s1,s2,10`	`s1 = s2 + 10`
`sub s1,s2,s3`	`s1 = s2 - s3`		
`slt t0,s1,s2`	`t0 = (s1<s2) ? 1 : 0`	`slti s1,s2,10`	`t0 = (s1<10) ? 1 : 0`
`lw s1,100(s2)`	`s1 = Memory[s2+100]` (32 bit)	`ld s1,100(s2)`	legge 64 bit
`sw s1,100(s2)`	`Memory[s2+100] = s1` (32 bit)	`sd s1,100(s2)`	scrive 64 bit

  `bne s4,s5,L` salta a L se s4!=s5 (L si trova a offset imm13=(&L-PC))

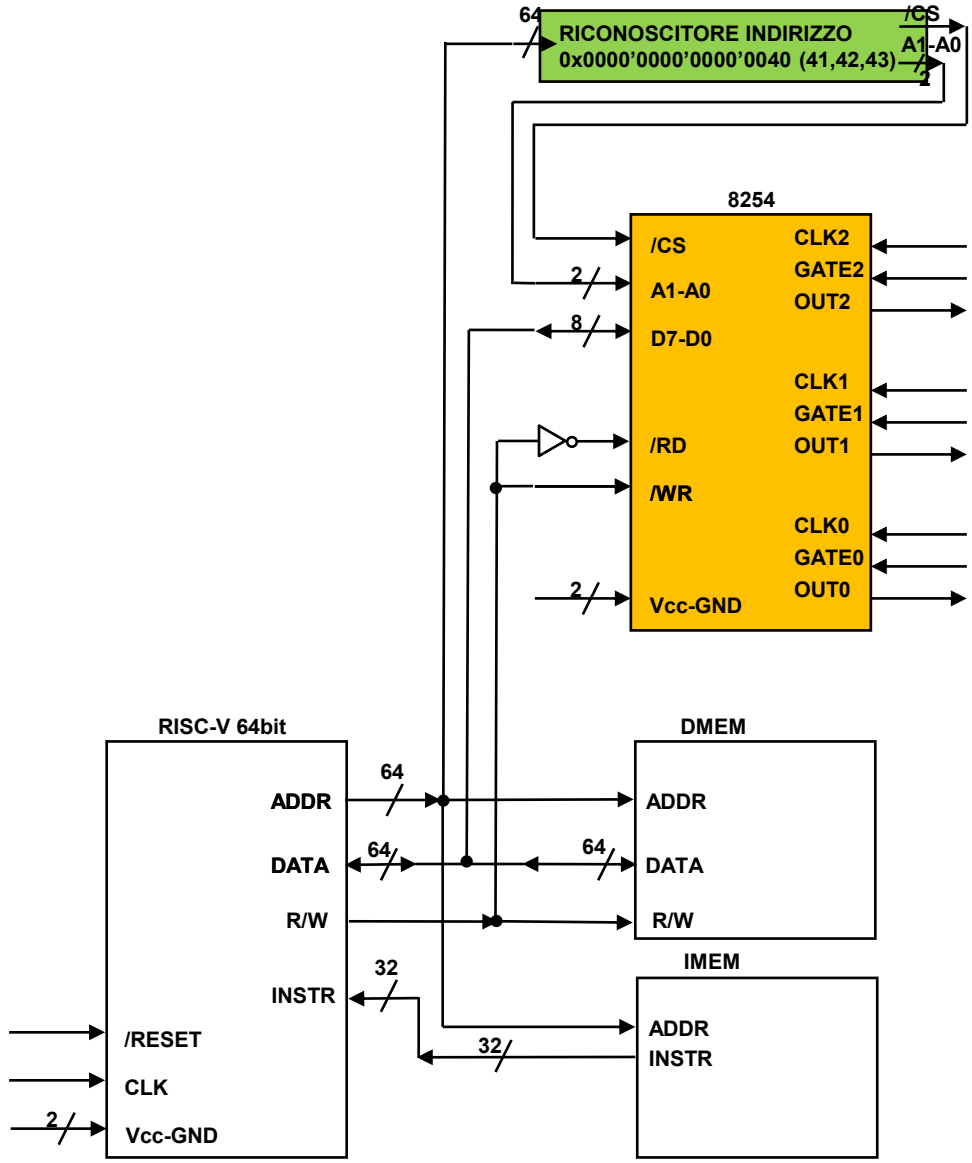
  `beq s4,s5,L` salta a L se s4==s5 (L si trova a offset imm13=(&L-PC))

  `lui x5,imm20` carica imm20 nei 20 bit alti

Lezione 8 - Esempi di dispositivi di I/O: il TIMER

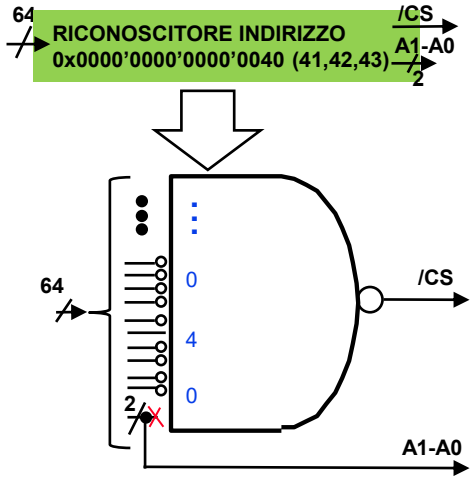
- Come riferimento consideriamo il timer Intel-8254
- A cosa può servire:
  - Generare ritardi programmabili in maniera accurata
  - Real time clock
  - Conteggio eventi
  - Generatore di frequenza programmabile
  - Generatore di onda quadra programmabile
  - Generatore di "impulso digitale"
  - Moltiplicatore binario di frequenza
  - Generatore di forma d'onda complessa
  - Controllo di motori
- È un buon esempio per mostrare la complessità interna di un'interfaccia di I/O
- Datasheet reperibili sul sito del corso
  https://web.archive.org/web/20150603122044/https://download.intel.com/design/archives/periphrl/docs/23124406.pdf

Timer Intel-8254: Visione elettrica del chip e di insieme



- /CS: Chip-Select
- A1-A0: indirizzamento
- D7-D0: scambio dati su 8 bit
- /RD, /WR: Read, Write
- Vcc-GND: alimentazione

- CLKj: clock del timer j
- GATEj: trigger hardware o congelamento conteggio
- OUTj: forma d'onda prodotta dal timer j



Timer Intel-8254: Visione Funzionale

- **Visione Funzionale del chip:**
  - **3 contatori indipendenti a 16-bit**
  - **6 modalità di conteggio (detti 'modi operativi')**

MODO	FUNZIONE	Condizione iniziale	Timing generato
0	Interrupt al termine del conteggio	OUT=X,GATE=1	OUT(N·T_C)=0,OUT(∞)=1
1	Programmable one shot	OUT=1,GATE=0	GATE=1→OUT(N·T_C)=0, OUT(∞)=1
2	Rate generator	OUT=X,GATE=1	OUT((N-1)·T_C)=1,OUT(1·T_C)=0,...
3	Square wave rate generator	OUT=X,GATE=1	OUT(~N/2·T_C)=1, OUT(~N/2·T_C)=0,...
4	Software triggered strobe	OUT=X,GATE=1	OUT(N·T_C)=1,OUT(1·T_C)=0,OUT(∞)=1
5	Hardware triggered strobe	OUT=1,GATE=0	GATE=1→OUT(N·T_C)=1,OUT(1·T_C)=0,OUT(∞)=1

GENERAZIONE DI UN GRADINO (verso uno)

GENERAZIONE DI ONDA PERIODICA

GENERAZIONE DI UN IMPULSO (verso zero)

Timer Intel-8254 (2)

- **Visione logica del chip (tutti registri a 8 bit):**

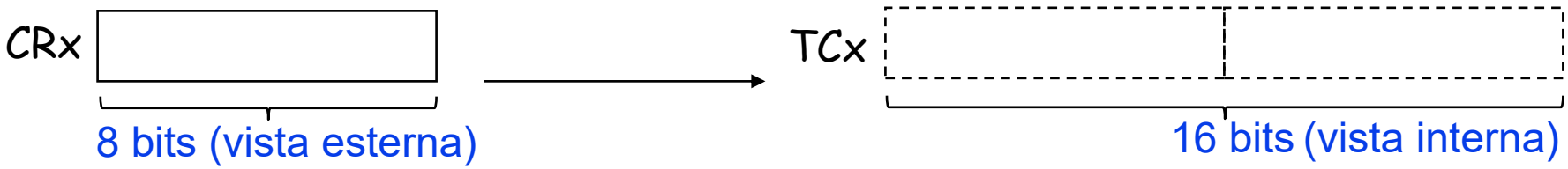
DR

A1-A0 /RD /WR			
00 1 0	CR0		Counter Register 0 Scrivendo in questo registro fisso la Time Constant 0
01 1 0	CR1		Counter Register 1 Scrivendo in questo registro fisso la Time Constant 1
10 1 0	CR2		Counter Register 2 Scrivendo in questo registro fisso la Time Constant 2

CR

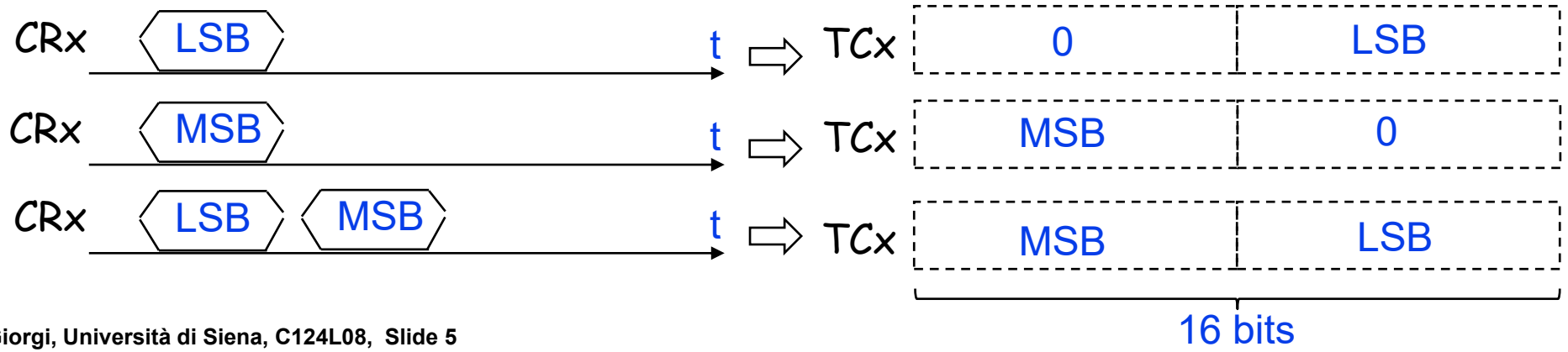
11 1 0	CWR		Control Word Register Scrivendo in questo registro fisso il funzionamento di un dato contatore oppure dò comandi particolari

Nota: la Time Constant è a 16 bit. CRj funziona come un "bus" a 8 bit verso un registro interno a 16 bit che contiene tale Time Constant



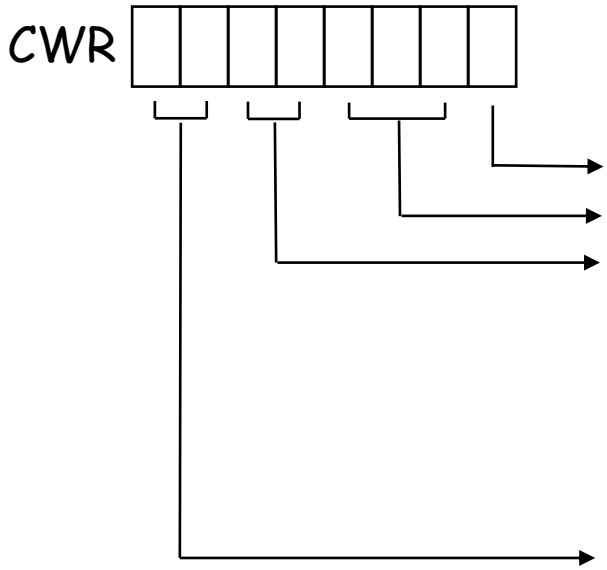
Modalità di lettura/scrittura

- Poiché nei dispositivi di I/O si tende ad utilizzare un numero limitato di indirizzi (nel nostro caso 4) si usa un registro per dare i comandi (CWR) e gli altri registri per leggere o scrivere dati (CR0,CR1,CR2)
- Dato che i contatori sono **internamente** a 16 bit e posso scrivere solo 8 bit (1 byte) per volta: in quale ordine vengono inviati tali byte?
  - LSB+MSB: invio prima il byte meno significativo (Least Significant Byte o LSB) e poi su quello più significativo (Most Significant Byte o MSB)
- posso risparmiare un invio se uno dei due byte è zero
  - LSB: invio solo il byte meno significativo (es. 0x00A4 → invio solo 0xA4)
  - MSB: invio solo il byte più significativo (es. 0x B500 → invio solo 0xB5)



Programmazione del timer

1) Scrivere in CWR per selezionare uno dei contatori, *modo*, *ordine LSB/MSB*
2) Scrivere in CRj la costante di tempo
3) Abilitare il conteggio con un segnale alto sul pin GATE
4) L'uscita OUT produrrà un segnale di uscita alla fine del (oppure durante il) conteggio a seconda del modo prescelto



Tipo di aritmetica di conteggio (0=binario, 1=BCD)
Selezione Modo (da 0 a 5)
Formato Letture/Scritture
- 01 – le letture/scritture seguono lo schema 'solo LSB'
- 10 – le letture/scritture seguono lo schema 'solo MSB'
- 11 – le letture/scritture seguono lo schema 'LSB+MSB'
- 00 – in questo caso sto dando un 'counter latch command' (vedi seguito)

Selezione Contatore (da 0 a 2)
se vale 11, allora il significato degli altri bit cambia (vedi seguito)

Es. CWR ← 1011'0110
significa che il timer #2 deve effettuare un conteggio binario in modalità #3, le scritture/letture seguono lo schema LSB+MSB

Lettura del conteggio (1)

- Ci sono tre maniere per leggere il valore di un contatore
  1) lettura semplice
  2) Counter Latch Command
  3) Read-Back Command

1) La lettura semplice consiste nel leggere il valore di CRj previo bloccaggio del conteggio tramite il segnale GATEj

- Se non si blocca il conteggio il valore letto può essere inconsistente

2) Counter-Latch Command

CWR	c	c	0	0	-	-	-	-

- (cc=0, 1 o 2 indica il contatore, '-' indica non specificato)
- Lettura di 1 o 2 byte da CRcc (a seconda dello schema di lettura/scrittura precedentemente impostato)

3) Read-Back Command

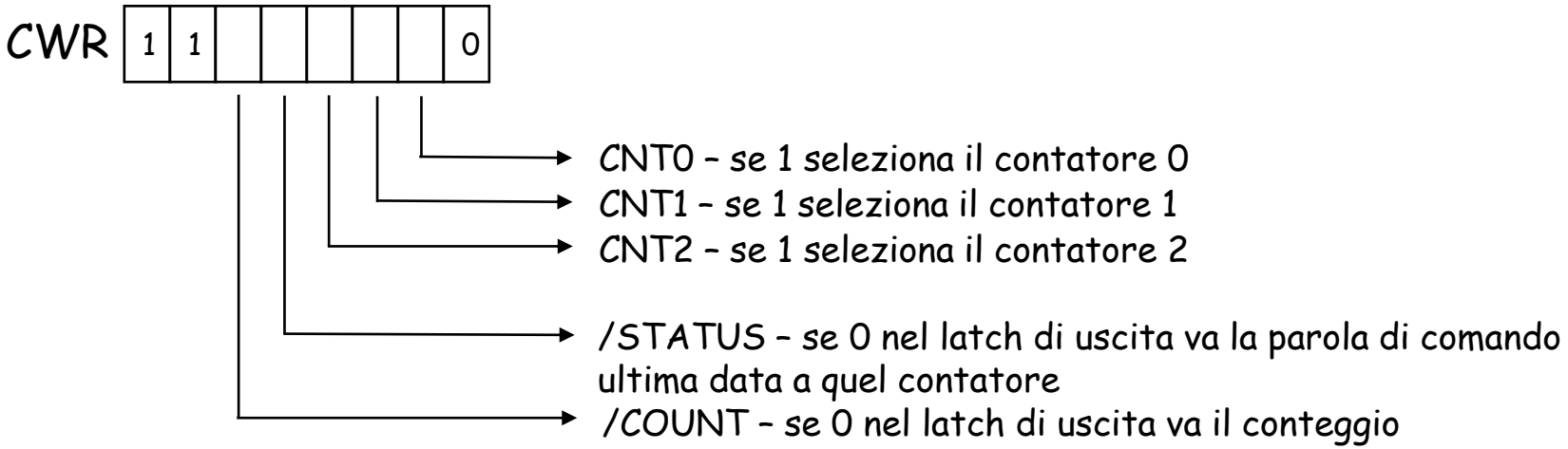
CWR	1	1						0

- V. slide successiva

Lettura del conteggio (2)

- **Read-Back Command**

  1) Si realizza in questo modo: scrittura in CWR delle seguenti info

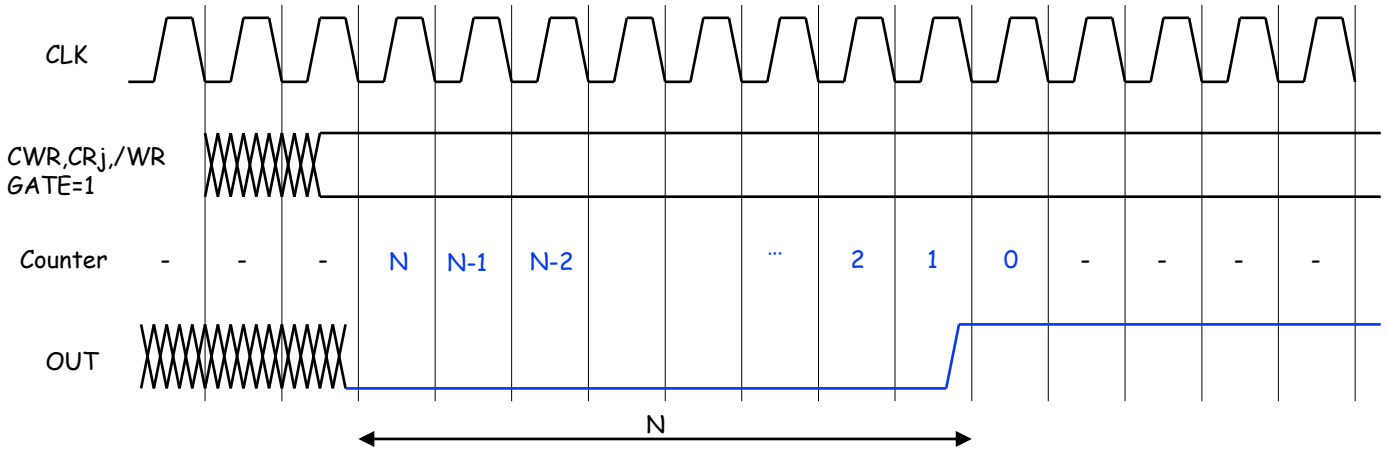


  2) Successiva lettura di CRx

- **Vantaggi:**
  - **Consente di leggere i conteggi di più contatori simultaneamente (caso di bit /COUNT attivo)**
  - **Consente di leggere come sono stati programmati i 3 contatori (caso di bit /STATUS attivo)**
    - **In particolare, i 6 bit meno significativi danno i valori precedentemente scritti in CWR per quel contatore**

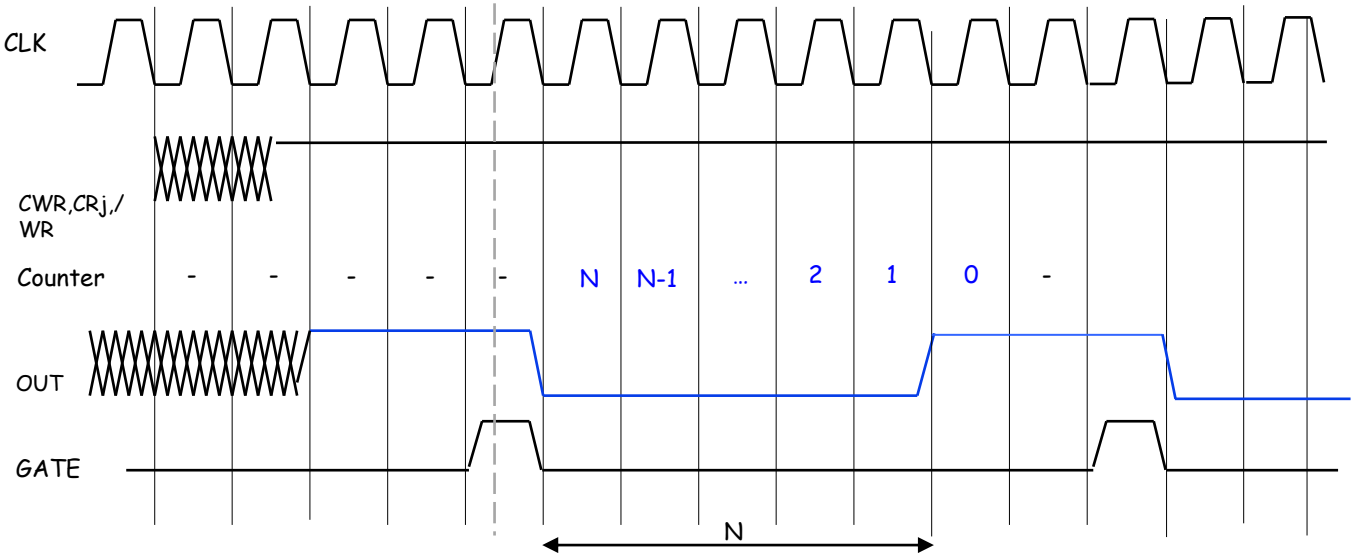
Modi per generare un gradino dopo il conteggio

- **Modo 0: conteggio abilitato dalla scrittura di CR**



Nota: usato ad esempio per generare un interrupt alla fine di un conteggio

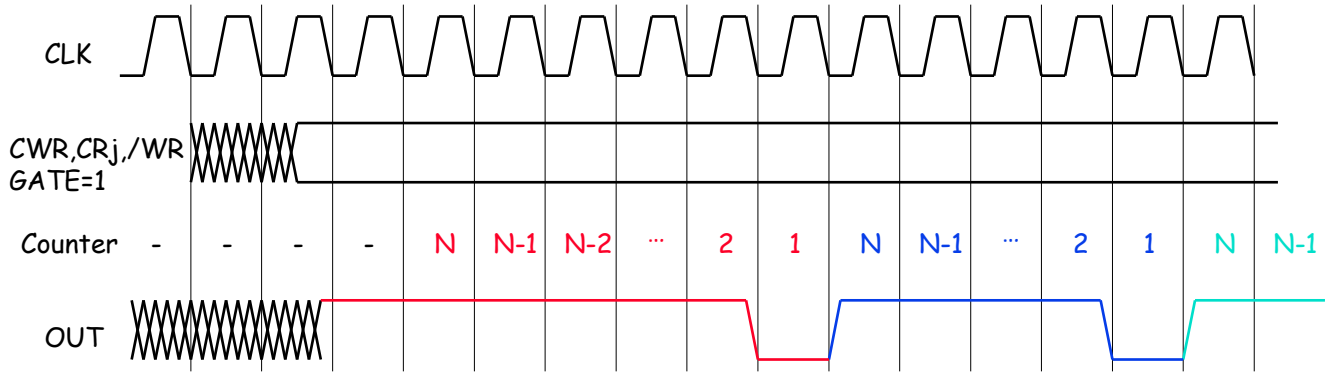
- **Modo 1: one shot (conteggio abilitato dal segnale hardware GATE)**



Nota: mentre OUT è agganciato ai fronti in discesa del clock (CLK), il segnale GATE viene viceversa campionato sui fronti in salita del clock. (per approfondimenti vedere manuale del chip 8254)

Modi per generare un'onda periodica

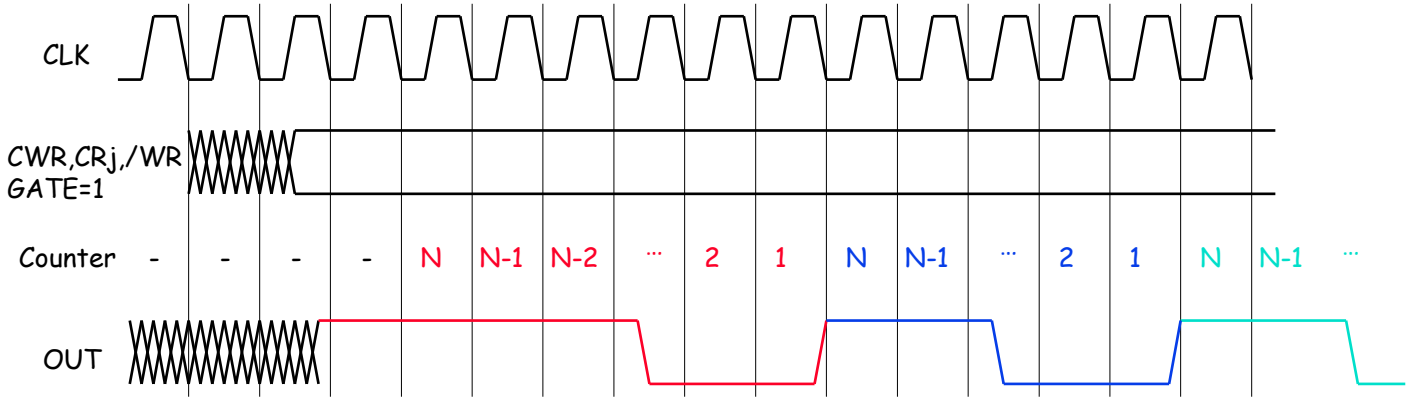
- **Modo 2: rate generator (divisore di frequenza)**
  → genera un onda quadra con duty-cycle pari a (N-1)/N



Nota1: Se f è la frequenza del segnale di clock (CLK), la forma d'onda d'uscita (OUT) avrà frequenza f/N

Nota2: L'uscita è inizialmente alta e diventa bassa, per la durata di un ciclo, quando il contatore raggiunge il valore 1

- **Modo 3: square wave mode (generatore d'onda quadra)**

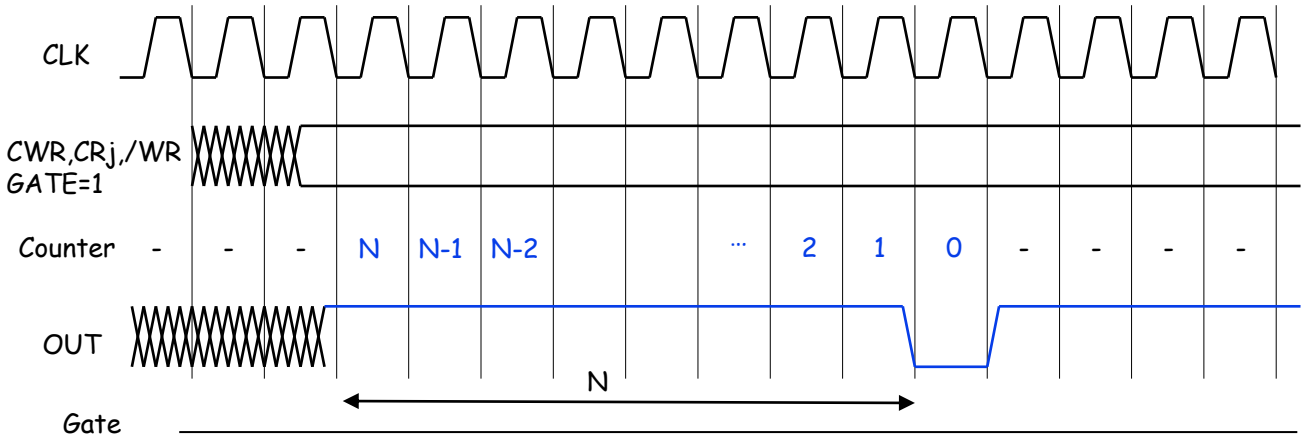


- **OUT rimane alto nei periodi N, N-1, … , (N+1)/2**
- **OUT rimane basso nei periodi N/2, (N-1)/2, … , 2, 1**

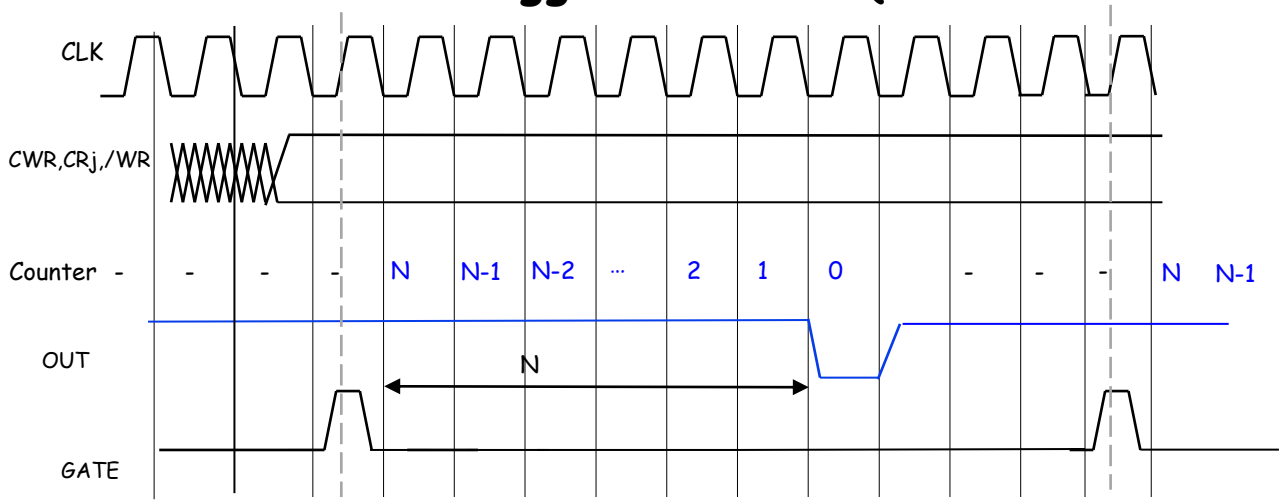
→ Il duty cycle è ½ se N è pari, altrimenti è ((N+1)/2)/N

Modi per generare un impulso a fine conteggio

- **Modo 4: software triggered strobe (abilitato alla scrittura di CR)**



- **Modo 5: Hardware triggered strobe (abilitato ad hardware da GATE)**



Nota: mentre OUT è agganciato ai fronti in discesa del clock (CLK), il segnale GATE viene viceversa campionato sui fronti in salita del clock. (per approfondimenti vedere manuale del chip 8254)

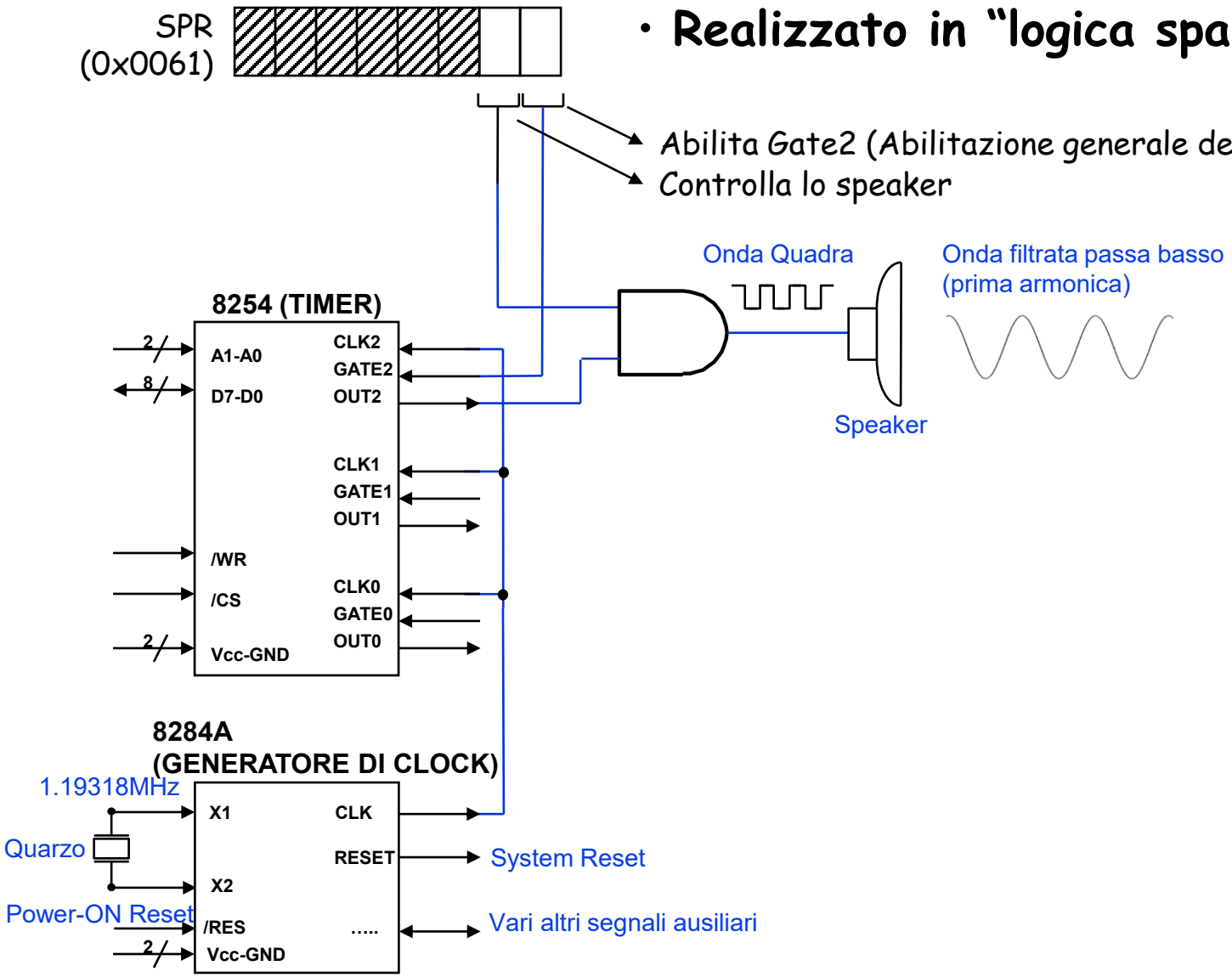
Riepilogo risorse 8254 nei Personal Computer

Chip name	Register name		I/O port	IRQ
Intel 8254	CR0	Counter Register 0	0x0040	IRQ0
Intel 8254	CR1	Counter Register 0	0x0041	-
Intel 8254	CR2	Counter Register 0	0x0042	-
Intel 8254	CWR	Comtrol Word Register	0x0043	-

Glue logic	SPR	Speaker Register	0x0061	

- **CLK0=CLK1=CLK2 = 1.19318MHZ**
- **TIMER-0: timer di sistema**
  - **GATE0 sempre attivo**
  - **OUT0 genera IRQ0**
- **TIMER-1: timer per il rinfresco della memoria**
  - **GATE1 sempre attivo**
  - **OUT1 genera impulsi con un periodo di 15ms per attivare il rinfresco della memoria**
- **TIMER-2: generatore di beep (v. slide successiva)**
  - **GATE2 controllabile tramite bit0 di SPR,Speaker Register (porta di I/O 0x0061)**
  - **OUT2 in AND col bit1 di SPR, l'uscita va poi sul beeper**

Speaker Register e generazione di suoni sullo speaker



Esempio di programmazione del Timer 8254

- **Generare una interruzione dopo Δt=50ms**
  - **Fc=1.19318 MHz (Tc=1/1.19318·10⁶ ≈ 838ns)**
  - **N=Fc·Δt=1.19318·10⁶·50·10-3 → N ≈ 59659**

```
int init_8254 (void)
{
    int cr_l, cr_h;

    outp(0x0043, 0x30);  /* contatore 0, costante di tempo a 16 bit,
                            modo 0, conteggio binario */

    cr_l = 59659 & 0x000000FF;
    cr_h = (59659 >> 8) & 0x000000FF;
    outp(0x0040, cr_l);
    outp(0x0040, cr_h);
}
```

→V. SLIDE 7

Lezione 9: Comunicazione Seriale Asincrona

- **Comunicazione**: ad un certo istante, è possibile individuare
  - Un dispositivo che agisce da trasmettitore (Initiator o TX)
  - Un dispositivo che agisce da ricevitore (Target o RX)

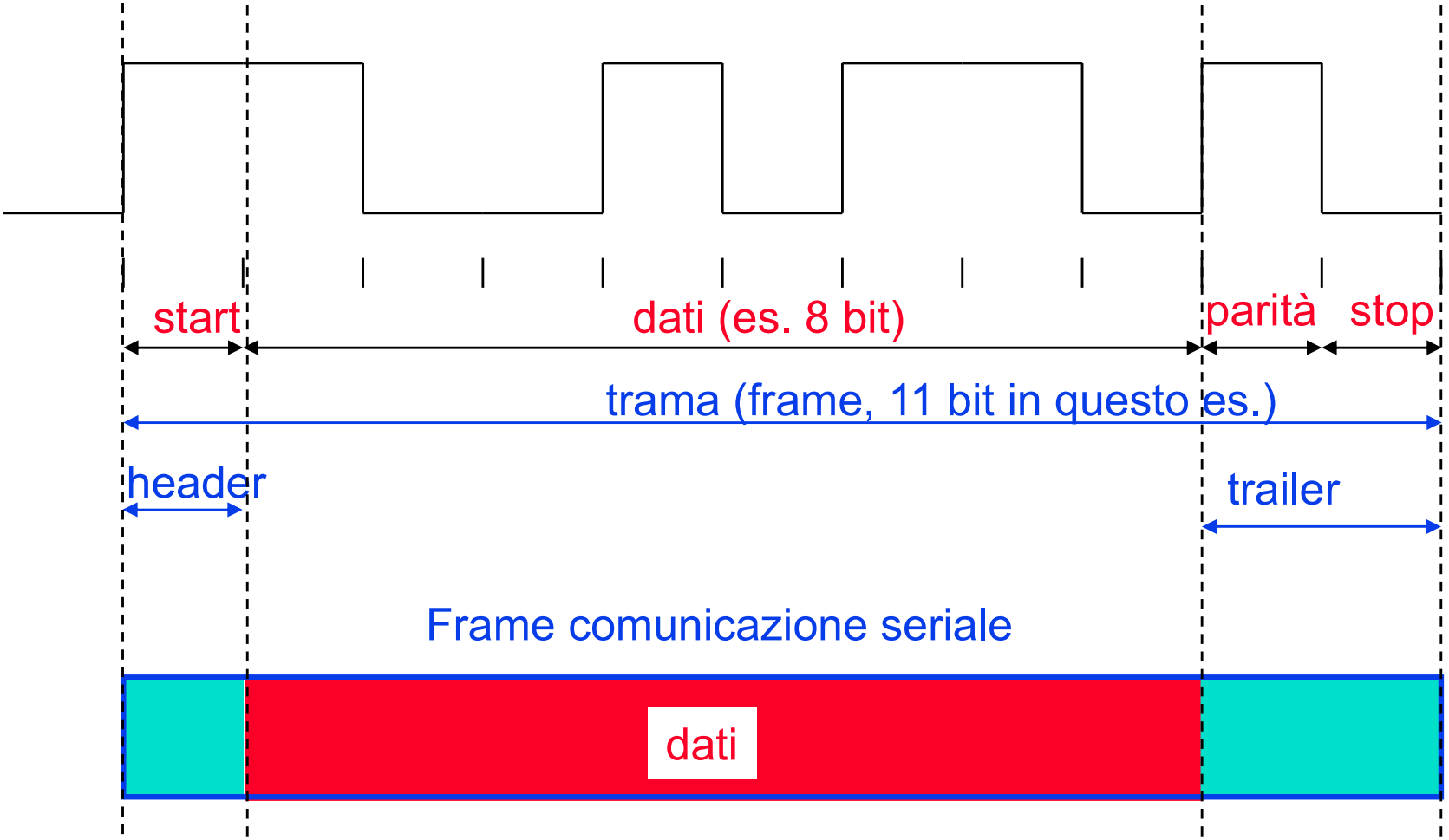
- **Seriale**: i bit che compongono il dato vengono trasmessi sul mezzo trasmissivo in sequenza
  - La trasmissione potrà essere anche wireless!
  - La trasmissione potrà usare 1 filo o più fili elettrici (modalità differenziale, cfr. Ethernet baseT)

- **Asincrona**: la trasmissione dei dati avviene seguendo un protocollo di comunicazione asincrono
  - Nessun segnale di clock è associato alla comunicazione

Caratteristiche principali di una ASC

- La velocità della comunicazione è caratterizzata dal numero di bit trasmessi per secondo o bit-rate
  - Se T è il tempo per trasmettere un bit, 1/T è il bit-rate
- Il dato è costituito da una trama (frame) di bit
  - L'informazione è tipicamente racchiusa tra un header o preambolo e un trailer o coda
  - L'header serve a marcare l'inizio della trasmissione
  - Il trailer serve a marcare la fine della trasmissione e a trasmettere un codice di verifica (CRC o parità)
- Ricevitore e trasmettitore si sincronizzano usando l'header
  - Il clock in ricezione cerca così di ottenere stessa fase del clock del trasmettitore: la frequenza deve però essere nota a priori!

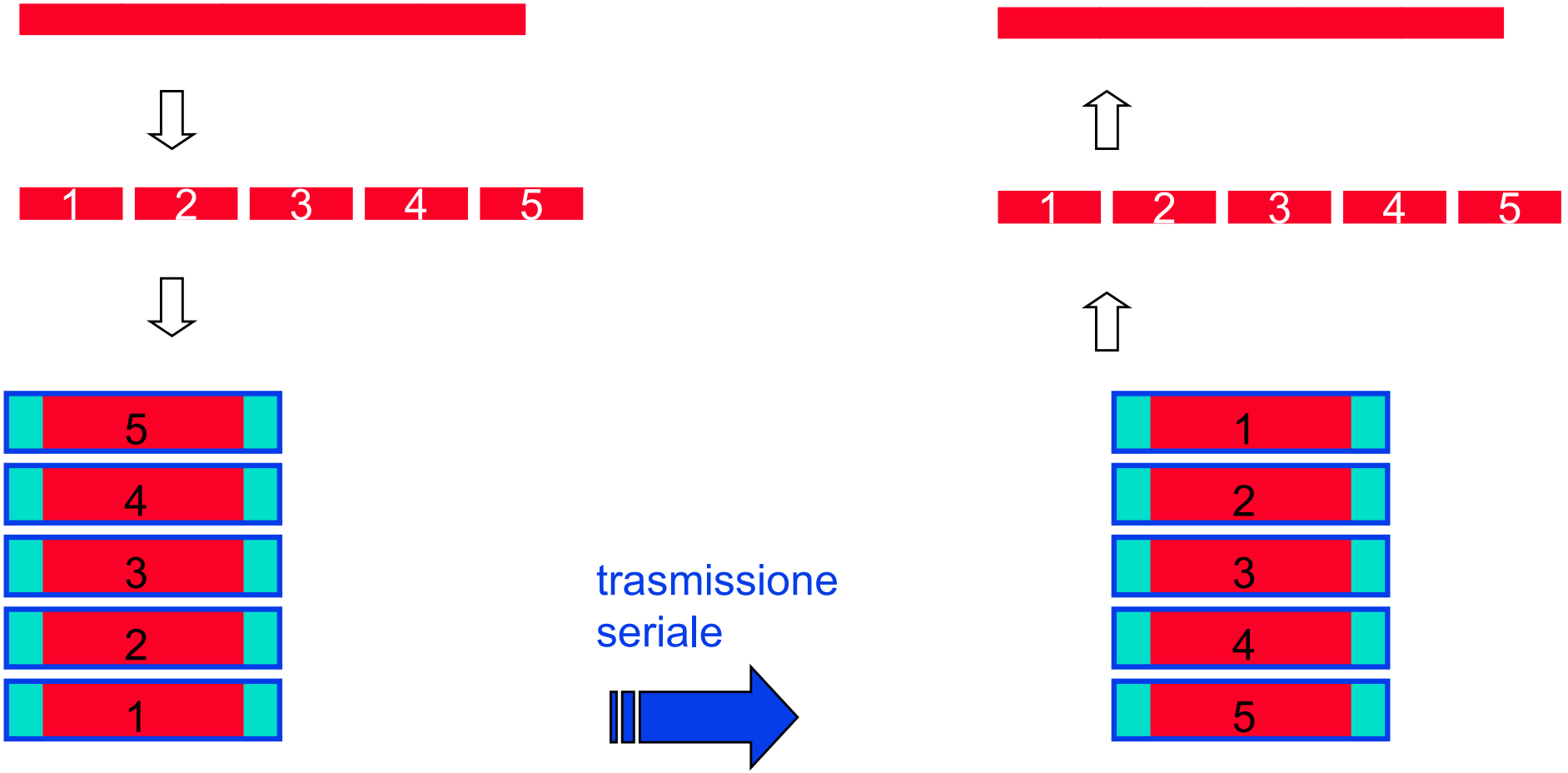
Trama seriale



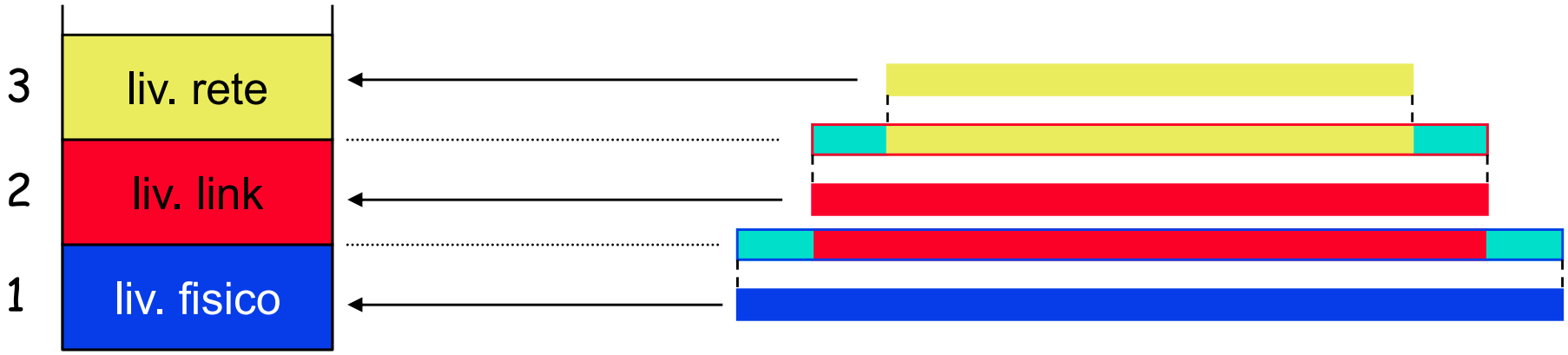
- **Informazioni che è necessario sapere a priori**
  - **Velocità di trasmissione (bit-rate, es. 1200 bit/s)**
  - **Formato della trama**
    (es. 1 bit di start, 8 bit dati, 1 bit di parità, parità positiva, 1 bit di stop)

Pacchettizzazione

Es. Trasmissione di 5 byte



Stratificazione (Stack di Comunicazione)



Stack di comunicazione:

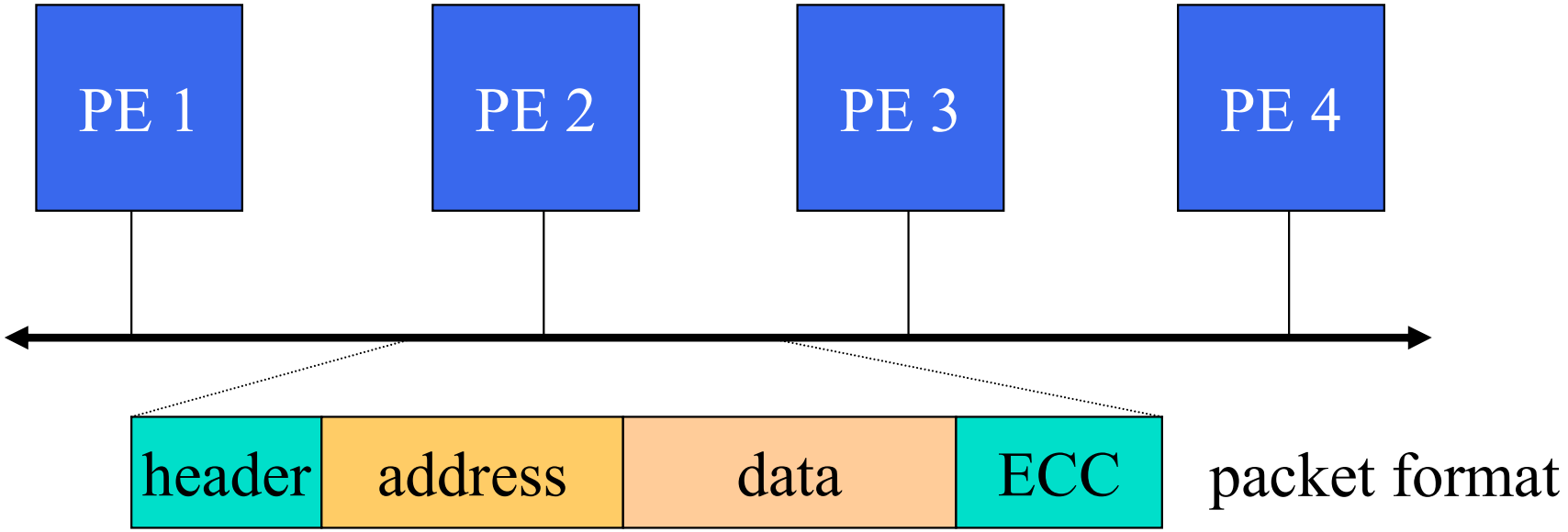
- verso l'alto, ogni livello fornisce una comunicazione con più proprietà
- verso il basso i dati vengono 'arricchiti' con ulteriori informazioni di controllo

Gestione delle reti per astrazioni successive

- International Standards Organization (ISO) ha sviluppato un modello per descrivere le reti chiamato Open Systems Interconnection (OSI)
  - Modello a 7 livelli divenuto standard noto come ISO/OSI
- Fornisce uno standard per classificare i componenti delle reti e le operazioni coinvolte

Livelli 1 & 2: Reti Seriali a Bus

- Una unica connessione fisica comune



- I2C, USB, FireWire, Ethernet, Wi-Fi…

→ Standard Ethernet – Livelli 1 & 2

- Risale ai primi anni '80
- Standardizzato come IEEE 802.3
- Dominante nelle reti locali (LAN)
- Versioni:
  - 10 Mb/s (IEEE 802.3, o Ethernet)
  - 100 Mb/s (IEEE 802.3u, o Fast Ethernet)
  - 1 Gb/s (IEEE 802.3ab, o Gigabit Ethernet)
  - 10 Gb/s (IEEE 802.3ae, o 10-Gigabit Ethernet)
  - 100 Gb/s (IEEE 802.3ba, o 100-Gigabit Ethernet)

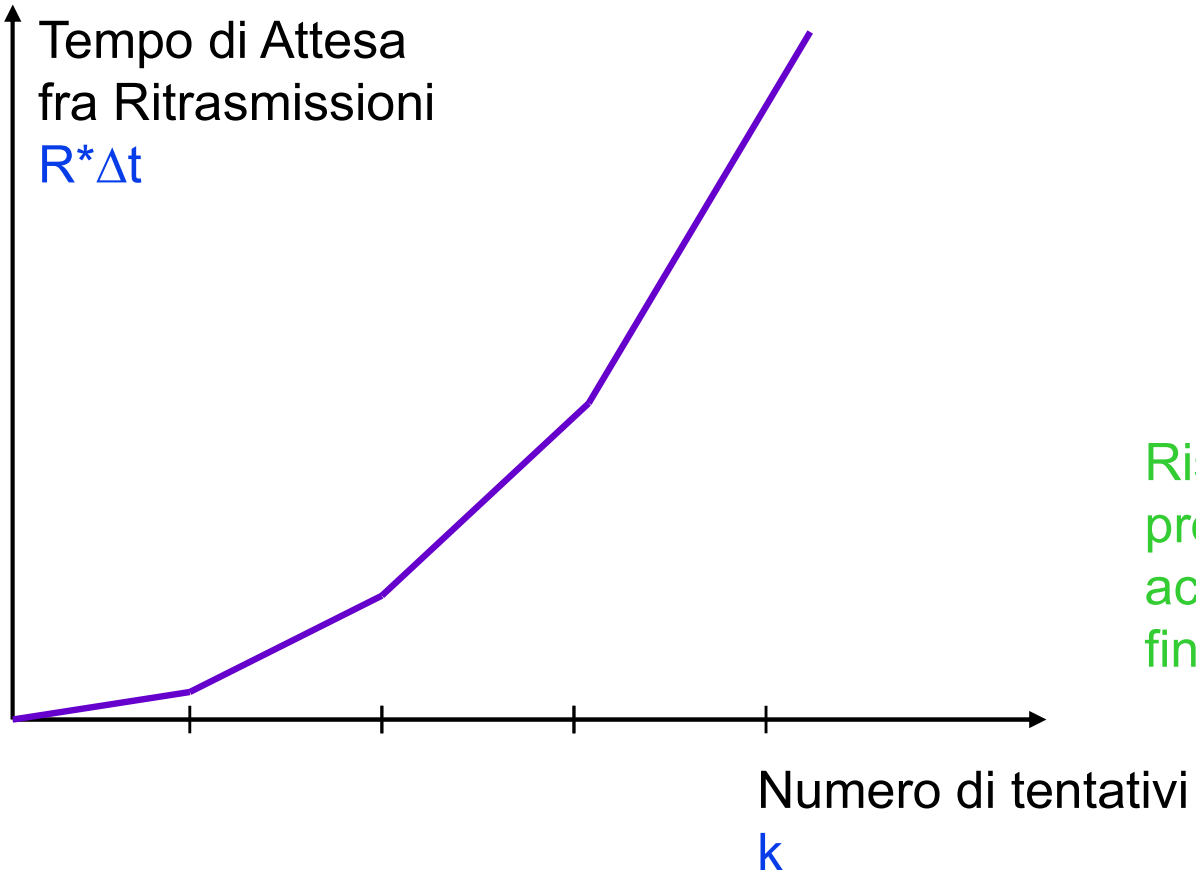
  - Wireless LAN(IEEE 802.11, o Wi-Fi)
  - Low-Power WLAN (IEEE 802.15, o Bluetooth)
- Topologia: "a bus"
- Obiettivo: ottenere una comunicazione affidabile usando un mezzo trasmissivo eventualmente non affidabile

CSMA/CD, Carrier Sense Multiple Access with Collision Detection

- Un nodo "ascolta prima di parlare" (carrier sense)
- Se il mezzo è libero, si attende un tempo "Defer Time"
- Se ci sono collisioni (multiple access) l'energia sul mezzo aumenta
- Un nodo cerca di rilevare collisioni durante la trasmissione (collision detect)
- Se un nodo rileva una collisione, continua a trasmettere 4-6 bytes per essere sicuro che la collisione venga rilevata dagli altri nodi e poi "lascia" il mezzo
- Se un nodo ha rilevato collisione, la trasmissione viene ripetuta dopo $R^*\Delta t$ per altri 15 tentativi
- Dopo 16 tentativi l'errore viene segnalato ai livelli superiori
- Δt è fisso mentre R viene calcolato secondo un "algoritmo di back-off"

Algoritmo di back-off per il calcolo di R

- Sia n l'n-esimo tentativo di trasmissione (n=1,2,…, 15)
- K= min(n,10)
- R è un numero casuale t.c. 0 <= R <= 2K



Risolve in maniera probabilisticamente accettabile i conflitti fino a 1024 nodi…

Formato del Pacchetto Ethernet

- In dettaglio
  - Preamble — 10101010101010… — 8 Bytes
  - Start frame — 11 — 8 Bytes
  - Source address — 6 Bytes — Ethernet FRAME
  - Destination address — 6 Bytes — Ethernet FRAME
  - Length — 2 Bytes — Ethernet FRAME
  - Data Payload — i dati veri e propri! — 46-1500 Bytes — Ethernet FRAME
  - Padding — Eventuali byte in coda al payload — 46-1500 Bytes — Ethernet FRAME
  - CRC — 4 Bytes — Ethernet FRAME

64-1518 Bytes

- Segue un IPG (Inter Packet Gap) di almeno 96 bit (12 bytes)…

Prestazioni della rete Ethernet

- La Qualità del servizio (Quality-of-Service, o QoS) tende a decrescere in maniera non-lineare ad alti livelli di carico
- Una conseguenza dell'algoritmo di back-off è che non è possibile stabilire il tempo massimo entro cui verrà trasmessa con successo una trama
  - Non si possono garantire deadline real-time!
  - Si possono ottenere livelli molto buoni di servizi purché si riesca a mantenere il carico ad un livello appropriato

Porte Seriali

- Metodo efficiente di comunicazione tra dispositivi
  - Tipicamente denominata UART (Universal Asynchronous Receiver and Transmitter)
  - Es. 16550° (es. Personal Computer), 16450, 8250, 8251A, MC68681
- Semplice porta seriale costituita da:
  - 2 registri a scorrimento (shift register)
    - Uno usato dal lato trasmettitore (Tx)
    - L'altro usato dal lato ricevente (Rx)
  - L'uscita del Tx di un dispositivo è collegata all'ingresso Rx dell'altro dispositivo
  - Trasmissione e ricezione guidate dallo stesso clock (generato localmente, non trasmesso!)
  - Quando Tx è vuoto o Rx è pieno (ricevuto 1 byte) viene generato un interrupt
- Nel caso di trasmissione seriale asincrona full-duplex
  - trasmissione contemporanea in entrambi i versi: i due dispositivi fungono sia da Tx che da Rx
  - Sono sufficienti 3 fili di collegamento (Tx, Rx, GND)
- Trasmissione seriale asincrona half-duplex
  - La trasmissione, in un dato istante, avviene in una sola direzione:
    uno dei due dispositivi fa da Tx l'altro da Rx (successivamente i ruoli si possono scambiare)
  - Sono sufficienti 2 fili (Tx, GND)

Uso di un buffer FIFO

- Utilizzo di un buffer FIFO per evitare la perdita di dati
  - in Rx:

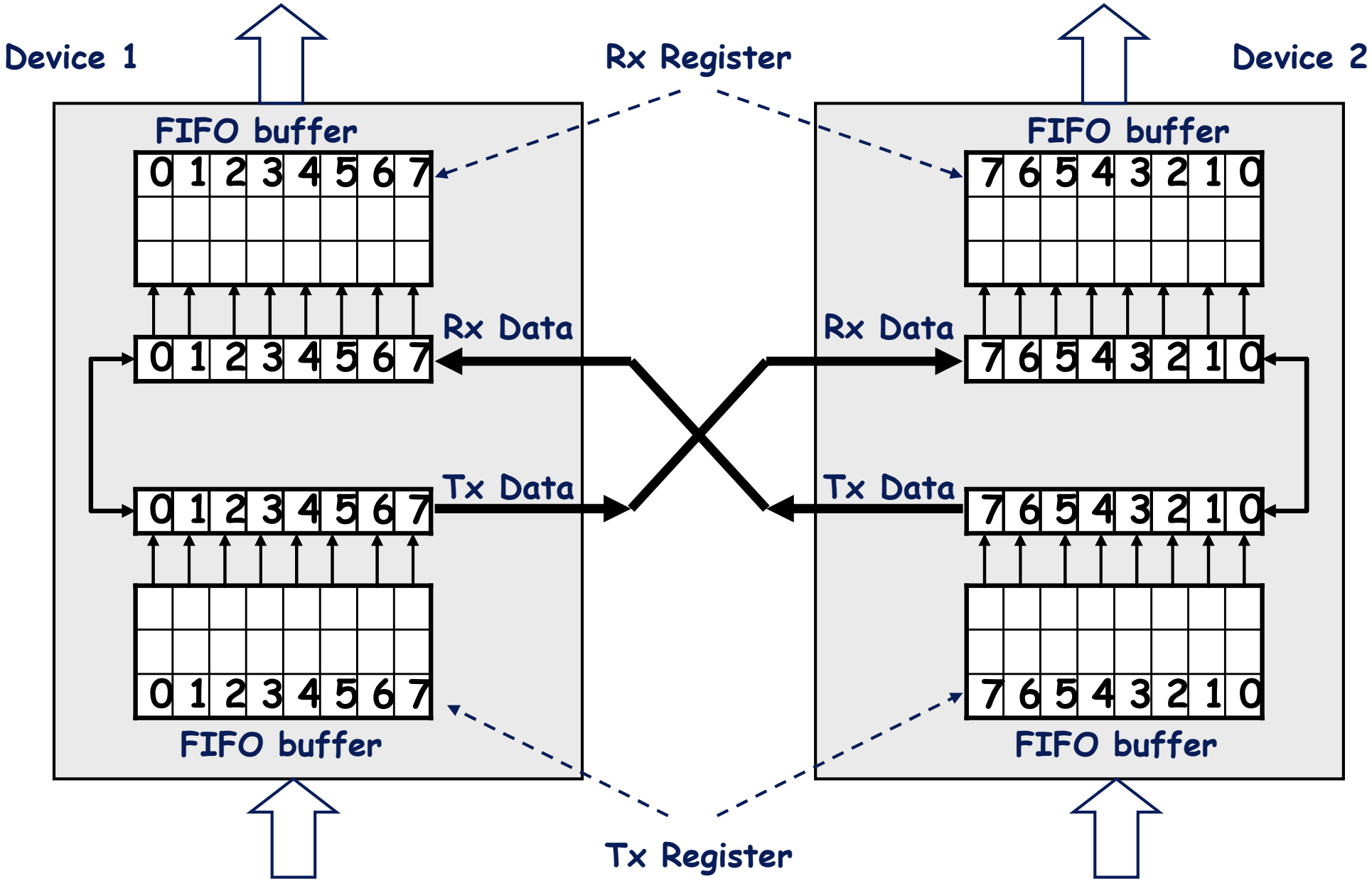
    dato ricevuto prima che sia stato letto quello precedente

  - in Tx:

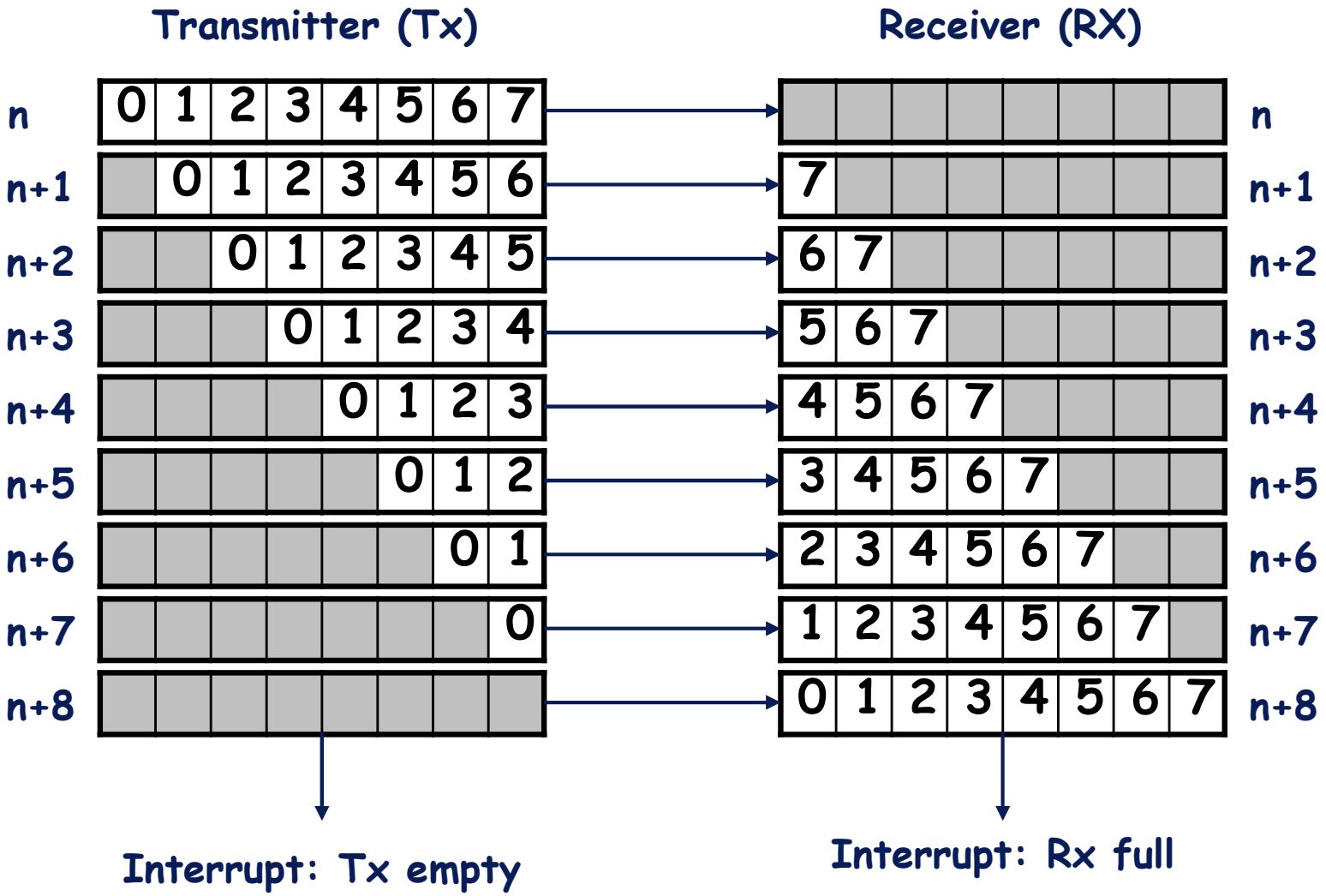
    dato inserito prima che sia stato trasmesso il precedente

- Dimensione del buffer FIFO:
  - Aumenta il throughput della porta
  - Riduce l'overhead della CPU (attesa della Tx o Rx)

Interfaccia Seriale Asincrona con Buffer FIFO



Es: trasmissione di un byte



Esempio di trasmissione su porta seriale

- Lato Tx:
  - Appena il dato è nel Tx-Register, viene trasmesso
  - Viene generato un interrupt per segnalare che il byte è stato trasmesso
- Lato Rx:
  - Il buffer riceve bit dopo bit il dato
  - Appena viene ricevuto un intero byte, viene trasferito nell'Rx-register
  - Viene generato un interrupt ed il byte viene letto
- In entrambi i casi
  - Viene messo a 1 un bit nel registro di stato (a fine trasmissione o ricezione)
  - Per poter proseguire, tale bit deve essere azzerato
    - Es. dalla routine di servizio

Controllo del flusso

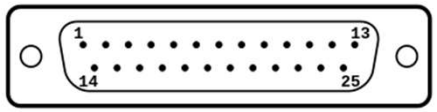
- Necessario per impedire che Tx invii più dati di quelli che Rx può leggere

- Due metodi
  - Hardware:
    - il chip UART del Rx individua un potenziale intasamento e attiva una linea apposita per avvertire il Tx
    - Tx interrompe la trasmissione
    - Appena Rx può ricevere dati, la linea viene disattivata
  - Software:
    - Il Rx invia caratteri speciali di controllo (XON e XOFF) al Tx
    - XOFF interrompe una trasmissione
    - XON riprende la trasmissione

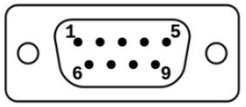
Standard EIA-RS232C – segnali e connettori

- **Segnali:**
  - In trasmissione, 1 logico=[-15V,-5V], 0 logico=[+5V,+15V]
  - In ricezione, 1 logico=[<-3V], 0 logico=[>+3V]
  - Il Tx deve vedere una capacità <2.5nF, una resistenza <7KOhm
  - Il Tx deve poter supportare un corto circuito con corrente <0.5A
  - Massima lunghezza cavo:
    per 20Kb/s →16m, per 9.6Kb/s →75m, per 1.2Kb/s →1000m
- **I connettori più comuni sono due:**
  - DB-25: connettore D-type a 25 pin
  - DB-9 : connettore D-type a 9 pin

DB-25	Sigla	Segnale	DB-9
1	GND	Chassis Ground	Not Used
2	TXD	Trasmit Data	3
3	RXD	Receive Data	2
4	RTS	Request To Send	7
5	CTS	Clear To Send	8
6	DSR	Data Set Ready	6
7	GND	Signal Ground	5
8	DCD	Data Carrier Detect	1
20	DTR	Data Terminal Ready	4
22	RI	Ring Indicator	9



DB25 - Connettore maschio*

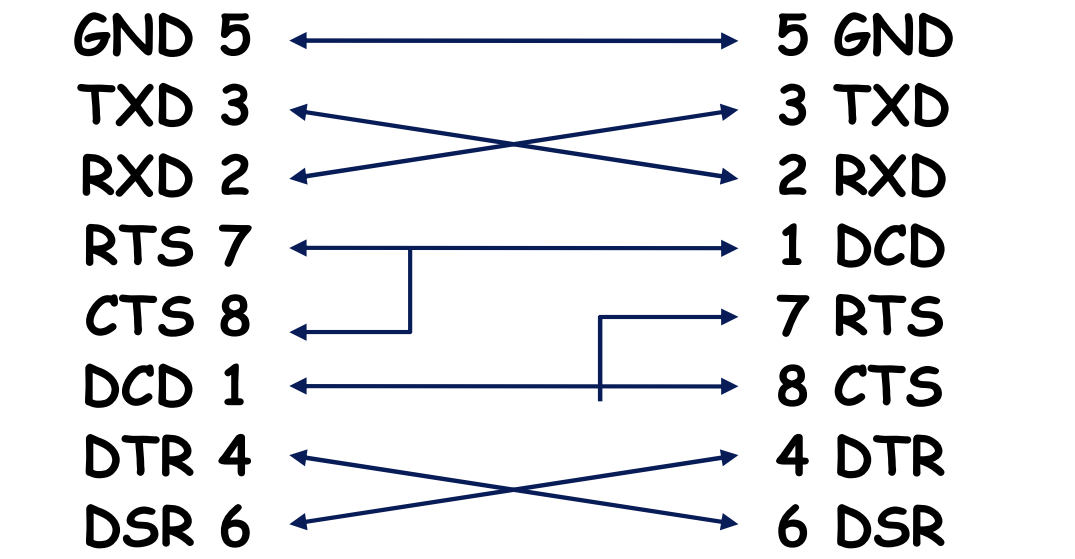


DB9 - Connettore maschio*

- **TXD (Trasmit Data):**
  linea di trasmissione dei dati connessa alla linea RXD del Rx
- **RXD (Receive Data):**
  linea di ricezione del dato, connessa alla linea TXD del Tx

Modem Cables

- Molte funzionalità di questi segnali derivano dal fatto che inizialmente furono pensati per connettere modem
- Per connessioni con altre periferiche, come stampanti, i segnali relativi ai modem devono essere "annullati" con l'uso di
  - cavi diretti oppure
  - cavi null-modem, es. per connettere due PC



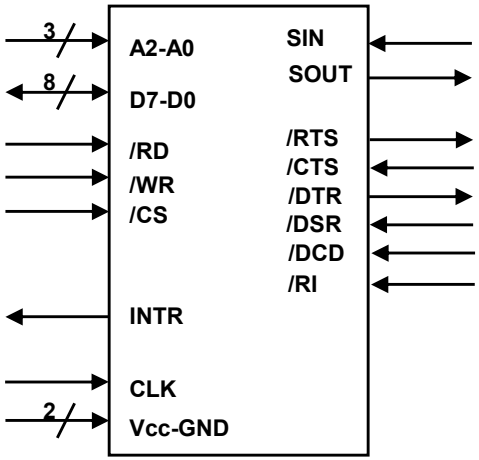
Connessione NULL-MODEM

Esempi di dispositivi di I/O: la Porta Seriale nei Personal Computer

- Tipicamente implementata con un chip tipo 16550A
  - Bit-rate: compreso fra 50 e 115000
  - Formato trama:
    - 1 bit di start
    - da 5 a 8 bit di dati
    - con o senza parità
    - 1, 1.5, 2 bit di stop
  - Riconoscimento dell'errore
    - Di parità (numero di 0 o 1 non corrispondente)
    - Di framing (bit di stop non ricevuti correttamente)
    - Di overrun (arrivo di un nuovo dato prima di aver prelevato il precedente)
  - Emissione/ricezione di break
    - Persistenza in stato attivo per un tempo più lungo della trama
  - Buffer dati da 16 byte
  - PORTE DI I/O ASSEGNATE e INTERRUPT (Risorse)
    - 0x03F8-0x3FF porta seriale #1 (COM1 o ttyS0) con IRQ4
    - 0x02F8-0x2FF porta seriale #2 (COM2 o ttyS1) con IRQ3
    - 0x03E8-0x3EF porta seriale #3 (COM3 o ttyS2) con IRQ4
    - 0x02E8-0x2EF porta seriale #4 (COM4 o ttyS3) con IRQ3
  - Interfacciamento RS-232 compatibile
    - poiché i pin SIN, SOUT, /RTS, /CTS, /DSR, /DCD, /RI sono a livelli TTL (0-5V), si rendono necessari dei chip 75188, 75189 per adattare i livelli di tensione a quelli richiesti dallo standard RS-232C

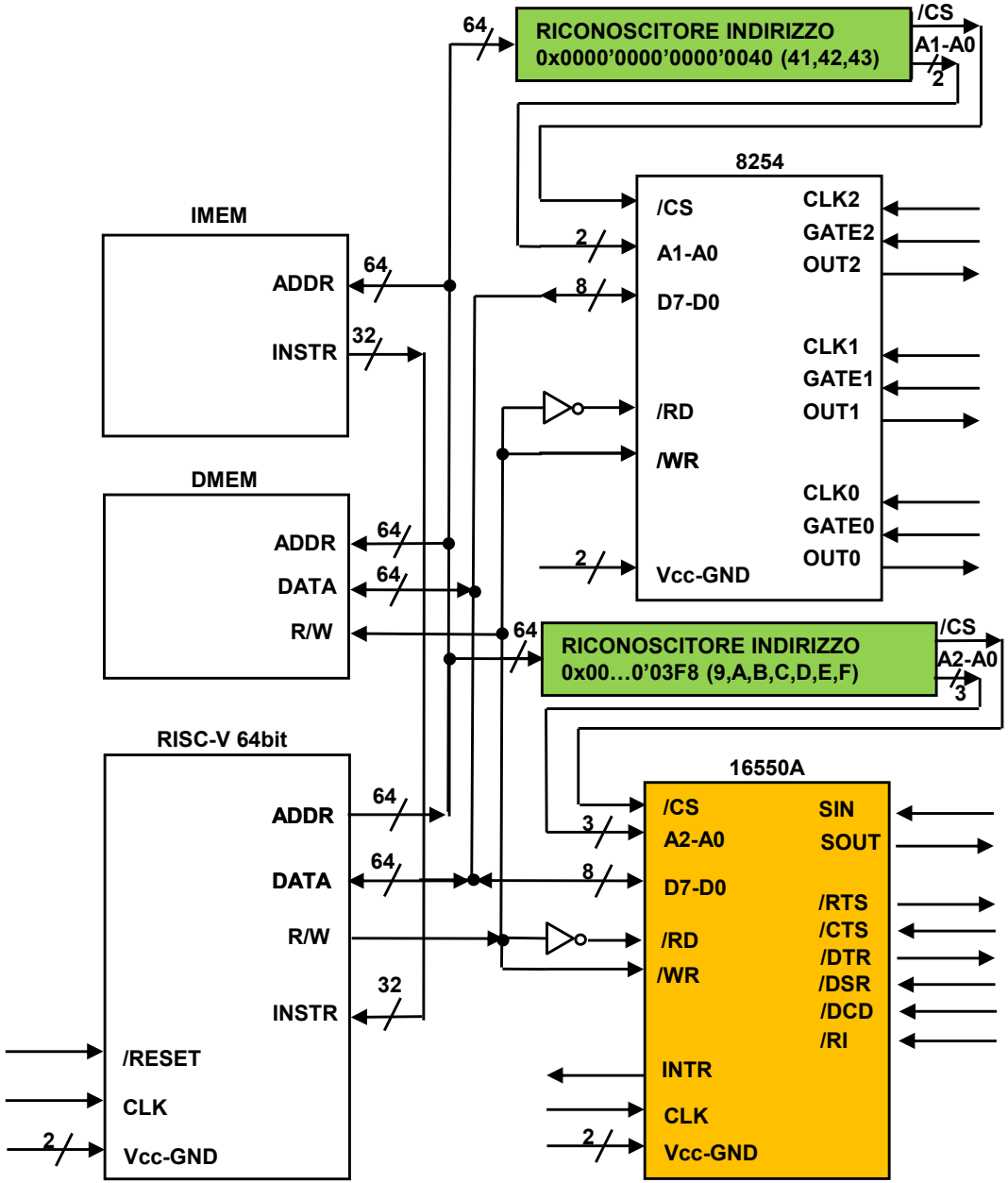
UART Intel-16550A

- Visione Elettrica del chip:

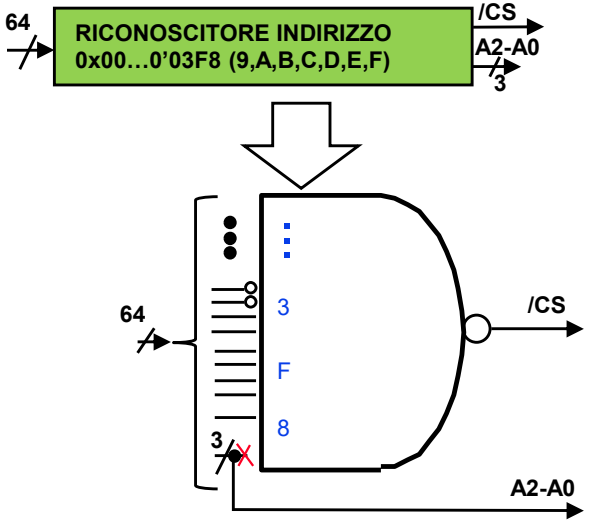


- A2-A0: indirizzamento
- D7-D0: scambio dati su 8 bit
- /RD, /WR, /CS: Read, Write, Chip-Select
- Vcc-GND: alimentazione
- CLK: clock locale per derivare le temporizzazioni
- SIN, SOUT: segnali Tx e Rx
- /RTS, /CTS, /DTR, /DSR, /DCD, /RI: segnali per l'interfacciamento secondo EIA-RS232C
- Altri pin possono essere presenti per usi generali e in dipendenza delle versioni specifiche del chip

UART Intel-16550A: visione di insieme



- /CS: Chip-Select
- A2-A0: indirizzamento
- D7-D0: scambio dati su 8 bit
- /RD, /WR: Read, Write
- Vcc-GND: alimentazione
- CLK: clock locale per derivare le temporizzazioni
- SIN, SOUT: segnali Tx e Rx
- /RTS, /CTS, /DTR, /DSR, /DCD, /RI: segnali per interfacciamento EIA-RS232C
- Altri pin possono essere presenti per usi generali e in dipendenza delle versioni specifiche del chip



UART Intel-16550A

- **Visione logica del chip (tutti registri a 8 bit):**

A2-A0	access	DLAB bit	mnemonic		
000	R	0	RBR		Receive Buffer Register Dove si trova il dato ricevuto
000	W	0	THR		Transmitter Holding Register Dove si mette il dato da trasmettere
001	RW	0	IER		Interrupt Enable Register
010	R	0	IIR		Interrupt Identification Register
010	W	0	FCR		FIFO Control Register
011	RW	-	LCR		**Line Control Register**
100	RW	-	MCR		Modem Control Register I bit 0 e 1 sono dei latch su /RTS e /DTS (in scrittura gli altri bit possono essere messi a 0)
101	RW	-	LSR		Line Status Register
110	RW	-	MSR		Modem Status Register
111	RW	-	SCR		Scratchpad Register Registro di appoggio da usare liberamente
001	RW	1	DLM		**Divisor Latch MSB**
000	RW	1	DLL		**Divisor Latch LSB**

Velocità di trasmissione: DLM e DLL

- Avendo posto a 1 il bit 7 di LCR (accesso ai Divisor Latches → DLAB=1) posso scrivere/leggere in DLM,DLL il valore della costante C che determina il bit-rate
  - DLM,DLL vengono interpretati come un intero senza segno a 16 bit
  - Detta FCK la frequenza del clock esterno e BR il bit-rate desiderato, si ha

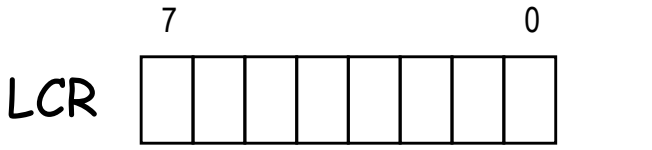
$$C = \frac{Fc}{16 \cdot BR}$$

Nota: nei PC Fc=1.8432MHz
Es. per ottenere un BR=9600
→C=12

BR	C
300	384
1200	96
4800	24
9600	12
115200	1

- Note su DLAB
  - DLAB non è altro che il bit 7 di LCR
  - I registri RBR, THR, IER, IIR, FCR si accedono ponendo DLAB=0
  - Per i restanti registri il valore di DLAB è ininfluente

Programmazione porta seriale: LCR, Line Control Register



n. di bit del dato 00→5, 01→6, 10→7, 11→8

n. di bit di stop: 0→1, 1→1.5 se il dato ha 5 bit, 2 altrimenti

1→bit di parità presente, 0→bit di parità assente

00→parità=#dispari di 1, 01→parità #pari di 1,
10→parità=#dispari a 0, 11→parità=#pari di 0

1→trasmetto un break (uscita a 0 logico)

Bit DLAB, se 1→posso leggere/scrivere la costante che determina il bit-rate

Esempio di programma che utilizza la porta seriale (1)

```
#include <sys/types.h>
#include <sys/stat.h>   /* open */
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>      /* stdin, stdout, stderr */
#include <sys/ioctl.h>
#include <termios.h>
#include <stdlib.h>
#include <sys/time.h>   /* timeval */
#include <stdlib.h>
#include <string.h>     /* memset used by FD_ZERO */

#define DEFAULT_DEV "/dev/ttyS0"
#define DEFAULT_SPEED 115200

int translate_speed(int spd);
void fd_setup(int fd, int speed, int flow);

int main(int argc, char **argv)
{
  int serfd, i, c=0, error=0, len, do_echo = 0;
  char txdata[127], rxdata[256];

  if (argc > 2) {
    printf("wrong syntax ! ./sertest [-e]"); exit(1);
  }

  if (argc == 2) {
    if ((strcmp("-e", argv[1]))==0) do_echo=1;
    else printf("wrong syntax ! ./sertest [-e]\n");
  }

  printf("Opening %s\n", DEFAULT_DEV);
  serfd = open(DEFAULT_DEV, O_RDWR);

  if (serfd < 0) exit(1);

  fd_setup(serfd, DEFAULT_SPEED, 1);

  for (i=0;i<127;i++) txdata[i]=i+64;
```

```
if (!do_echo)
  {
    while (!error){
      len = 0;
      i=0;
      printf("-%d- Sending data...", c++);
      write(serfd, txdata, 127);
      printf("done.\n");

      /* wait for 127 chars */
      while (len < 127)
      {
        len += read(serfd, rxdata+len, 127);
        printf("Got [%d/127]...\r", len);
        fflush(stdout);
      }

      /* now compare the received data */
      printf("Got response, comparing... ");

      if (memcmp(txdata, rxdata, 127) == 0)
        printf("OK.\n");
      else
        error = 1;
    }
  }
```

Esempio di programma che utilizza la porta seriale (2)

```
else
  {
    printf("Running as echo server.\n");
    while (!error) {
      len = 0;

      /* wait for 127 chars */
      while (len < 127)
      {
        len += read(serfd, rxdata+len, 127);
        printf("Got [%d/127]...\r", len);
        fflush(stdout);
      }

       /* now compare the received data */
      printf("-%d- Checking RX data... ", c++);

      if (memcmp(txdata, rxdata, 127) == 0)
      {
        printf("OK...echoing.\n");
        write(serfd, rxdata, 127);
      }
      else
      {
        printf("ERROR\n");
        error = 1;
      }
    }
  }

  printf("Not OK !\n");
  for (i=0;i<127;i++)
    printf("%c", txdata[i]);

  printf("\n");

  exit(1);
}
```

```
int translate_speed(int spd)
{
  int speed_c = 0;

  switch(spd) {
   case 9600:   speed_c = B9600; break;
   case 19200:  speed_c = B19200; break;
   case 38400:  speed_c = B38400; break;
   case 57600:  speed_c = B57600; break;
   case 115200: speed_c = B115200; break;
   case 230400: speed_c = B230400; break;
   case 460800: speed_c = B460800; break;
   default:     printf(
     "Bad baudrate %d is not valid.\n", spd); break;
  }
  return speed_c;
}

void fd_setup(int fd, int speed, int flow)
{
  struct termios t;

  if (fd < 0) { perror("fd_setup"); exit(1); }

  if (tcgetattr(fd, &t) < 0)
     { perror("tcgetattr"); exit(1); }

  cfmakeraw(&t);

  t.c_cflag &= ~CBAUD;
  t.c_cflag |= translate_speed(speed) | CS8 | CLOCAL;
  t.c_oflag = 0; /* turn off output processing */
  t.c_lflag = 0; /* no local modes */

  if (flow) t.c_cflag |= CRTSCTS;
  else t.c_cflag &= ~CRTSCTS;

  if (tcsetattr(fd, TCSANOW, &t) < 0)
     { perror("fd_setup : tcsetattr"); exit(1); }
  return;
}
```